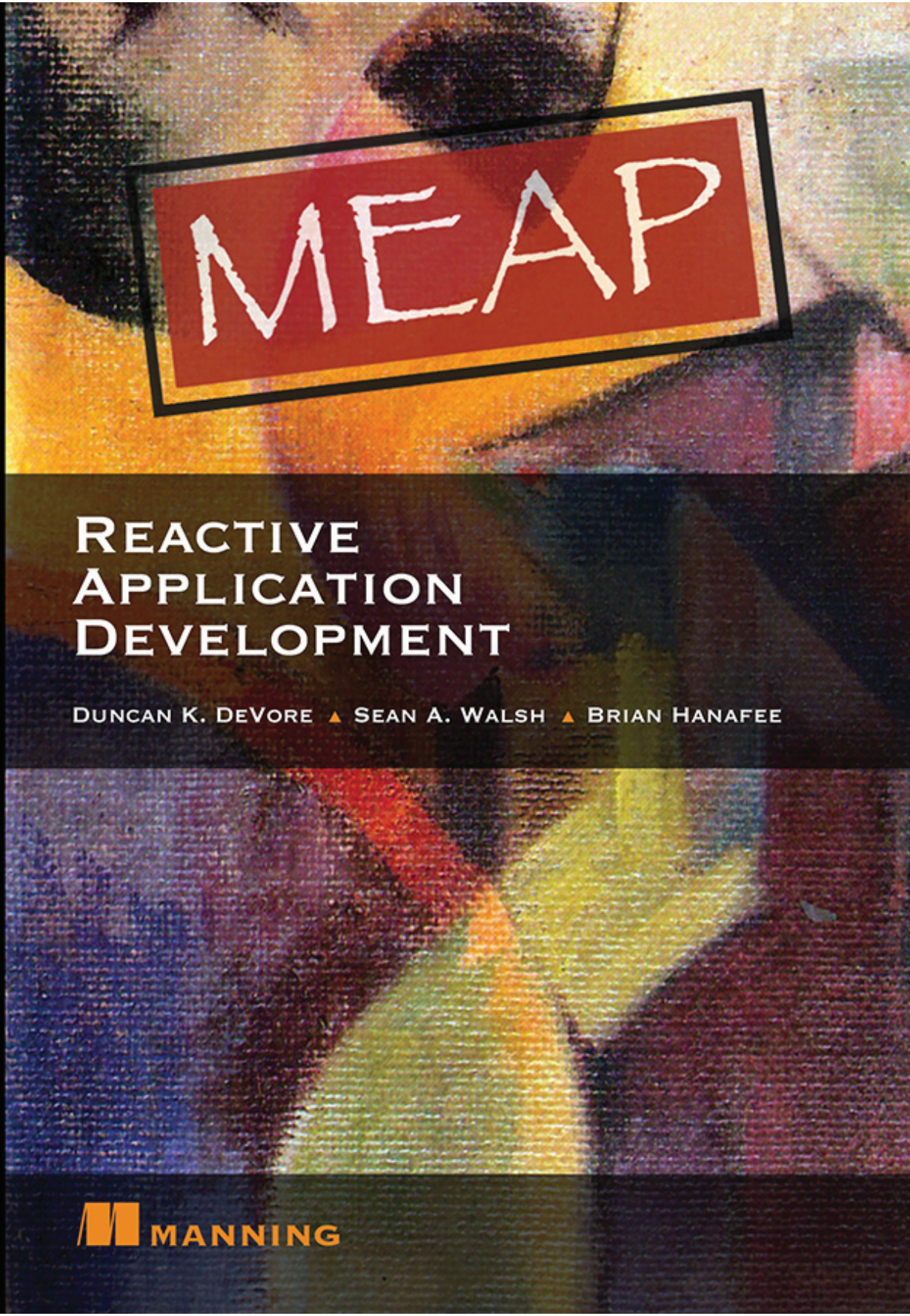
MEAP
Reactive Application Development
Duncan K. DeVore ▲ Sean A. Walsh ▲ Brian Hanafee
MANNING

MEAP Edition
Manning Early Access Program
Reactive Application Development
Version 9

Copyright 2017 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

welcome

Thank you for purchasing the MEAP for *Reactive Application Development.* We wrote this book so it is understandable by programmers and technical types, as well as business people, so everyone can understand the complexity and importance of properly designed reactive applications in an ever-changing technical landscape.

We've laid out the book in the order of our own discoveries over the past five years, which started with Scala/Akka and led to a complete redesign of large systems we needed to scale. Our process built on what we already knew about *Domain Driven Design* as a foundation that we then expanded upon with *Command Query Responsibility Segregation* and Event Sourcing *(CQRS/ES)*.

These tools, in combination with Akka, have allowed us to have *elastic, resilient, responsive and message-driven*--reactive applications. Your journey will mirror ours, and by the end of the book, you'll be able to identify domains, and distribute them as microservices, carved up in terms of CQRS/ES.

We're releasing the first two chapters to start.

- Chapter 1 is an introduction to reactive applications, how they've been built in the past, and how and why the landscape has changed, mandating a new way to build these typically large applications to serve an ever-growing user base.
- Chapter 2 is a primer on the Akka toolkit, which we will use throughout the book, with code examples usually also shown in Scala.

Looking ahead, the rest of Part 1 will cover using the Akka toolkit in distributed designs, domain-driven design, and an overview of Command Query Responsibility Segregation (CQRS). Part 2 will devote individual chapters to each of the pieces of CQRS, and also modularity. Part 3 will explores security, deployment, testing and finally the details of building a professional reactive application

Please take advantage of the Author Online forum. We will read your comments and continuously strive to improve the book, which we intend to be the blueprint for building reactive applications.

—Duncan Devore and Sean Walsh

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

brief contents

PART 1: FUNDAMENTALS

1 *What is a reactive application?*

2 *Getting started with Akka*

3 *What is Akka?*

4 *Akka basic toolkit*

5 *Domain driven design*

PART 2: BUILDING A REACTIVE APPLICATION

6 *Using Remote Actors*

7 *Akka clustering*

8 *CQRS and event sourcing*

9 *A reactive interface*

10 *Production readiness*

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

1

What is a Reactive Application?

This chapter covers

- The changing world of technology: impact on software
- Applications with massive user bases
- Traditional vs reactive: modeling complex, distributed software
- The Reactive Manifesto

One of the most fascinating things found in nature is the ability of a species to adapt to its changing environment. The canonical example of this is Britain's Peppered Moth. When newly industrialized Great Britain became polluted in the nineteenth century, slow-growing, light-colored lichens that covered trees died, and resulted in a blackening of the trees bark. The impact of this was quite profound: lightly-colored peppered moths, which historically had been well camouflaged and the majority, now found themselves the obvious target of many a hungry bird. Their rare dark-colored sibling, who had been conspicuous before, now blended into their recently polluted ecosystem. As the birds, changed from eating dark-colored to light-colored moths, the previously common light-colored moth became the minority, and the dynamics of Britain's moth population changed.

So what do moths have to do with programming? Well, moths in and of themselves are not particularly interesting in this regard, rather how they adapted to their environment is. The peppered moth was able to survive because of a genetic mutation that allowed it to react to its changing environment. This is exactly what a Reactive Application is: an application that reacts to its changing environment by design. It's constructed from the beginning to react to load, react to failure and react to users. This is achieved by the underlying notion of reacting to messages, but more on that later.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

With the ever-growing complexities of modern computing, we must be able to build applications which exhibit this trait. As user expectations of split-second performance, spikes in application load, demands to run on multi-core hardware for parallelism, and data needs expand into the petabytes, modern applications must embrace these changes by incorporating this behavior into their DNA. A reactive application embraces these challenges, as it is designed from the ground up to meet them head on.

While the peppered moth achieved its adaptation by way of a genetic mutation, reactive applications achieve it through a set of well-founded principles, patterns and programming techniques. The key for the peppered moth was DNA that included the basic building blocks for mutation. The same is true for reactive applications.

PRINCIPLES THAT DRIVE REACTIVE APPLICATIONS

Sound programming principles such as message driven, elasticity, resilience and responsiveness must be embedded in a reactive application's DNA from the beginning. Let's define each of these principles, which are contained in the Reactive Manifesto[1], essentially a blueprint for building reactive applications:

- Message Driven - Based on asynchronous communication where the design of sender and recipient are not affected by the means of message propagation. This means you can design your system in isolation without worrying about how the messages are transmitted. Message-driven communication leads to loosely coupled design that provides scalability, resilience and responsiveness.
- Elastic - Reacting to load; the system stays responsive under varying workload. Reactive applications can actively scale up or scale down based upon usage or other metrics utilized by system designers, saving money on unused computing power but most importantly ensuring the servicing of growing or spiking user base.
- Resilient - Reacting to failure; the system stays responsive in the face of failure. Failure is expected and embraced and since many systems exist in isolation, a single point of failure remains just that, the system responds appropriately with strategies for restarting or reprovisioning, seamless to the overall systems.
- Responsive - React to users; the system responds in a timely manner if at all possible. Responsiveness is the cornerstone of usability and utility, but more than that, responsiveness means that problems may be detected quickly and dealt with effectively.

The reader should note that Reactive Applications are not boilerplate applications and are challenging to build. They are designed to react to changes in their surrounding environment

[1] We'll go into more detail on the Reactive Manifesto throughout the book. Jonas Bonér, Dave Farley, Roland Kuhn and Martin Thompson have contributed to this blueprint for building reactive applications, which can be found here: http://www.reactivemanifesto.org/ .

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

without having to write new code, and that is a hefty task. Additionally they are based on a set of principles and techniques, which are not new but are just now becoming mainstream.

For example, many current applications on the JVM favor frameworks like Spring or Hibernate, while Reactive Applications tend to favor toolkits such as Akka, which is both a toolkit and runtime for building highly concurrent, distributed, and resilient, message-driven applications.

Don't let this new paradigm, with its use of robust toolkits like Akka give you pause however. This book will teach you a very different way of building applications, embracing the traits listed above, and will be able to solve the complex problems associated with distributed systems, concurrent programming, fault tolerance and more. This chapter will introduce you to the key principles of reactive applications we'll explore in the rest of the book.

1.1 Why do I need a reactive application?

Arguably one of the greatest inventions of mankind in the last 50 years has been Internet. It's origins date back to the 1960's, when the United States government commissioned research to build a robust, fault-tolerant computer network that began with a series of memos by J.C.R. Licklider of MIT in August, 1962, known as the "Galactic Network" concept. He envisioned a globally interconnected network of computers that would allow users to access data and programs from anywhere in the world. J.C.R. Licklider was the Director of the Information Processing Techniques Office (IPTO) within the Pentagon's ARPA, the Advanced Research Projects Agency which we know today as DARPA, the Defense Advanced Research Projects Agency.

Then in 1964, a professor from MIT, Leonard Kleinrock, published the first book on packet switching theory. Kleinrock convinced J.C.R. Licklider's successor, Lawrence G. Roberts, that the theory of communicating using packets rather than circuits was the next major step for networking computers. To explore this, Thomas Merrill and Lawrence Roberts, connected two computers together, the TX-2 computer in Massachusetts with the Q-32 in California via a low speed dial-up line. This significant event represented the first ever wide-area network and allowed time-sharing based computers to interchange data and run programs on a remote machine. This effort led to a DARPA funded RFQ in 1968 known as the ARPANET. The RFQ focused on the development of a key component, an interface for packet switches called Interface Message Protocol (IMP's) and was won by Frank Heart of Bolt Beranek and Newman (BBN) in December of 1968. Work by BBN and UCLA led to the first computer node coming online in September of 1969.

1.1.1 Distributed Computing

The result of this work in the 1960's and 70's by the United States, as well as some additional work from Great Britain and France, ultimately paved the way for what we know as the Internet today. Additionally, this resulted in a new computer model known as Distributed Systems which represented a shift in the computing paradigm. Prior to this the foundational

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

computer model was very large and very expensive mainframe systems affectionately referred to as "Big Iron."

Mainframes historically represent a centralized computing model that focuses on efficiency, local scalability and reliability. While this model is very effective, it is also very expensive and out of the reach of many companies with the cost of memory, storage units, processing cores, etc. reaching into the millions of dollars. As a result, many see Distributed Systems as a less expensive way of achieving and exceeding the raw computing power that typical mainframe configurations represent. That being said distributed systems do not preclude mainframes.

A distributed system consists of any number of computer configurations such as mainframes, minicomputers, personal computers, etc. The ultimate goal of Distributed Systems is to network a group of computers to work as a single system. We'll cover dealing with distributed systems throughout the book, with special focus in ch 3 and 4.

1.1.2 Cloud Computing

The advent of distributed systems with the continual progress towards more powerful and less expensive computing hardware paved the way to what we know today as cloud computing. Cloud computing represents another significant paradigm shift in how we write and manage computer applications.

While distributed systems focus on the technical details of how inter-connect independent computer systems, cloud computing focuses on the economics side of the equation. It represents a departure from the norm of managing, operating and developing IT systems that provide substantial economic savings as well as greater agility and flexibility, and this trend is here to stay.

In January of 2008 Amazon announced that Amazon Web Services now consume more bandwidth than their entire global network of retail services, as shown in the figure below.

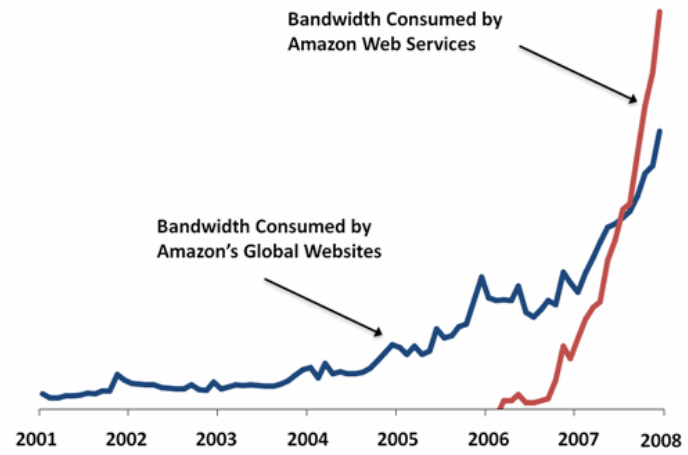

Figure 1.1 AWS Surpasses Amazon's Entire Global Retail Services Network in 18 Months

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

(http://aws.amazon.com/blogs/aws/lots-of-bits/)

This new landscape of distributed cloud computing represents a dramatic change for the modern programmer, much like the Industrial Revolution of the nineteenth century did for the Peppered moth. Recent hardware enhancement such as multi-core CPU's and multi socket servers provide computing capabilities that were non existent as little as 5 years ago.

Figure 1.2 shows the current state of storage, CPU, and bandwidth compared to the number of network nodes.

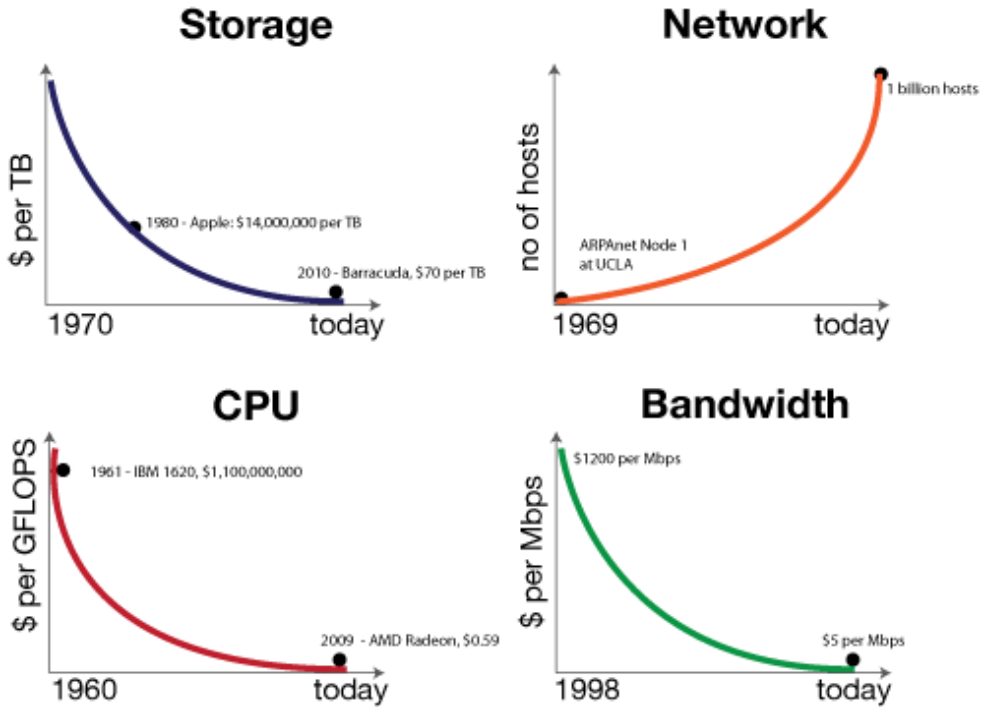

Figure 1.2 Decrease in costs of storage, CPU, and bandwidth with increase in network nodes (http://radar.oreilly.com/2011/08/building-data-startups.html)

As the figure illustrates, the decrease in the cost of storage, CPU cycles, and bandwidth coupled with an increase of network nodes, means it's shaping up to be a very competitive environment:[2] an environment the Reactive Paradigm has been designed to meet, because of

[2] Setting the stage: http://strata.oreilly.com/2011/08/building-data-startups.html

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

ready distribution across this vast ocean of processing power while maintaining resilience and responsiveness. We'll look more closely at cloud computing issues in ch 11, when we look at deployment.

The best way to understand the advantages of a reactive architecture over other approaches is through a comparison example. So let's do just that. We'll use a construct everyone is familiar with: the web shopping cart. We'll consider a simple example of a customer browsing online inventory, choosing items, and checking out, in both monolithic (applications where all layers are mutually dependent) and reactive architectures, and explore how each solves the complexities we've just explored, to show the stark differences between the two approaches and the notable advantages of a reactive solution.

1.2 Web shopping cart: complexity beneath the surface

Before we dig into the comparison, there are some things to consider about the shopping cart. On the surface, it seems like a simple use-case, but there is a lot more going on than meets the eye. The internet has opened a vast door, where the customer is king, and as a result, modern retailers have to fight for customers; they need to have an edge that draws their customer base back. To facilitate this, online sites craft a scenario where the shopper is browsing a catalog blithely, and tossing items into a cart; meanwhile, in the background, the application is busy checking inventory, pulling up reviews, finding images to display, and perhaps enticing the customer with a discount.

Each of these activities require interaction with other systems: managing responses and handling failure, while the shopper is none the wiser. In a traditional monolithic application these interactions will be slow or may fail entirely, because these applications are only as strong as their weakest link. You will see though that the reactive design paradigm deals with these challenges in an isolating and succinct manner, maximizing overall performance and dependability.

1.2.1 Monolithic architecture: difficult to distribute

Historically, since the dawn of web development, the majority of web applications have been designed based on a monolithic architecture. A *monolithic* architecture is an architecture where functionally discernible aspects of the system are not architecturally separate, as shown in Figure 1.3 below, which shows our shopping cart example as it might look in a monolithic architecture.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

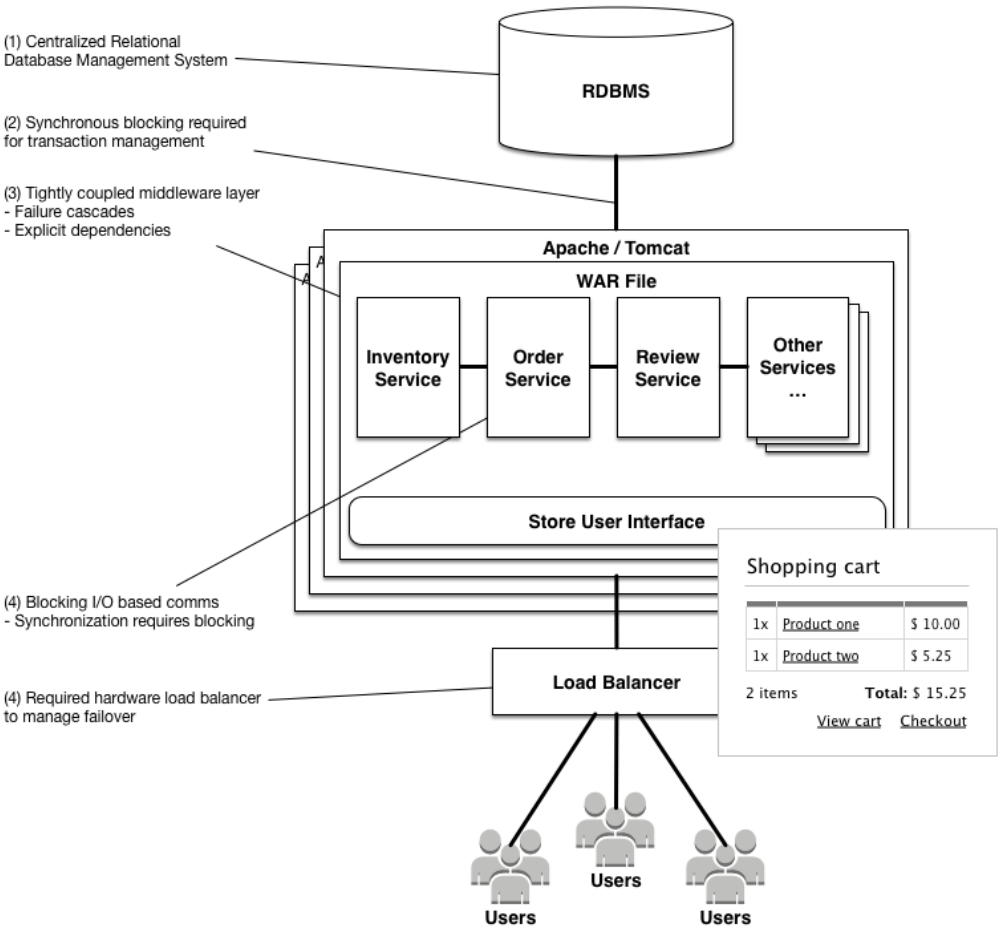

Figure 1.3 Shopping cart modeled in typical monolithic architecture style

As the figure shows, components such as data access, error handling, user interface, etc. are all tightly coupled. Blocking I/O is the norm, and for fault tolerance a hardware component is usually required.

CENTRALIZED RELATIONAL DATABASE MANAGEMENT SYSTEM

Looking at the monolithic architecture from the top down, you can see that it centers upon a centralized relational database management system. As a result, we encounter the the first of many challenges with this model.

The majority of relational databases today use synchronous blocking drivers for transaction management, which kills scalability because the components must all run in the same

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

application space such as a single JVM process, typically sharing a connection pool and not to mention, becoming a single point of failure. Optimization is commonly made been made over time using queueing technologies and ESBs, enterprise service buses to orchestrate processes and allow system to system communication and admittedly had provided measures of improvement since blocking at a service level is better than a blocked database transaction. However, this is not enough as any blocking is bad blocking and will always result in diminishing returns if not reduced returns as more hardware is thrown at the problem.

TIGHTLY COUPLED MIDDLEWARE

Next, we have a tightly coupled middleware layer comprised of several services that typically rely on blocking synchronous I/O based communication. This tight coupling and blocking I/O compromises scalability, but even worse, makes it very difficult to version our API. We cannot update only part or our API because of the interdependencies. More often than not, we have to reason about the entire system as a whole because of the rippling effect that tight coupling causes. The blocking nature of these communications can become a substantial bottleneck. One can imagine calls made out to the other service components to retrieve daily deals, reviews by other customers, images and so forth. In order to create the final composite view, we must wait until all associated data is retrieved which can result in a timely delay.

To solve some of these problems, more often than not we enter the dark world of concurrent programming. We attempt to spawn threads, write synchronized blocks, lock on our mutable state, use atomic variables and use the other tools provided us in a threaded environment. While threaded programming provides a useful abstraction for concurrent or parallel execution, the price becomes unsustainable. We begin to experience significant problem being able to understand our code, predict or determine what it's supposed to do, and as a result our code becomes widely non-deterministic.

> *Edward A. Lee Sums this up very nicely:*
> *Although threads seem to be a small step from sequential computation, in fact, they represent a* ***huge step****. They discard the most essential and appealing properties of sequential computation:* ***understandability, predictability****, and* ***determinism****. Threads, as a model of computation, are wildly non-deterministic, and the job of the programmer becomes one of* ***pruning*** *that* ***non-determinism****.*[3]

These services, in a monolithic application, share a common domain model, usually built on top of an ORM (object relational mapping) abstraction layer, that implements a CRUD (create, read, update and delete) to manage the domain's current state, and (as we will see shortly) this CRUD pattern can significantly diminish the value of our data. Traditionally, these services

[3] The Problem with Threads, Edward A. Lee, Berkeley 2006

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

are designed with strict dependencies on one another, and rely on blocking I/O for communication.

LOAD BALANCER

Finally, to support fault tolerance and load spikes, we must implement a load balancer; shown at the bottom of the figure. While load balancers help mitigate load spikes to some degree, they don't address the underlying problem in that they fail/retry at the entire operation level rather than just the failing part. In order for the architecture to be truly fault tolerant, they must be resilient by accepting failure and healing themselves at runtime. This notion of resilience needs to be baked into the architecture from the beginning and cannot be bolted on as an afterthought. Another technique that is commonly used is server clustering. The challenge with this approach is its extremely costly and can open us up to the ominous cascading failure scenario that brings down the entire cluster.

All of this IO blocking at all levels, is proven to be a very bad thing and must be avoided at all costs as dictated by Gunther and Amdahl's laws, which we'll explain next.

UNIVERSAL SCALABILITY LAW

It's been proven that blocking of any kind, anywhere in the system will measurably impact scale due to:

1. *Contention;* waiting for queues or shared resources.
2. *Coherency*; the delay for data to become consistent

Originally this was first shown by computer architect Gene Amdahl who theorized that blocking will cause diminishing returns with regards to scaling, which became known as *Amdahl's Law*. Neil J. Gunther, a computer system researcher further proved further that the blocking would actually reduce concurrency as a system is scaled, this holds true today and is called *Gunther's Law* but is widely known as *The Universal Scalability Law*. The reason for this is point 2 from the above in that the cost of the coherency as the system grows becomes a drag on the overall system and actually causes a loss over overall scale, as figure 1.4 shows below.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

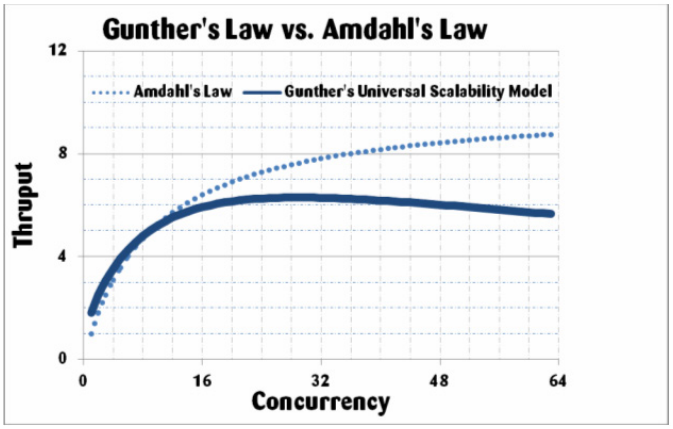

Figure 1.4 Gunther and Amdahl's laws (http://cmg.org/publications/measureit/2007)

The figure overlays both Gunther and Amdahl's laws, and clearly shows that while Amdahl *theorized* diminishing returns, Gunther proved that concurrency and therefore scale will actually drop off after a point. This means that no matter how much hardware you throw at the problem, you will actually make a blocking system *worse*.

Consistency is another topic often taken for granted when designing traditional monolithic systems, as you have tightly coupled services connected to a centralized database. These systems default to strong consistency, because access to data in terms of both reads and writes are guaranteed to be *consistently ordered*, meaning every single read must follow the last right and vice versa. This consistency model is great as far as always having single point access to the latest data, but has a very high cost in terms of distribution as identified by the CAP theorem.

CONSISTENCY AND CAP THEOREM

In *Theoretical Computer Science*, CAP Theorem, also known as Brewer's Theorem, states that its impossible in Distributed Systems to simultaneously provide all three of the following guarantees:

- Consistency: all nodes see the same data at the same time
- Availability: a guarantee that every request receives a response about whether successful or not
- Partition Tolerance: the system continues to function regardless of message failure or partial system failure

Figure 1.5 shows a Venn diagram of the overlap of these three guarantees.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

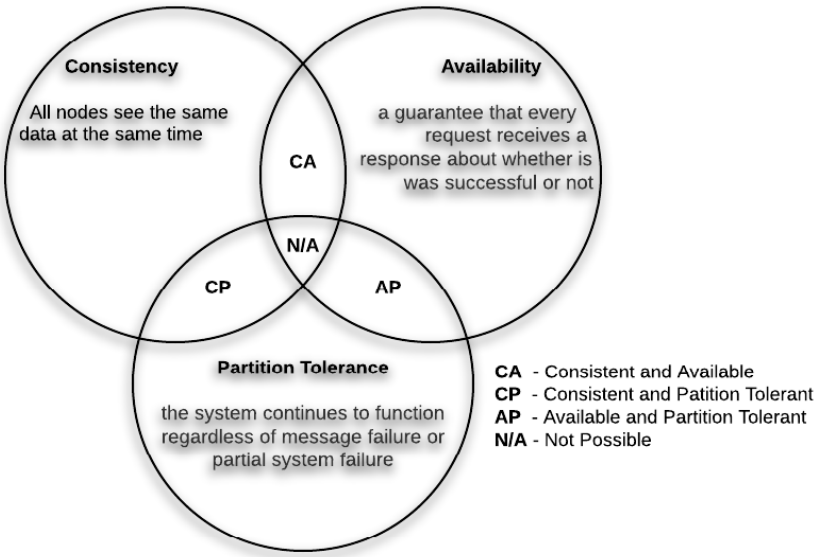

Figure 1.5 The CAP Theorem Venn diagram

As the figure shows, in distributed computing it's not possible to have all of your cake and to eat it too. By design, distributed systems are asynchronous and loosely coupled and rely on patterns such as atomic shared memory systems, distributed data stores, and consistency models to achieve availability and partition tolerance. A properly designed system must have partition tolerance so the decision must be made to either have higher availability or greater consistency.

CONSISTENCY MODEL

In distributed computing, a system supports a given consistency model if operations follow specific rules as identified by the model. The model specifies a contractual agreement between the programmer and the system, wherein the system guarantees that if the rules are followed, data will be consistent and the results will be predictable.

- Strong Consistency

  Strong consistency or *linearizability*, is the strongest method of consistency and guarantees that data reads are guaranteed to reflect the very latest writes, across all processes. Strong consistency is incredibly expensive in terms of scale and must therefore be avoided at all costs in building reactive applications.

- Eventual Consistency

  Eventual consistency is a consistency model used in distributed computing that informally guarantees that, if no new updates are made to a given data item,

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

eventually all accesses to that item will return the last updated value. Eventual consistency is a pillar of modern distributed systems, often under the moniker of optimistic replication, and has origins in early mobile computing projects. A system that has achieved eventual consistency is often said to have converged, or achieved replica convergence. Eventually Consistent services are often classified as as Basically Available Soft state Eventual consistency semantics as opposed to a more traditional ACID (Atomicity, Consistency, Isolation, Durability) guarantees. This is a key point as it is one of the key factors that allow distribution.

- Causal Consistency

  Causal consistency is a stronger consistency model that ensures that the operations processes in the order expected. Causal consistency is the strongest method of consistency achievable while retaining availability. More precisely, partial order over operations is enforced through metadata. For example. If operation A occurs before operation B, then any data store that sees operation B must see operation A first. There are three rules that define potential causality:

- Thread of Execution: If A and B are two operations in a single thread of execution, then A -> B if operation A happens before B.
- Reads-From: If A is a write operation and B is a read operation that returns the value written by A, then A -> B.
- Transitivity: For operations A, B, and C, if A -> B and B -> C, then A -> C. Thus the causal relationship between operations is the transitive closure of the first two rules.

Causal consistency is stronger than eventual consistency because it ensures that these operations appear in order; causal consistency is difficult to achieve in a distributed system because of multiple distributed parties in any "transaction."

Even Akka, as you'll see in our reactive architecture model of the shopping cart up next, does not have an out-of-the-box implementation of causal consistency, so the burden is on the programmer to implement it. The most common way to implement causal consistency in an Akka-based actor model is through Become/Unbecome, via the Process Manager pattern, which we'll talk more about in Chapter 2.

Why Akka?

If one looks at building system in terms of building a house it becomes very clear that the tools are of utmost important for guaranteed success. A builder craftsman accumulates tools over the years, always affording more and better tools proven to get the job done. Software craftsmen do the same thing in that we learn and constantly are given access to new technologies to build our software. Akka is an important item in our toolkit and is a runtime and software library for building highly concurrent, distributed, resilient and message-driven applications on the JVM; Akka is by nature reactive. At its heart, Akka relies on a mathematical model of concurrent computation, known as the "actor model." In this model, the "actor" provides a lightweight programming construct that:

- **sends and receives messages,**
- **makes local decisions,**

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

- and creates new actors,
- all asynchronously without locks.

Akka's Value Proposition

A single, unified programming model for:

- simpler concurrency - write code with the illusion of single-threadedness without locks, synchronized or atomic variables.
- simpler distribution - distributed by default, remove or local configuration.
- simpler fault tolerance - decouples communication from failure through supervision.

As a result of the challenges of concurrency non-determinism, consistency guarantees, and other rapid technology changes such as multi-core processors, and pay-as-you-go cloud services, we've learned that monolithic architectures don't translate well to the modern world of distributed computing. Problems around concurrency, transaction management, scalability and fault tolerance are rampant as we've seen above.

Next, let's look at reactive architectures and see how they fare against these challenges.

1.2.2 Reactive architecture: distributable by default

Reactive applications adopt a radically different approach than monolithic ones. Rather than building an architecture based on a non-distributed environment and then trying to retrofit it with locks for concurrency, load balancers, etc., reactive applications assume a distributed environment.

From the ground up they bake in the four key traits explained earlier: message driven, elastic, resilient and responsive; as the next figure shows, with its depiction of the shopping cart in a reactive application.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

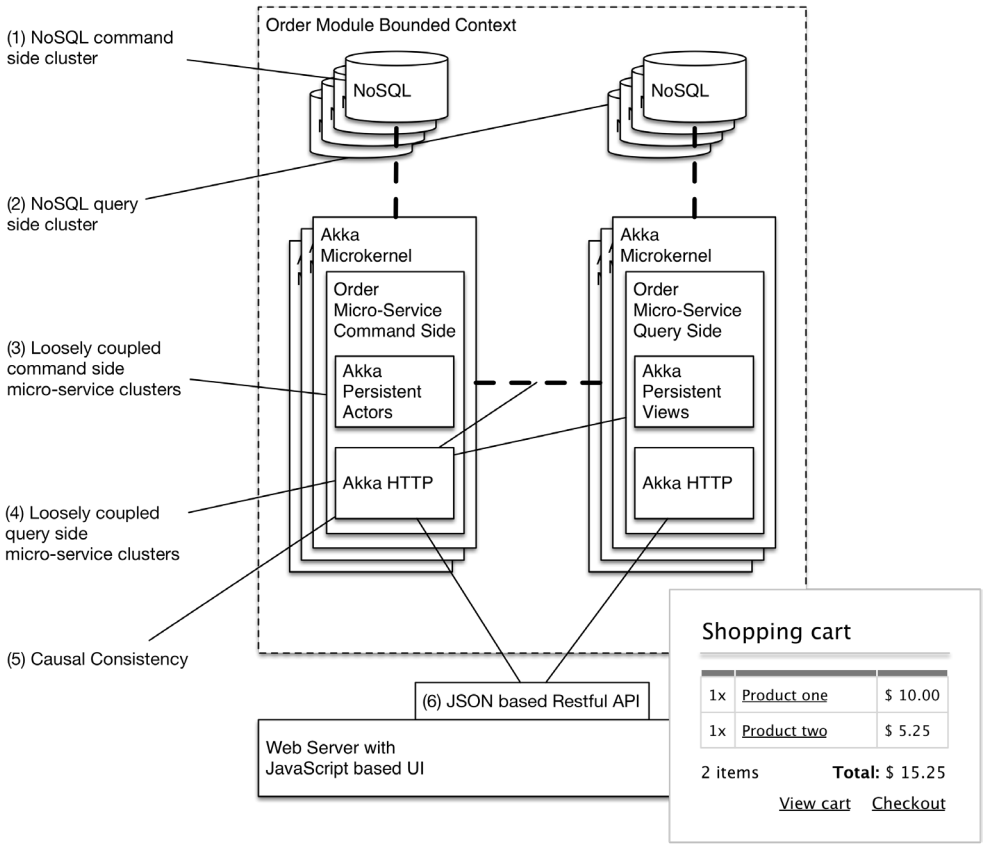

Figure 1.6 Shopping cart modelled in reactive architecture style

If this architecture seems more complicated than the monolithic one, that's because it is. Distributed applications are not easy to build, but with the advent of Akka, the task is a bit easier than it once was. For brevity sake, we only show the order micro-service; structurally the inventory, review and other services would be identical.

Looking at the figure from the top down:

- Order service

  The first thing you'll notice at the top of the figure is there is not a single centralized data store like there was in the monolithic example. The order service has been split into two sides: a command side and a query side, with each supported by a clustered NoSQL data store and sitting atop their own JVM. This pattern is commonly referred to as CQRS (Command Query Responsibility Segregation; we'll break this pattern into its key parts (C,Q,R,S), and deal with each in chapter 5, 6, and 7.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

Each side of the order service is micro in nature, and sits on top of a clustered Akka Microkernel, which we'll explain in chapter 2. The concept you should focus on for now is that you design your application as a suite of small services: each running its own process, loosely coupled and communicating with a lightweight, message-driven process, in our case Akka. These services are wrapped by the Akka Microkernel, which offers a bundling mechanism and is distributable as a single payload. There is no need for a Java Application Server or a startup script.

- Loosely coupled command side micro-service clusters

  The command side uses Akka Persistence as its storage mechanism, and Akka HTTP to process commands from the UI. Akka Persistence provides durability to your application by persisting the internal state of each actor. This allows for recovery when the actor is started, restarted after a JVM crash or by a supervisor, or migrated in a cluster and is the foundation to resilience in the Reactive Manifesto.

  Akka HTTP, provides an actor-based, asynchronous, lightweight and fast REST/HTTP layer for your application. The command construct will be explored in more detail in chapter 3.

- Loosely coupled query side micro-service clusters

  The query side uses Akka Persistent views to project data from the data store and Akka HTTP to deliver the projected data to the UI.

- Consistency models

  Finally, the two sides are synchronized using a technique known as a *consistency model*, a common technique used in distributed computing to keep isolated system synchronized, which we first discussed when looking at the CAP theorem in the previous section. As we discussed, consistency models can be either *eventual* (eventually all accesses to that item will return the last updated value) or *causal* consistency (ensures that the operations processes in the order expected).

We'll talk more about the importance of consistency models later in chapters 3 and 5, but for the time being you can think of consistency as the logical glue that holds the command and query sides together. It's essentially a contractual agreement that says whatever happens on the command side will make its way to the query side bound by some metrics like content and time. The query construct will be explored in more detail in chapter 5.

Next, let's look in greater depth at the principles of the Reactive Manifesto, which we introduced at the very beginning of the chapter, and see how the reactive architecture we just saw is based on these principles.

1.2.3 Understanding the Reactive Architecture

The foundation for reactive architectures is the Reactive Manifesto. Like many of us, Jonas Bonér, CTO of Typesafe, grew increasingly frustrated with the way modern applications were being architected, and felt there needed to be a clear, concise way to articulate the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

philosophies behind good distributed design. As a result, in September 23, 2013, the Reactive Manifesto (V1) was released and updated on September 16, 2014 (V2). The manifesto centers on four cornerstone attributes that lay the foundation for building reactive applications, shown in the figure below.

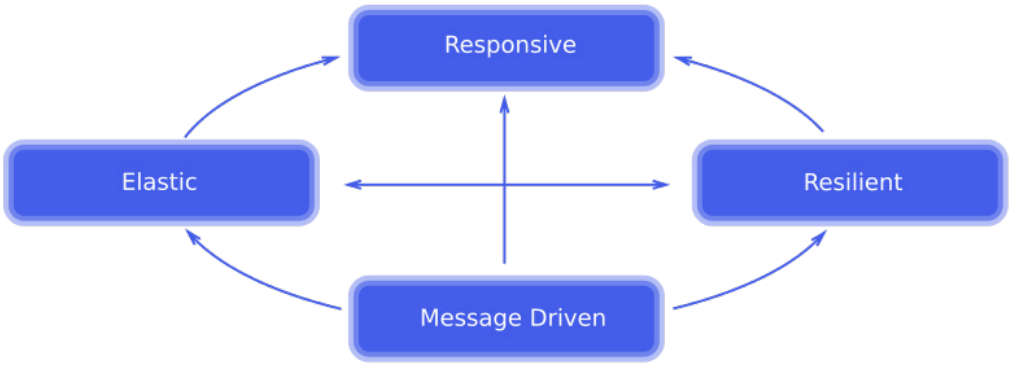

Figure 1.7 Traits of the Reactive Manifesto (http://www.reactivemanifesto.org/images/reactive-traits.svg)

You've already met the 4 traits shown in the figure earlier in the chapter, but now we'll explore their implications for reactive design.

MESSAGE DRIVEN

Message-driven architectures are loosely coupled, asynchronous, and non-blocking. Before we go any further, lets define what these terms mean, as they are paramount to being message driven.

- Loose coupling: A loosely coupled system, is a system where its components depend on each other in the least amount practical.
- Asynchronous: Able to execute a task without waiting (non-blocking) for it to complete.
- Non-blocking: Never waiting for a task to complete.

The result of this pattern is no concrete dependencies, and it allows the use of a distributed domain model which is crucial for scalability and leads to lower latency and higher throughput. As a result, reactive architectures are naturally scalable with the ability to elastically scale in and out. This type of architecture mitigates financial risk, allowing for the on-demand use of hardware and services made popular by Amazon. When load is low, you spin down services, and when it spikes, you spin them back up. Since you're only paying for what you use, you save money! Details of distributed domain modeling will be covered in chapter 4.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

ELASTIC

Elastic architectures are key for distributed computing. They are expected to expand and contract upon the changing load demands, achieved this by adding elasticity, the ability to add or remove nodes on the fly. This unique feature gives them the ability to scale both in and out and up and down, without redesigning or rewriting the application. Elasticity also mitigates risk in that hardware can now be utilized on demand, eliminating the need to keep a bank of un-utilized servers waiting for a spike in load. The technique that achieves this is location transparency. *Location transparency* uses logical names to find network resources, removing the need to know the physical location of both the users and the resource. We'll cover elasticity in more detail in chapter 2 and 3, when we explore the Akka Actor model.

RESILIENT

Reactive applications don't utilize traditional fault tolerance techniques. Instead, they embrace the notion of resilience. The Merriam-Webster definition of resilience is:

- the ability of a substance or object to spring back into shape
- the capacity to recover quickly from difficulties

Resilience is achieved by accepting failure and making it a first-class citizen in the programming model managed through isolation and recovery techniques like the bulkhead pattern which allow the application to self-heal. An example of this might be in the shopping cart example, we'll discuss again shortly. Imagine a scenario where the shipping module fails temporarily. In a reactive system, the user would still be able to interface with the shopping cart, add and delete items, while the shipping module (in the background) identifies failure and repairs itself. Resilience will be discussed in several chapters (3, 4, 5), when we deal is the distributed domain model and other error recovery concepts.

RESPONSIVE

Finally, reactive applications are responsive. Users are not interested in what your application does under the covers. They expect it to work the same in both high and low-load situations and failover and non-failover modes. Today's applications are expected to be real-time, engaging and collaborative: to respond to a user's actions without hesitation. For example, as mentioned above when the shipping module fails, the application will continue to respond. Reactive applications use stateful clients, streaming and observable models, among other things, to provide a rich, collaborative environment for the user. These are covered in great detail in chapters 7 and 8.

We've introduced a large number of concepts in this section and most importantly learned to use asynchronous message passing and share nothing designs0; don't worry, we'll cover all of these in detail throughout the book. For now its only matters that you understand the general concepts behind a reactive architecture. Now, let's dig a little deeper into the specifics of implementing these two architectures with our shopping cart example. We'll look at the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

details of placing an online order in both monolithic and reactive shopping carts, showing the distinct advantage of the reactive paradigm, the message-driven trait of reactive applications, and how that is distinctly different from the monolithic approach.

1.2.4 Monolithic shopping cart: creating an order

As noted earlier, the architectural problems around concurrency, scalability, and fault tolerance are significant in monolithic applications, but there are other challenges as well. One such challenge is the value of the data a monolithic application persists. Typically, monolithic applications store domain information in current state form, as opposed to behavioral form. As a result, much of the intent of the data stored is lost. To better explain this problem, lets look closer at our customer adding items to their shopping cart in both monolithic and reactive designs.

As you've seen, in a monolithic architecture, we would typically build our shopping cart application in a client-server fashion utilizing CRUD (create, read, update and delete) to manage the current state of our domain model. A customer browses available inventory, chooses four items, adds shipping information, and then checks out. Meanwhile in addition to the workflow just stated, many other things are happening in the background. Images are being fetched; reviews are being loaded, daily deals are being presented, etc., all of which we will need to deal with. To keep our example simple, we will focus the issues monolithic applications have around data persistence.

This order would most likely be wrapped in a single transaction using an ORM implementation that inserts into three tables: order, order_item, and shipping_information. The information stored represents the current state of the shopping cart order, as shown below.

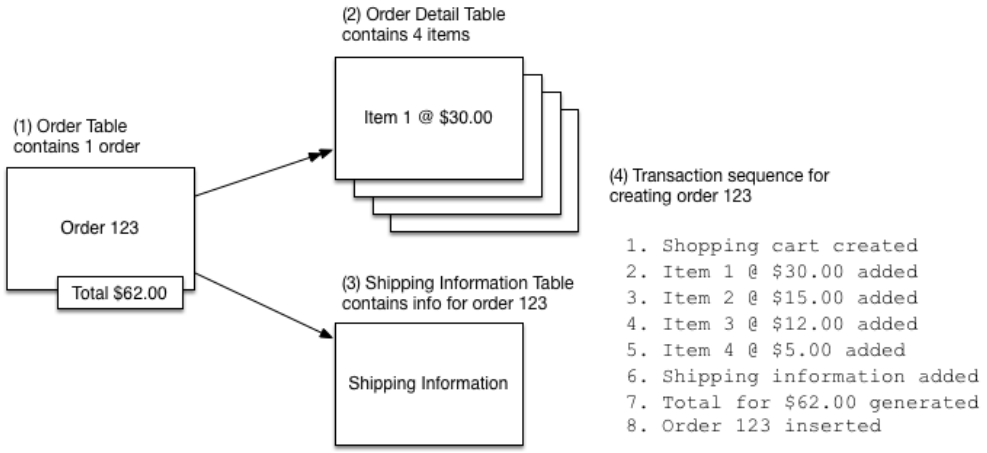

Figure 1.8 CRUD shopping cart current state after create

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

Now at some point in the future, prior to the order being shipped, the customer decides they no longer want one of the items they ordered. They log back into the shopping cart application, fetch their order, and delete the unwanted item. The order as it stands now (shown below in Figure 1.9), consists of three items totaling $47.00. The notion that item two has been deleted is lost, but more on that shortly.

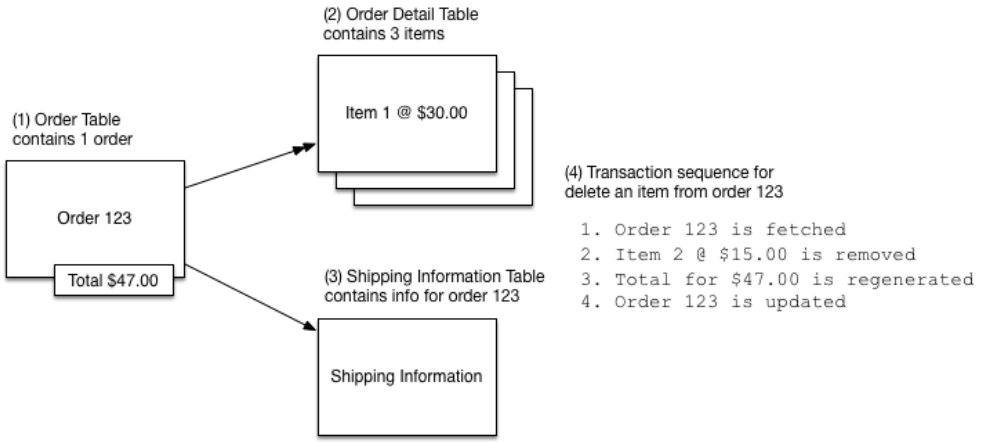

Figure 1.9 CRUD shopping cart current state after customer deletes an item

Concerned about the decrease in revenue from deleted items, the manager who oversees the shopping cart application asks the development team to generate a report for all items removed by the customers before orders ship. Therein lies the rub!

THE PROBLEM: USER INTENT NOT CAPTURED

Because the domain model by way of CRUD, only stores current state, the deleted data is lost. The development team will have to add this task to a future sprint and implement an audit log that tracks deleted items. Even worse, once implemented, they will only be able to track deletes from that point going forward, which has substantial implications about the value of the data!

What we should be looking to capture is the intent of our users because, from a business perspective, customer behavior is paramount. Rather than modeling our domain as a current state model, we should look at it in the form of user behavior as a sequence of recorded transactions or events. The CRUD model of persisting current state, which we are all so familiar with, does capture behavior, but the behavior it captures is system behavior, in the form of creating, reading, updating and deleting. As we've seen above, this doesn't tell us much about our users and compromises the value of our data. The reasons most systems

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

today rely on this model is primarily a result of the general acceptance of RDBMS as the center of web architecture. Fortunately, this is not the only way to view persistence.

Event sourcing, or persisting a sequence of events (behaviors), (not to be confused with message driven, which instead means reacting to an message), provides a means by which we can capture the real intent of our users. In an event sourcing system, all data operations are viewed as a sequence of events that are recorded to an append-only store. We'll talk more about event sourcing in chapter 2, but for now we'll look at two examples that best showcase its capabilities: First, the canonical example, a bank account register, and then we'll redo the CRUD shopping cart as a reactive shopping cart.

What's the difference between Message, Command, and Event?

The distinction between Messages, Commands, and Events is important, and one we need to make before going too far into our architecture discussion. Messages can come in two flavors: abstract or concrete. We'll look at two simple examples below to explain.

ABSTRACT MESSAGE

You can think of an *abstract message* as a blank sheet of paper, a structure to capture a conversation between two parties. The paper in and of itself is not a conversation until I write something on it. To start off our discussion, I write down on the paper a request to borrow a book from you. The Abstract Message has now become (been implemented as) a Command, a request "to do" something. In response, you write back to me that you have mailed the book. At this point, the Abstract Message has become (been implemented as) an Event, the notification that something "has occurred". In this example, both Command and Event are forms of Message, in computing lingo they implement the Message Interface.

CONCRETE MESSAGE

A *concrete message* is like an envelope. It is a container that has a payload. That payload can be anything. In the example above, the payload would be either a command or an event. The distinction here is message is concrete, just like command and event.

1.2.5 Event Sourcing: A banking example

In a mature business model the notion of tracking behavior is quite common. Consider for example a bank accounting system. A customer can make deposits, write checks, make ATM withdrawals, transfer monies to another account, etc.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

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

Figure 1.10 Bank account register transaction log with five transactions

Here we see a typical bank account register.

- The account holder starts out by depositing $10,000.00 into the account.
- Next, they write a check for $4,000.00;
- make an ATM withdrawal;
- write another check, and
- finally make a deposit.

We store a record of each transaction as an independent event. To calculate the balance, the delta, or change caused by the current transaction, is applied to the last known value (the sum of all previous transactions). As a result, we have a verifiable audit log that can be reconciled to ensure validity. The present balance at any point can be derived by replaying all the transactions up to that point. Additionally, we have captured the real intent of how the account holder manages their finances.

Imagine a scenario where the bank only persisted current state for the account. When the account holder tries to reconcile their account, they notice a discrepancy. They double check their reconciliation and conclude the bank must have made a mistake. They quickly call the bank, and state their case, upon which the bank promptly replies, "I'm sorry; we have no record of that transaction. We only store the last update to your balance." That would be ludicrous; while this is an extreme example of loosing the users intent, changes to the balance, unfortunately, this is what happens in a CRUD based monolithic application.

1.2.6 Reactive shopping cart: creating an order with event sourcing

Another way we can look at events, or what some might think of as transactions, is that they are a notification that something has happened; they are indicative or evidential in mood, as they state a recorded fact. We'll discuss details such as this for event sourcing more in-depth especially how it relates to CQRS and commands in general later in chapter 5 and 6.

For now, let's dig back into our shopping cart example and model it in an event sourcing fashion, as shown in Figure 1.11.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

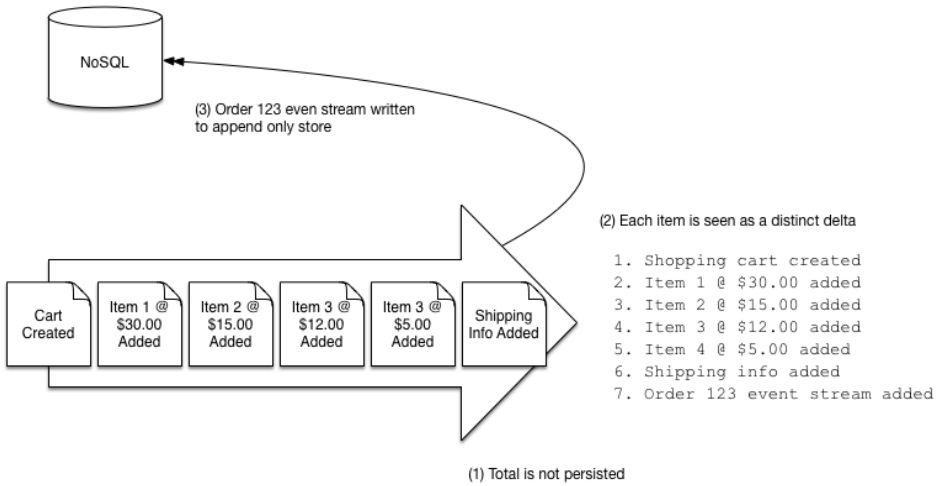

Figure 1.11 Reactive shopping cart stores events

As you can see, the workflow addresses the same concerns the earlier CRUD example, with crucial differences.

- There is no total generated.
- Each item is a distinct delta that persists in sequential order.
- The entire construct as a stream of deltas are written to an append-only store.

As the figure above shows, the reactive shopping cart has no current state of the order or line items that persist. Rather, a sequence of deltas that capture user behavior are stored in order. Additionally, note the indicative tense of an event: item added, shipping information added. Events are something that has happened, which is an important distinction when compared to commands. You can reject a command as its a request to do something, whereas an event, you cannot reject because it represent something that has already occurred. We'll talk about this later in the book where we compare and contrast events and commands with code to show you how this is done.

As a wrap up of what we've covered on these differences between monolithic and reactive applications, the next figure compares creating an order in shopping carts for both approaches.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

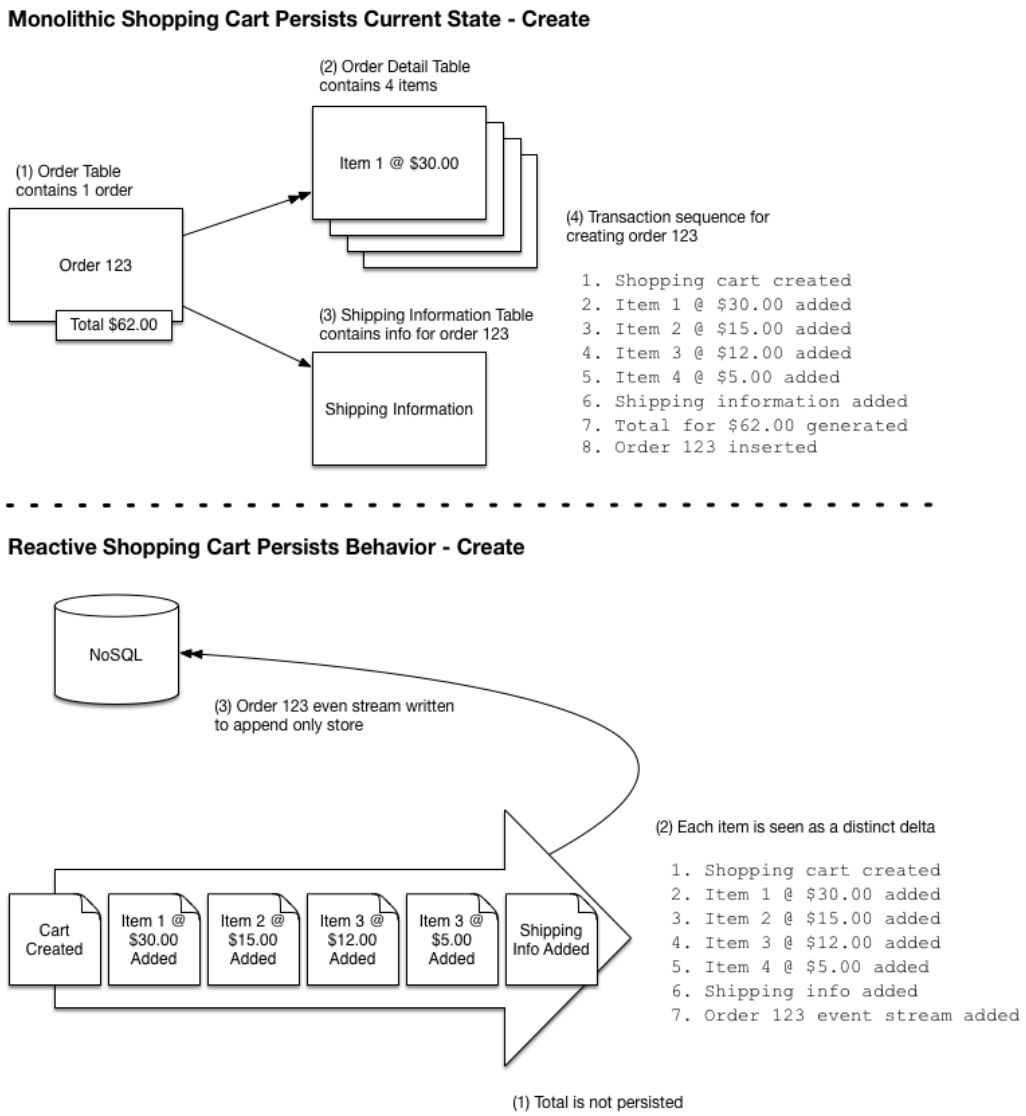

Figure 1.12 Crud shopping cart creates current state vs reactive, which persists behavior

Next, like we did with the monolithic shopping cart earlier, let's look at what happens when at some point in the future, prior to the order being shipped, the customer decides they no longer want one of the items they ordered.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

REACTIVE HANDLING OF A DELETED ITEM

The customer logs back into the shopping cart application, fetches their order and deletes the unwanted item, as shown in this next figure.

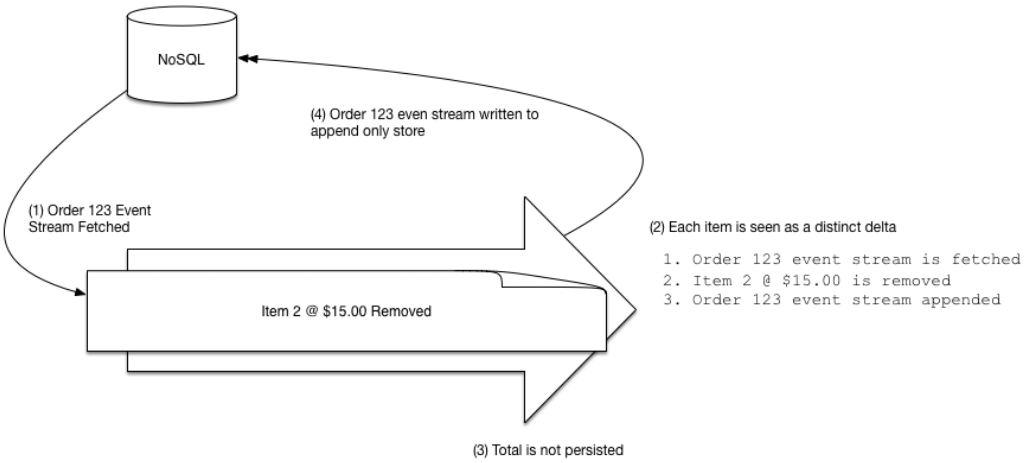

Figure 1.13 Reactive shopping cart appends delete event

Again, the workflow is similar to the CRUD example, with subtle but crucial differences:

- No total is generated.
- The delete is seen as a distinct delta that will persist at the end of the event stream.

Now, as with the CRUD shopping cart, the manager who oversees the shopping cart application asks the development team to generate a report for all items removed by the customers before orders ship. From a data perspective here is where a reactive application shines.

- We have everything we need to craft the report, because we capture the intent of the user in the form of events, rather than the current state of the model.
- Deletes are not updates to our current state as with a CRUD solution; they are simply an event captured in a user's behavior workflow.
- In a reactive system, deletes are explicit and verifiable, whereas in a CRUD solution they are implicit and require tracking.

The next figure compares both monolithic and reactive carts dealing with a deleted item.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

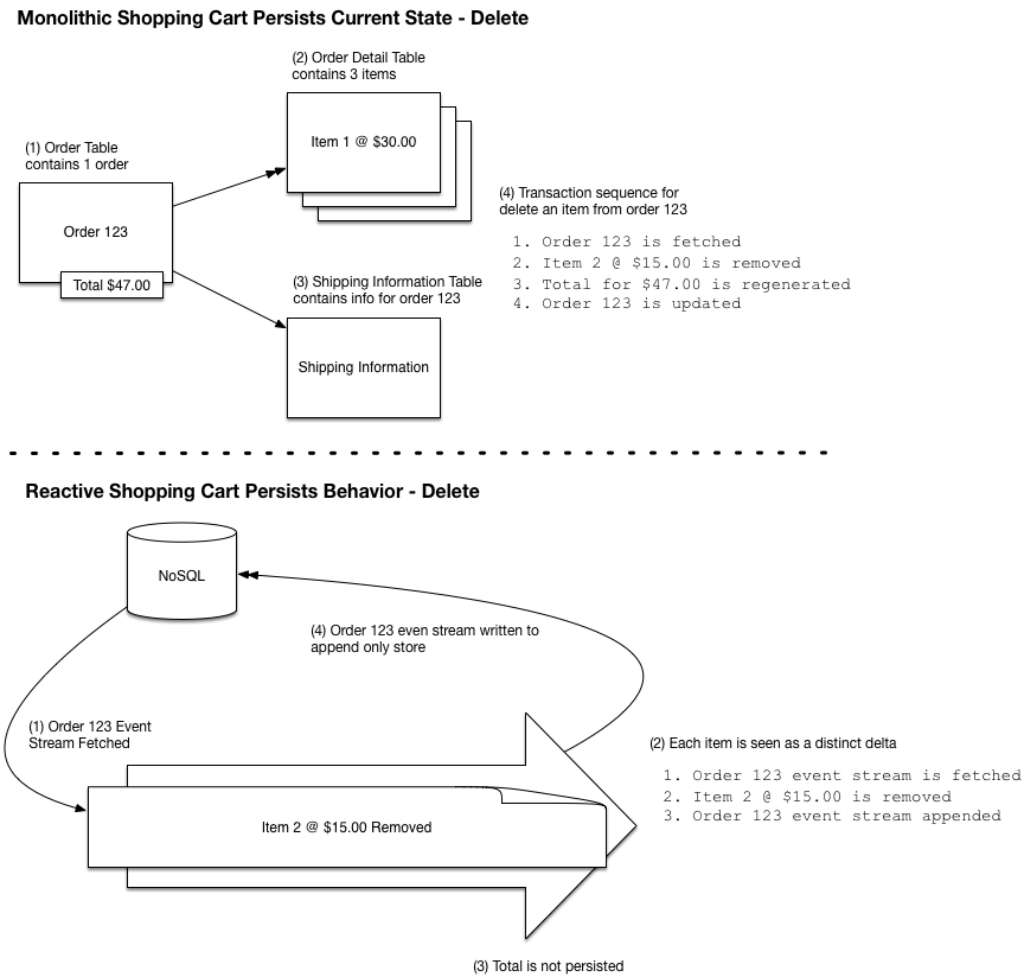

Figure 1.14 Crud shopping cart delete overwrites state vs reactive appends delete event

So far, we've shown you how a reactive architecture solves several issues with reacting to events and transactions: the first characteristic of reactive applications from the Reactive Manifesto. But what else can reactive applications react to?

1.3 What are reactive applications reacting to?

There is a coming tidal wave that will have a significant impact on how we reason about application design. This tidal wave has a name: The Internet of Things(IoT). By 2020 it is

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

theorized that the Internet will be comprised of some 30 billion interconnected devices, a web of connections shown in the next figure.[4]

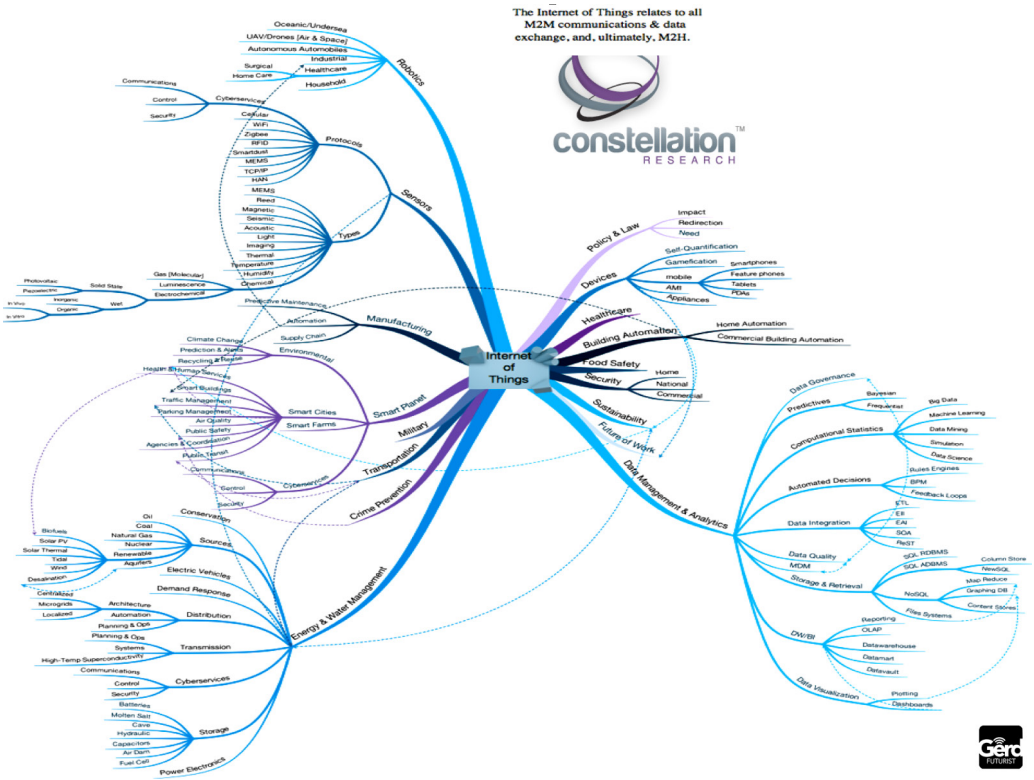

Figure 1.15 The Internet of Things (IoT) logical view
(http://cfile29.uf.tistory.com/image/2119F45051F5B41D1AE4CA)

Consider for a moment the impact of 30 billion devices. Currently, there are approximately 12 billion devices interconnected, and we can already see the effect it's having: slow websites, downtime longer than expected, even Gmail has interruptions every couple of months. Imagine tripling that number in just seven years; the impact will be staggering. To make

[4] ABI Research: More Than 30 Billion Devices...

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

matters worse, it's estimated that 99% of the physical objects (homes, cars, buildings, wearables, etc.) that may someday join the Internet are still unconnected[5].

REACTING TO LOAD, FAILURE AND USERS

IoT will have a drastic impact on application failure rate as a result of load spikes, and will compromise the application's ability to respond to its user base. Monolithic applications will crumble under these conditions, which in turn will affect every company's bottom line.

If modern programmers are going to be successful and not succumb to the fate of the light-colored peppered moth during the Industrial Revolution, they must learn new tools and techniques: tools and techniques designed for this new environment, that embrace Distributed Systems and Cloud Computing. This is exactly what Reactive Applications are all about.

1.4 What You Will Learn

We believe that the reactive paradigm is here to stay and will influence and shape the world of computing for many years to come. To facilitate that process, we will present you with a variety of philosophies, patterns and technologies that you may not be familiar with. Don't let that give you pause. The purpose of this book is to walk you step-by-step through that process, and at the end you will be equipped to meet that goal.

PHILOSOPHIES AND PATTERNS

Following is a list of philosophies, patterns and technologies, broken down by the traits of the Reactive Manifesto, to give you a better sense of where we are heading. We list a chapter number the first time a topic is mentioned, so you get a sense of where we'll cover that concept. Each of the patterns and technologies below will be discussed to the level of understanding required to build a distributed application. We'll walk you step-by-step through each of these, so don't get too worried. Strap in, you're in for a great ride.

Asynchronous communication with loosely coupled design.

- Akka - a toolkit and runtime for building highly concurrent, distributed, and resilient message driven applications on the JVM.
- Akka Actors - lightweight concurrent entities that process messages asynchronously using a message-driven mailbox pattern as we'll see in chapter 2 and 3.
- Akka HTTP - embeddable HTTP stack entirely built on Akka actors.
- CQRS-ES - A set of patterns communicating via event messages, covered in chapters 5, 6, and 7.

[5] Forbes Magazine: How many things...

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

ELASTIC

Capable of expansion and upgrade on demand.

- Akka Clustering - fault-tolerant decentralized peer-to-peer based cluster membership service with no single point of failure or single point of bottleneck.
- Akka Sharding - actors with an identifier, can be automatically distributed across multiple nodes in the cluster.
- Akka Streams - streaming model that protects each consumer of data from being overwhelmed by its producer by propagating back pressure.
- Command Query Responsibility Segregation (CQRS) - models used for commands (write) are different from the models used to query (read).
- Distributed Domain Driven Design (DDDD) - a distributed approach to software development for complex needs by connecting the implementation to an evolving model; covered in chapter 4.
- Elasticity - the system expands and contracts according to load.
- Event Sourcing - persisting a sequence of behaviors.

RESILIENT

More than fault tolerance, the ability to self heal.

- Akka Clustering - fault-tolerant decentralized peer-to-peer based cluster membership service with no single point of failure or single point of bottleneck.
- Akka Persistence - enables stateful actors to persist their internal state so that it can be recovered when an actor is started, restarted after a JVM crash or by a supervisor, or migrated in a cluster.
- Akka Sharding - actors with an identifier, can be automatically distributed across multiple nodes in the cluster.
- Akka Streams - streaming model that protects each consumer of data from being overwhelmed by its producer by propagating back pressure.
- Failure Detection - responsibility for detection of node failures or crashes in a distributed system.
- Modular / Micro-Service Architecture - way of designing software applications as suites of independently deployable services; we'll cover this in chapter 8.

RESPONSIVE

The ability to respond regardless of circumstances.

- Command Query Responsibility Segregation (CQRS) - models used for commands (write) are different from the models used to query (read).
- Futures - a data structure used to retrieve the result of some concurrent operation.
- Akka HTTP - embeddable HTTP stack entirely built on Akka actors.
- Akka Streams - streaming model that protects each consumer of data from being

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

overwhelmed by its producer by propagating back pressure.

TESTING

- Test Driven Development (TDD) - a software development process that relies on the repetition of a very short development cycle: first the developer writes an (initially failing) automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test, and finally refactors the new code to acceptable standards; covered in chapter 12.
- Behavioral Driven Development (BDD) - combines the general techniques and principles of TDD with ideas from domain-driven design and object-oriented analysis and design to provide software development and management teams with shared tools and a shared process to collaborate on software development.
- Testkit - The Akka testkit provides all the means necessary to test your actors asynchronously, which mimics how they behave in real world at runtime.
- MultiJVM Testing - supports running applications (objects with main methods) and ScalaTest tests in multiple JVMs at the same time. Useful for integration testing where multiple systems communicate with each other.

1.5 Summary

- The Reactive Manifesto:
  - is Message driven
  - is Elastic, expanding and contracting with load
  - is Resilient in the face of failure
  - is Responsive to users
- You have learned the limitations and pitfalls of traditional, monolithic architectures.
- You have learned about distributed computing and how blocking limits concurrency and therefore distribution and scale.
- We have discussed consistency models and the CAP theorem.
- We have discussed the reasons why Reactive design solves the distributed programming problems existing today.

We'll now apply these theories in chapter 2, where we'll show you how apply them with the Akka toolkit.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

2

Getting Started with Akka

This chapter covers

- Building an actor system
- Distributing and scaling horizontally
- Applying reactive principles

You understand from the first chapter the tenets of Reactive design, but haven't seen them in practice. This chapter will change that. In this chapter, you will build a simple reactive system using the *Actor* model that was introduced in chapter 1. The actor model is one of the most common reactive patterns. Actors can send and receive messages, make local decisions, create new actors, and do all of that asynchronously and without locks. The example will be built using the Akka toolkit, which you also saw previously. Akka is a powerful system for creating and running actors. It is written in the Scala language, and the examples in this chapter are also written in Scala. The next two chapters explain Akka in more depth.

The system you will build consists of just two actors passing messages to each other, but the same skills may be used to create much larger applications. You next will learn how to scale the system horizontally by adding more copies of one of the actors. Finally, you will see how the approach produces a system that is both message-driven and elastic, so has two of the four reactive properties from the Reactive Manifesto.

2.1 Understanding Messages and Actors

Reactive systems are message-driven, so it comes as no surprise that messages play a key role. Actors and messages are the building blocks to an actor system. An actor receives a message and does something in response to it. That something might include performing a computation, updating internal state, sending more messages, or perhaps initiating some I/O.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

Of course, much the same could be said of an ordinary function call. To understand what an actor is, it is useful first to consider some of the problems that can arise from an ordinary function call. A function accepts some input parameters, performs some processing, and returns a value. The processing might be quick, or could take a long time. However long it takes, the caller is blocked waiting for the return value.

2.1.1 Moving from Functions to Actors

If a function includes an I/O operation, then control of the processor core most likely will be handed off to another thread while the caller is waiting for a response. The caller will not be able to continue processing until both the I/O operation is complete and the scheduler hands control back to the original processing thread, as shown in figure 2.1. This maintains the illusion for the caller that it made a simple synchronous call. What actually happened was that any number of other threads may have been running in the background, potentially even changing data structures referenced by the original input parameters.

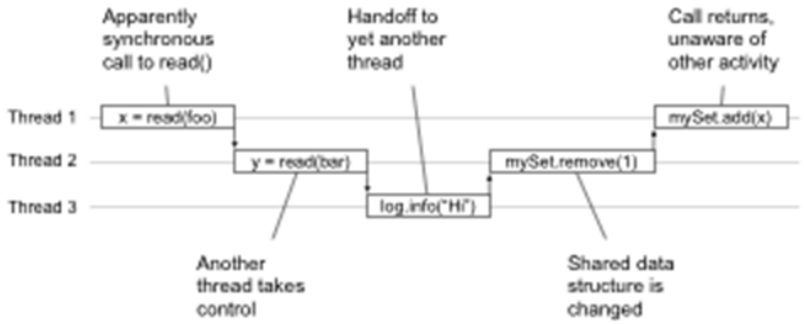

Figure 2.1 The illusion of a synchronous call can be the source of unexpected behavior.

The developer might know that the function is liable to take a long time and design the system to accommodate thread safety and timing, but this is not always the case. Sometimes the amount of time cannot be predicted. For example, if the function has a cache of recently used data in memory but must go to a database if the data is not in the cache, then the amount of time the function requires may vary by many orders of magnitude from one call to the next. Ensuring that the caller and callee have the correct synchronization and thread safety without deadlocks can be extremely difficult. If the API is properly encapsulated, the entire implementation might be replaced with one that has completely different characteristics. An excellent design around the original characteristics could become an inappropriate design with respect to the replacement.

The result is often complex code littered with exception handlers, callbacks, synchronized blocks, thread pools, timeouts, mysterious tuning parameters, and bugs that developers never seem to be able to replicate in the test environment. What all of these things have in common

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

is that they have nothing to do with the business domain. Rather, they are aspects of the computing domain imposing themselves on the application.

The actor model pushes these concerns out of the business domain and into the actor system.

ACTORS ARE ASYNCHRONOUS MESSAGE HANDLERS

A simplified view of an actor is a receive function that accepts a message and produces no return value. It just processes each message as it is received from the actor system. The actor system manages a mailbox of messages addressed to the actor, ensuring that the actor only has to process one message at a time. An important consequence of this design is the sender never calls the actor directly. Instead, the sender addresses a message to the actor and hands it to the actor system for delivery.

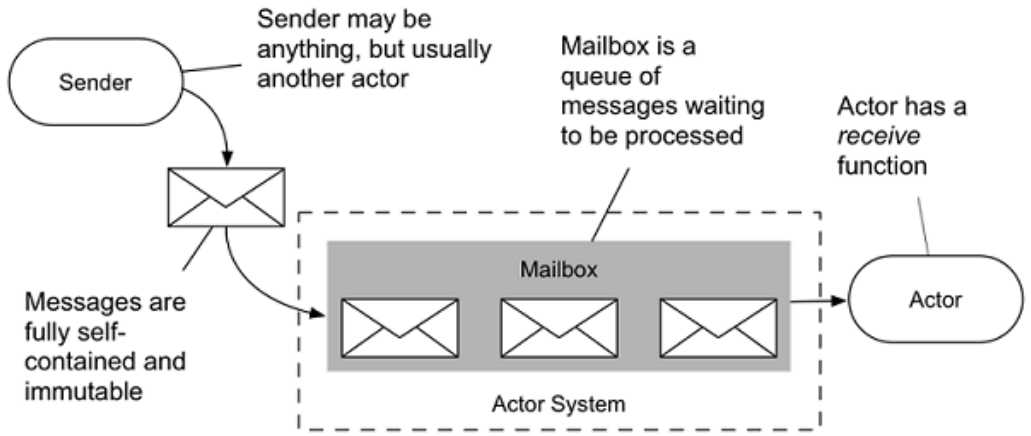

Figure 2.2 The sender obtains a *reference* to address messages to the actor through the actor system

Actors remove many of the problems of function calls by abandoning the illusion that everything is synchronous. Instead, actors take the approach that everything is one-way and asynchronous. The underlying system takes responsibility for delivering the message to the receiving actor. It may be delivered immediately, or some time later. An actor receives a new message only when it is ready to process a message, one at a time. Until then, the actor system holds on to the message. That means the sender may proceed immediately to other tasks rather than waiting for a response that may come some time later, or perhaps not at all. If the receiving actor has a response for the sender, it is handled with another asynchronous message.

> **TIP** **Senders never call actors directly. All interaction between sender and actor is mediated by the actor system.**

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

MESSAGES ARE SELF-CONTAINED AND IMMUTABLE

As messages are passed between actors, they might move to an actor system on an entirely different server. Messages must be designed so that they can be copied from one system to another. That means all of the information has to be contained within the message itself. Messages cannot include references to data that is outside the message. Sometimes a message doesn't make it to the destination, and must be sent again. Later, you will learn that the same message might be broadcast to more than one actor.

For this to work, the most important aspect of a message is that it must be *immutable*. Once it is sent, it is read-only and cannot be allowed to change. If it were to change after being sent, there would be no way to know whether the change happened before or after it was received, or perhaps while it was being processed by another actor. If the message happens to have been sent to an actor on another server, the change might have been made before or after it was transmitted, and there is no way to know. Worse, if the message has to be sent more than once, it is possible that some copies of the message include the change and some do not. Immutable messages make all those worries go away.

2.1.2 Modeling the Domain with Actors and Messages

Actors should correspond to real things in the domain model. The example in this chapter consists of a tourist, who has an inquiry about a country, and a guidebook, which provides guidance to the tourist.

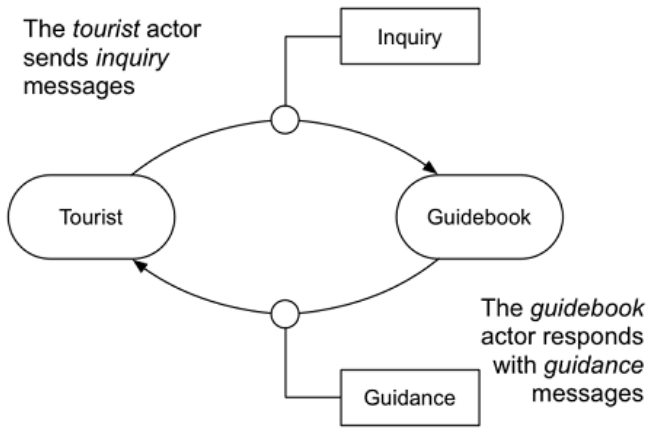

Figure 2.3: The tourist actor sends inquiry messages to the guidebook. The guidebook actor responds with guidance messages returned to the tourist actor.

The example actor system is shown in figure 2.3. It contains two actors, a *tourist* and a *guidebook*. The tourist sends an *inquiry* message to the guidebook, and the guidebook sends *guidance* messages back to the tourist. Messages are one-way affairs, so the inquiry and the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

guidance are defined as separate messages. Just like real life, the tourist must be prepared to receive no guidance, a single guidance message, or even multiple guidance messages in response to a single inquiry. (The tourist in the example will be able to receive multiple guidance messages, but deciding what to believe would be the subject of an entirely different book.)

2.1.3 Defining the Messages

You know already that messages are self-contained and immutable, and now have identified some messages that will be needed for the example. In Scala, case classes provide an easy way to implement messages. The example messages, shown in listing 2.1, define a case class for each message. It follows the convention that messages are defined in the companion object to the actor that receives the message. The `Guidebook` actor will receive `Inquiry` messages that include a `String` for the country code that is being inquired about. The `Tourist` actor will receive `Guidance` messages, which contain the original country code and a description of the country. The original country code is included in the guidance so that the information in the message is fully self-contained. Otherwise, there would be no way to correlate which guidance goes with which inquiry. Finally, the `Start` message will be used later to tell the `Tourist` actor to start sending inquiries.

Scala Case Classes

For readers not familiar with Scala, a case class is defined by a class name and some parameters. By default, instances of a case class are immutable. Each parameter corresponds to a read-only value that is passed to the constructor. The compiler takes care of generating the rest of the boilerplate for you. The concise Scala definition

```
case class Inquiry(code: String)
```

produces a class equivalent in Java to

```
public class Inquiry {
   private final String code;

   public Inquiry(String code) {
      this.code = code;
   }

   public String getCode() {
      return this.code;
   }

   // …more methods are generated automatically
}
```

Case classes generate more than just the getters. They automatically include correct equality, hash codes, a human-friendly `toString`, a `copy` method, a companion object, support for pattern matching, and additional methods that are useful for more advanced functional programming techniques.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

Listing 2.1 Message Definitions

```
object Guidebook {
  case class Inquiry(code: String)                          (1)
}
object Tourist {
  case class Guidance(code: String, description: String)    (1)
  case class Start(codes: Seq[String])                      (2)
}
```

(1) The Inquiry and Guidance messages are simple case classes
(2) The Start message will be used to start the tourist sending inquiries

Now that the messages are defined, it's time to move on to the actors.

2.1.4 Defining the Actors

The example requires a `Tourist` actor and a `Guidebook` actor. Most of the behavior of an actor is provided by extending the `akka.actor. Actor` trait. One thing that cannot be built in to the actor trait is what to do when a message is received, because that is specific to the application. You provide that by implementing the abstract `receive` method.

THE TOURIST ACTOR

As shown in listing 2.2, the `receive` method on the `Tourist` defines cases to handle the two types of message expected by the tourist. In response to a `Start` message, it extracts codes and sends an `Inquiry` message to the `guidebook` actor for each one it finds. It also receives `Guidance` messages, which it handles by printing the code and description to the console.

The tourist needs to address messages to the guidebook, but actors never keep direct references to other actors. Notice that the `guidebook` is passed to the constructor as an `ActorRef`, not an `Actor`. An `ActorRef` is a *reference* to an actor. Since an actor may be on a completely different server, having a direct reference is not always possible. In addition, actor instances may come and go over the lifetime of the actor system. The reference provides a level of isolation that allows the actor system to manage those events. It also prevents actors directly changing the state of other actors. All communication between actors must be through messages.

Listing 2.2 Tourist Actor

```
import akka.actor.{Actor, ActorRef}

import Guidebook.Inquiry
import Tourist.{Guidance, Start}

class Tourist(guidebook: ActorRef) extends Actor {

  override def receive = {
    case Start(codes) =>                          (1)
      codes.foreach(guidebook ! Inquiry(_))       (2)
    case Guidance(code, description) =>
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

```
      println(s"$code: $description")          ❸
  }
}
```

❶ Extract the codes from the message
❷ For each code, send an inquiry message to the guidebook using the "!"operator
❸ Print the guidance to the console

The "!" Operator

The use of the "`!`" operator to send messages from one actor to another may be confusing the first few times you encounter it. It is a method defined by the `ActorRef` trait. Writing

```
ref ! Message(x)
```

is equivalent to writing

```
ref.!(Message(x))(self)
```

Both methods use the `self` value, which is an `ActorRef` provided by the `Actor` trait as a reference to itself. The "`!`" operator takes advantage of Scala infix notation and the fact that `self` is declared as an implicit value.

THE GUIDEBOOK ACTOR

The `Guidebook` in listing 2.3 is similar to the `Tourist` from listing 2.2. It processes just one message, an `Inquiry`. When it receives an inquiry, it uses a few classes built in to the `java.util` package to generate a rudimentary description suitable for the example. It then produces a `Guidance` message to send back the tourist.

The guidebook needs to address messages back to the tourist that sent the inquiry. An important difference between the `Guidebook` and the `Tourist` is how each acquires a reference to the other. In the `Tourist`, a fixed reference to the guidebook is provided as a parameter to the constructor. Because there could be many tourists all consulting the same guidebook, that approach does not work here. It would not make sense to tell a guidebook in advance about every tourist who might use it. Instead, the guidebook sends the guidance message back to the same actor that sent the inquiry. The `sender` inherited from the `Actor` trait provides a reference back to the actor that sent the message. This reference can be used for simple request-reply messaging.

> **NOTE** Knowing that Akka is used for concurrent applications, you might expect that the `sender` reference would have to be synchronized to avoid the receive processing for one message inadvertently responding to the sender of another. As you will learn in the next two chapters, the Akka design prevents this ever happening. For now, just rest assured that you don't need to worry about it.

Listing 2.3 Guidebook Actor

```
import akka.actor.Actor

import Guidebook.Inquiry
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

```
import Tourist.Guidance

import java.util.{Currency, Locale}

class Guidebook extends Actor {
  def describe(locale: Locale) =                    (1)
    s"""In ${locale.getDisplayCountry},
[CA]|${locale.getDisplayLanguage} is spoken and the currency
[CA]|is the ${Currency.getInstance(locale).getDisplayName}"""

  override def receive = {
    case Inquiry(code) =>
println(s"Actor ${self.path.name}                   (2)
[CA]responding to inquiry about $code")
      Locale.getAvailableLocales.
filter(_.getCountry == code).                       (3)
        foreach { locale =>
          sender ! Guidance(code, describe(locale)) (4)
        }
  }
}
```

(1) Use Java built-in packages to produce a rather basic description
(2) Print a log message to the console
(3) Find every locale with a matching country code. This implementation is rather inefficient
(4) Send the guidance back to the sender

Now that you have two complete actors and some messages to pass between them, you will want to try them yourself. First, you need to set up your development environment to build and run an actor system.

2.2 Setting up the Example Project

The examples in this book are built using *sbt*, which is a build tool commonly used for Scala projects. The home page for the tool is www.scala-sbt.org, and there you can find instructions to install the tool for your operating system. The example code is available online. You can retrieve a copy of the complete example using the command

```
git clone https://github.com/bhanafee/HelloRemoting.git
```

The layout of the project is shown in figure 2.4, and is similar to that used by other build tools such as Maven and Gradle. The project includes source code and the following files:

- *build.sbt*—contains the build instructions
- *build.properties*—tells sbt which version of sbt to use

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

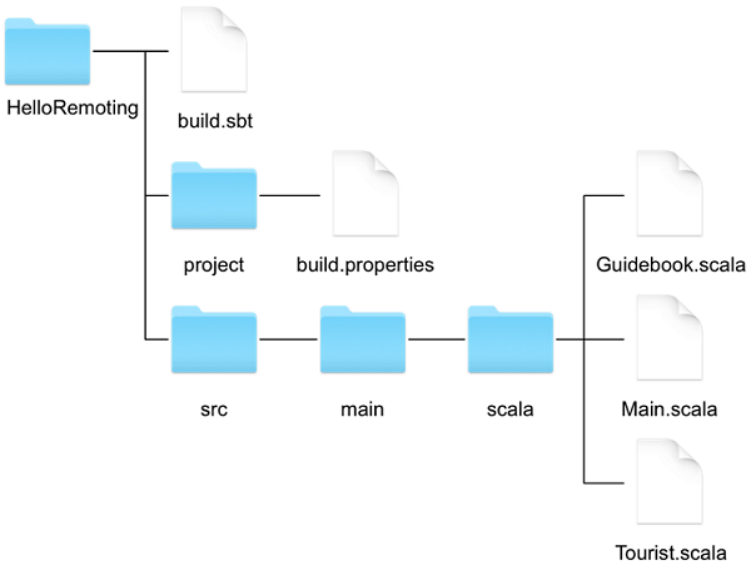

Figure 2.4 The layout of the sbt project follows the same pattern used by other build systems such as maven and gradle.

As with other modern tools, sbt prefers convention over configuration. The `build.sbt` file shown in the following listing contains just a project name and version, Scala version, repository URL, and a dependency on `akka-actor`.

Listing 2.4 build.sbt

```
name := "Guidebook"

version := "1.0"

scalaVersion := "2.11.8"                                    (1)

resolvers += "Lightbend Repository" at
[CA]http://repo.typesafe.com/typesafe/releases/             (2)

libraryDependencies ++= Seq(
  "com.typesafe.akka" %% "akka-actor" % "2.4.8"             (3)
)
```

(1) The example was tested with Scala 2.11.8
(2) Typesafe is now known as Lightbend, but the repository at typesafe.com is still supported
(3) The example was tested with Akka 2.4.8. The %% tells SBT to use a version of the library that was compiled for the version of Scala defined above

The `build.properties` file allows sbt to use a different version of itself for each project. The default version available by typing

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

```
sbt about
```

at the console. The complete one-line file is shown here.

Listing 2.5 build.properties

```
sbt.version=0.13.12          (1)
```

(1) The example was tested with sbt 0.13.12

You have already seen the source code for the messages and the two actors. Those will remain the same throughout the rest of this chapter. Whether the example actors are in one actor system or spread across multiple actor systems across many servers is determined entirely by configuration and the `Main` programs that drive the system. In the next section, you will run both actors in one actor system. After that, you will learn how to scale the system using multiple actor systems.

2.3 Starting the Actor System

Akka does not require much to get started. You just create the actor system and add some actors. The Akka library does the rest. Sometimes, as shown in figure 2.5, it is useful to get things moving by sending a first message into the system. In the next two chapters, you will learn a bit more about what Akka is doing behind the scenes. To learn even more about the internals, see Raymond Roestenburg, Rob Bakker, and Rob Williams's *Akka in Action* (Manning, 2016).

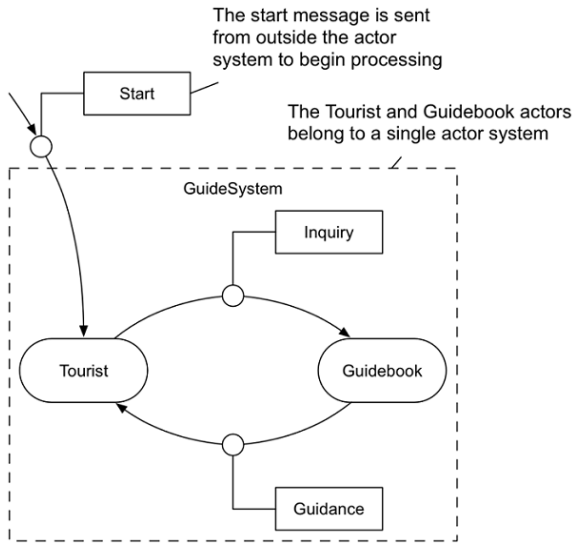

Figure 2.5 Both the tourist and the guidebook actors will be deployed into the same actor system. The start message is sent from outside the actor system.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

2.3.1 Creating the Driver

The driver program shown in listing 2.6 does as expected. It creates an actor system, defines the actors, and sends the Start message. The definition of the actors is interesting. Actor instances may come and go over the life of the actor system. The actor system takes responsibility for creating new instances, so needs enough information to construct a new instance. That information is passed via `Props`. The steps are:

1. Create a `Props` object that contains the class of the actor and the constructor parameters, if any.
2. Pass the `Props` to the `actorOf` method to create a new actor and assign it a name. This method is defined by the `ActorRefFactory` trait. That trait is extended by several classes including `ActorSystem`, which is used in the example.
3. Record the `ActorRef` returned by `actorOf`. Callers do not receive a direct reference to the newly created actor.

Listing 2.6 The Main driver application

```
import java.util.Locale

import akka.actor.{ActorRef, ActorSystem, Props}        (1)

import Tourist.Start                                    (2)

object Main extends App {
  val system: ActorSystem = ActorSystem("GuideSystem") (3)

  val guideProps: Props = Props[Guidebook]              (4)

val guidebook: ActorRef =
[CA]system.actorOf(guideProps, "guidebook")             (5)

  val tourProps: Props =
[CA]Props(classOf[TouristActor], guidebook)             (6)

  val tourist: ActorRef = system.actorOf(tourProps)     (5)

  tourist ! messages.Start(Locale.getISOCountries)      (7)
}
```

(1) Akka library
(2) Start message shown previously
(3) Create the actor system that will contain the actors
(4) Props define a "recipe" for creating instances of the Guidebook actor
(5) Create an actor based on the Props, returning a *reference* to the actor
(6) The Props for a Tourist include a reference to the Guidebook actor
(7) Send an initial message to the tourist actor

There is nothing special about the driver. Because it extends the `App` trait, it automatically has a main function just like any other Scala or Java application.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

2.3.2 Running the driver

The output of the build is a JAR file. You could use sbt to generate the build, then use the java command to launch it, but during development it is easier to let sbt take care of that too. Use

```
sbt run
```

to build and launch the application. As with nearly any framework, the first time you build the application may take some time because the dependencies need to be downloaded. Sbt uses Apache Ivy (http://ant.apache.org/ivy) for dependency management, and Ivy will cache the dependencies locally.

Now for the part you have been waiting for. If everything has built successfully, you should see the guidebook printing a message for every inquiry it receives, and the tourist printing concise travel guidance for every country. Congratulations! You have just started your first actor system. A more sophisticated application would send another message to tell the actors to shut themselves down gracefully. For now, use control-C to stop the actor system.

2.4 Distributing the Actors over Multiple Systems

Actors are lightweight objects. At about 300 bytes, the memory overhead required per actor is a small fraction of the stack space consumed by a single thread. It is possible to hold a lot of actors in a single JVM. At some point, that still is not enough. The actors have to scale across multiple machines.

You have already put in practice the most important concepts that make it possible. Actors refer to each other only via actor references. The tourist actor refers to the guidebook using an `ActorRef` supplied to the constructor, and the guidebook actor refers to the tourist only through the sender `ActorRef`. An `ActorRef` may refer to a local actor or to a remote actor, so both of those actors are already capable of working with distributed actor systems. Whether the references are to local or remote actors makes no difference to the code.

The first step towards making the messages work in across multiple machines was making them immutable. It would not be possible to change the content of a message after it has been sent from one machine to another. The remaining step towards making them fully self-contained is that the messages must be serializable so they can be transmitted and reconstructed by the actor system that receives the message. Once again, Scala case classes come to the rescue. As long as the properties within the case class can be serialized, the whole class can be serialized too.

Finally, the system will need some way to resolve references to actors in remote actor systems. You will learn how to do that in this section.

2.4.1 Distributing to Two JVMs

When the example moves from one to two JVMs, the actors and messages remain the same. What does change? The new version is shown in figure 2.6. The primary difference between

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

this and the example previously shown in figure 2.5 is that there are two actor systems. Each JVM needs its own actor system to manage its own actors.

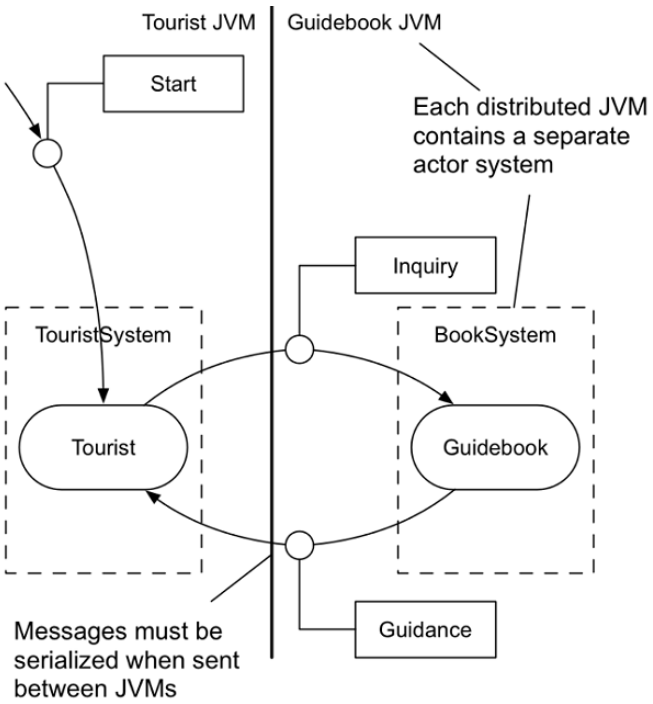

Figure 2.6 Actors communicating across local JVMs

If you cloned the original example from the git repository, you can use

```
git checkout two_jvm
```

to switch from the previous example using one JVM to this example with two.

2.4.2 Configuring for Remote Actors

As you might expect, distributing actors requires a little more setup than when everything is in one JVM. It requires configuring an additional Akka library called `akka-remote`. The affected files are:

- *build.sbt*—addsthe dependency on `akka-remote`.
- *application.conf*—provides some configuration information for remote actors.

The change from the previous `build.sbt` example is nothing more than the inclusion of the additional library, as shown here.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

Listing 2.7 build.sbt for remote actors

```
name := "Guidebook"

version := "1.0"

scalaVersion := "2.11.8"

resolvers += "Lightbend Repository" at "http://repo.typesafe.com/typesafe/releases/"

libraryDependencies ++= Seq(
  "com.typesafe.akka" %% "akka-actor" % "2.4.8",  (1)
"com.typesafe.akka" %% "akka-remote" % "2.4.8"    (1)
)
```

(1) Add dependency on remote actors. The Akka version numbers should match

The configuration file shown in listing 2.8 is read automatically by Akka during startup. The tourist and guidebook JVMs in this example are able to use the same configuration file. More complex applications would require separate configuration files for each JVM, but the example is simple enough that one can be shared. The syntax is "Human-Optimized Config Object Notation" (HOCON), which is a JSON superset designed to be more convenient for humans to edit.

Listing 2.8application.conf for remote actors

```
akka {
  actor {
    provider = "akka.remote.RemoteActorRefProvider"   (1)
  }
  remote {
    enabled-transports = ["akka.remote.netty.tcp"]    (2)
    netty.tcp {
      hostname = "127.0.0.1"                          (3)
      port = ${?PORT}                                 (4)
    }
  }
}
```

(1) Replace the default LocalActorRefProvider with the RemoteActorRefProvider
(2) Enable remote communication using the TCP protocol. Check the Akka documentation for other choices, such as Secure Sockets Layer (SSL) encryption
(3) The "remote" actors in the example will run on your local machine
(4) Obtain a port number from the PORT environment variable. If none is specified, it will default to 0 and Akka will choose one automatically

2.4.3 Setting up the Drivers

Now that the configuration steps are complete, the next step is to add a program to act as a driver for the guidebook actor system.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

THE GUIDEBOOK DRIVER

The driver for the guidebook actor system is a reduced version of the original driver for the entire system. Other than removing the tourist actor, the only change is to provide unique names for both the actor system and for the guidebook actor. The names will make it easier for the tourist actor to obtain an `ActorRef` to the guidebook. The complete listing is shown here.

Listing 2.9 Driver for the guidebook JVM

```
import akka.actor.{ActorRef, ActorSystem, Props}

object GuidebookMain extends App {
  val system: ActorSystem = ActorSystem("BookSystem")(1)

  val guideProps: Props =Props[Guidebook]       (2)
val guidebook: ActorRef =
[CA]system.actorOf(guideProps, "guidebook")(3)
}
```

(1) Name the actor *system* uniquely
(2) Produce an ActorRef the same as in the single JVM example
(3) Name the *actor* uniquely

Now that theguidebook driver is complete, you can move on to the tourist driver.

THE TOURIST DRIVER

The constructor for the tourist actor requires a reference to the guidebook actor. In the original example, this was easy because the reference was returned when the guidebook actor was defined. Now that the guidebook actor is in a remote JVM, this will not work. To obtain a reference to the remote guidebook actor the driver:

4. Obtains a URL-like path to the remote actor
5. Creates an `ActorSelection` from the path
6. Resolves the selection into an `ActorRef`

Resolving the selection causes the local actor system to attempt to talk to the remote actor and verify its existence. Because this takes time, resolving the actor selection into a reference requires a timeout value and returns a `Future[ActorRef]`. You do not need to worry about the details of how a future works. For now, it is sufficient to understand that if it resolves successfully, the resulting `ActorRef` is used just as it was in the single JVM example. The complete driver is as follows.

> **NOTE** The `scala.concurrent.Future[T]` used here is not the same as a `java.util.concurrent.Future<T>`. It is closer to, though not the same thing as, the `java.util.concurrent.CompletableFuture<T>` found in Java 8.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

Listing 2.10 Driver for the tourist JVM

```
import java.util.Locale


import akka.actor.{ActorRef, ActorSystem, Props}
import akka.util.Timeout            /
import tourist.TouristActor


import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.duration.SECONDS
import scala.util.{Failure, Success}


object TouristMain extends App {
  val system: ActorSystem = ActorSystem("TouristSystem") (1)


  val path =                                              (2)
    "akka.tcp://BookSystem@127.0.0.1:2553/user/guidebook"


  implicit val timeout: Timeout = Timeout(5, SECONDS)     (3)


  system.actorSelection(path).resolveOne().onComplete {   (4)
    case Success(guidebook) =>

      val tourProps: Props =                              (5)
[CA]Props(classOf[TouristActor], guidebook)               (5)
      val tourist: ActorRef = system.actorOf(tourProps)   (5)

      tourist ! messages.Start(Locale.getISOCountries)    (5)


    case Failure(e) => println(e)                         (6)
  }
}
```

(1) **Name the actor *system* uniquely**
(2) **Specify the remote URL path for the guidebook actor**
(3) **Wait up to 5 seconds for the guidebook to respond**
(4) **Convert the path to an ActorSelection and resolve it**
(5) **If the guidebook is resolved successfullly, continue the same as in the single JVM example**
(6) **If the guidebook fails to resolve, fail with an error**

At this point, you have configuration and drivers to run the original tourist and guidebook actors in separate actor systems on separate JVMs. Notice that the messages and actors are unchanged from the original example. This is not uncommon. Actors are designed to be distributable by default.

Now it is time to try them.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

2.4.4 Running the Distributed Actors

To run two JVMs, you will need two command prompts. Start by opening a terminal session just like you did previously for the single actor system in section 2.3. This time, the `sbt` command line will have to specify which main class to use, because there are two. Recall that the `application.conf` file in listing 2.8 specifies that the listener port should be read from the `PORT` environment variable, so you will have to specify that as well.

Because the guidebook will wait forever for actors to contact it, but the tourist will wait for only a few seconds to find a guidebook, the guidebook will be started first. The command line

```
sbt -D"PORT=2553" "run-main GuidebookMain"      ❶
```

❶ **The double quotes around the –D parameter are necessary on Windows but optional on other platforms**

should result in several messages to the console, ending with a log entry that tells you the book system is now listening on port 2553.

Next, open a second terminal window. The command line to run the tourist is almost the same

```
sbt –D"PORT=2552""run-main TouristMain"
```

The differences are the port number and the choice of main class to run.

If everything has gone as expected, the tourist should print the same guidebook information as in the original example. Congratulations! You have just created a distributed actor system.

As an exercise, try opening a third terminal and running another tourist on a different port number. It works. That is because the guidebook always responds to the *sender* of a message. It doesn't care whether there is one tourist or a thousand. If there are thousands of tourists, you might want to have more than one guidebook actor too. In the next section, you will learn how to do that.

2.5 Scaling with Multiple Actors

Shortly after Akka 2.0 was released in 2012, a benchmark (http://letitcrash.com/post/20397701710/50-million-messages-per-second-on-a-single) demonstrated sending 50 million messages per second on a single machine. That is far more than a single Guidebook actor would be able to handle. Recall that the actor system guarantees that no more than one thread has access to the actor at a time. Eventually, there would be too many incoming messages for a single actor and it would be necessary to have multiple guidebook actors to service all the requests.

Actor systems make adding multiple actors easy! An actor-based system handles scaling to multiple actors uniformly whether they are local or remote. The additional guidebook actors may run in the same JVM or in separate JVMs. In the rest of this chapter, you will learn how to put additional instances of the same actor in the same JVM, then you will learn how to scale

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

horizontally to another JVM. This is just a first taste of things to come. Chapter 4 will revisit these concepts in greater depth.

Before extending the actor system, it is worth taking a look at how traditional systems that do not use actors approach the same problem.

2.5.1 Traditional Alternatives

In a traditional system, scaling would be handled quite differently depending on the decision to put additional instances in the same JVM or to deploy them remotely. If they were in the same JVM, a system that does not use actors might instead use an explicit thread pool to balance requests, as shown in figure 2.7.

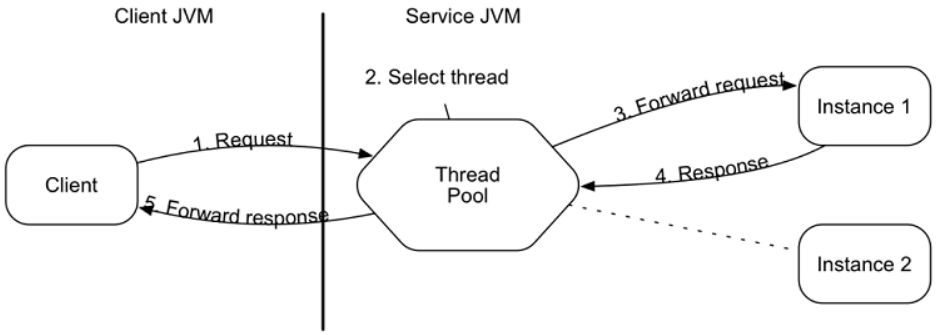

Figure 2.7 A thread pool can be used to manage access to multiple instances of a service.

If they were on separate JVMs, the system could use a dedicated load balancer that sits between the client and service, as shown in figure 2.8. In most cases, it is assumed that the communication through the load balancer uses HTTP as the protocol.

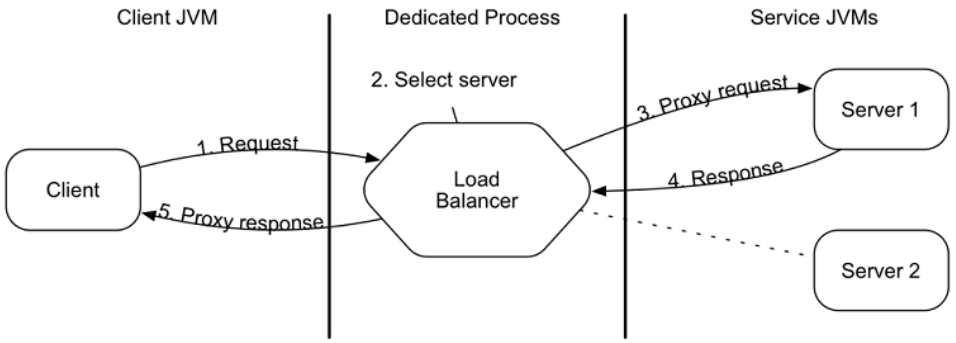

Figure 2.8 A traditional load balancer would introduce a separate process between the client and server.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

There are many excellent load balancer implementations available. HAProxy (www.haproxy.org) is a dedicated software solution, or NGINX (www.nginx.com) may be configured as a reverse proxy. Some companies even produce hardware solutions, such as the BIG-IP Local Traffic Manager from F5 Networks, Inc. Those solutions are all outside the scope of this book, because they are not necessary. Instead, load balancing is handled by the actor system.

2.5.2 Routing as an Actor Function

In an actor-based system, the load balancer can be treated as just an actor specialized for routing messages. The client treats the `ActorRef` to the router no differently than it would treat a reference to the service itself. You have already seen that local and remote actors are treated uniformly. That continues to hold true here. The client does not need to concern itself with whether the router is local or remote. That decision can be made as part of system configuration, independent of how the client or service is coded.

Returning to the guidebook example, the tourist actor sends an inquiry message to the router, the router selects a guidebook actor, and the guidebook sends the guidance directly back to the tourist, all as shown in figure 2.9.

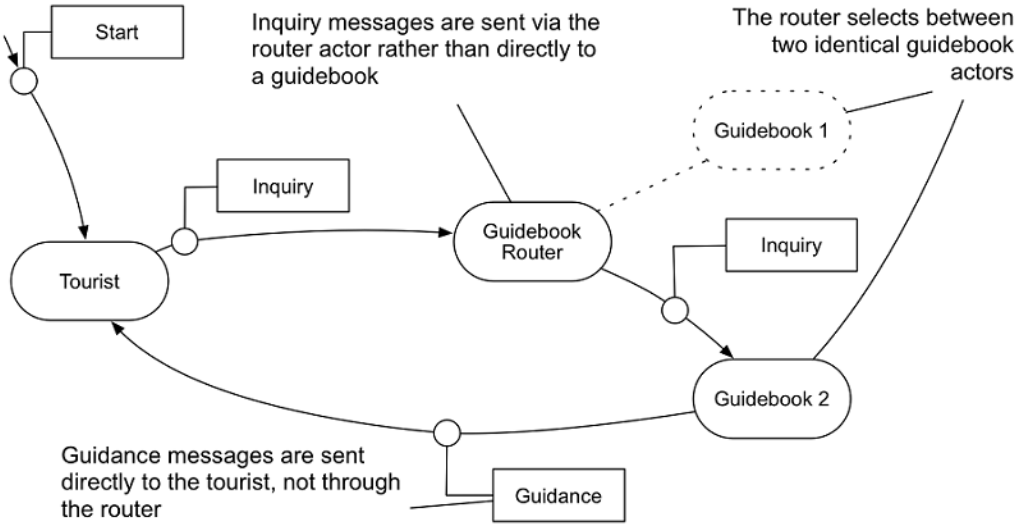

Figure 2.9 Sending messages from the tourist to the guidebook through a router that performs load balancing

Recall that the guidebook actor sends its response back to the sender of a message. You may wonder how that works when the message comes from the router rather than the original client. The answer is that the router does not pass a reference to itself as the sender. It

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

forwards the *original* sender, so the routed message appears to have come directly from the client.

2.6 Creating a Pool of Actors

A single actor, such as the guidebook example, handles only one request at a time. That greatly simplifies coding because the actor does not have to worry about synchronization. It also means that the guidebook can become a bottleneck, because there is only one of them and every request has to wait for it to become available. The simplest way to scale is to add a pool of guidebook actors within the a single actor system, and create a router to balance the inquiries. Figure 2.10shows this approach.

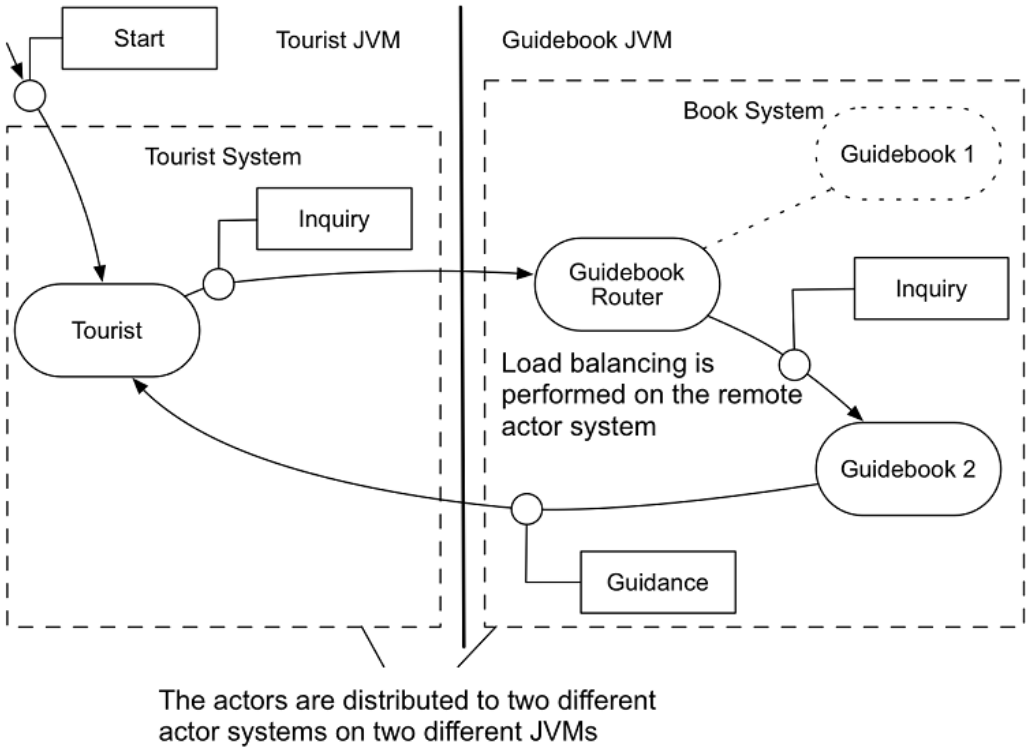

Figure 2.10 One way to scale is to create a pool of guidebook actors within the same actor system.

The `Tourist` and `Guidebook` actors remain unchanged from the previous examples. In fact, the entire tourist system remains the same. As you will see in the next section, only the guidebook system needs to be changed to incorporate the change.

If you cloned the original example from the git repository, you can use

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

```
git checkout pool
```

to switch to this example with two JVMs and an actor pool.

2.6.1 Adding the Pool Router

The pool router is an actor that takes the place of the original guidebook actor. Like any other actor, that means there needs to be a `Props` for the router actor. It is possible to configure an actor pool entirely within code, but preferable to use a configuration file. Akka routing includes a convenient `FromConfig` utility that tells it to do so. The driver in listing 2.11 passes the original guidebook `Props` to `FromConfig` so that Akka knows how to create new pool members, and everything else comes from the configuration file.

Listing 2.11 Driver for the guidebook JVM with a pool of guidebooks

```
import akka.actor.{ActorRef, ActorSystem, Props}
import akka.routing.FromConfig                          (1)

object GuidebookMain extends App {
  val system: ActorSystem = ActorSystem("BookSystem")

  val guideProps: Props = Props[Guidebook]              (2)

val routerProps: Props =
[CA]FromConfig.props(guideProps)                        (3)

  val guidebook: ActorRef =
[CA]system.actorOf(routerProps, "guidebook")            (4)
}
```

(1) Import library to read the pool configuration from application.conf
(2) Props for the guidebook actor are unchanged
(3) Wrap the pool configuration around the original Props for the guidebook actor
(4) The name of the actor must match the name in the configuration file

CONFIGURING THE POOL

Akka includes a number of built-in pool routers. One of the most commonly used is the round-robin pool. This implementation creates a set number of instances of the actor, and forwards requests to each in turn. Some of the other pool implementations are described in Chapter 4. Listing 2.12 shows how to configure a round-robin pool containing five instances of the guidebook actor. These instances are called *routees*.

Listing 2.12application.conf with a pool of guidebook actors

```
akka {
  actor {
    provider = "akka.remote.RemoteActorRefProvider"
    deployment {
      /guidebook {                  (1)
        router = round-robin-pool   (2)
        nr-of-instances = 5         (2)
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

```
      }
    }
  }
  remote {
    enabled-transports = ["akka.remote.netty.tcp"]
    netty.tcp {
      hostname = "127.0.0.1"
      port = ${?PORT}
    }
  }
}
```

❶ Configure a pool for the guidebook actor
❷ Use the built-in round robin pool implementation with 5 instances in the pool

As you can see, Akka makes it easy to create many actors in a pool. You create a pool actor, give it the `Props` needed to create new pool entries, and configure the pool as needed through the configuration file.

2.6.2 Running the Pooled Actor System

Running the pool of actors is easy too. It is the same as running the two actor system example shown previously in this section. As before, the command line

```
sbt -"DPORT=2553" "run-main GuidebookMain"
```

starts the guidebook system, and the command line

```
sbt –D"PORT=2552""run-main TouristMain"
```

starts the tourist system, which is completely unchanged. The output on the tourist console should be the same as before. The difference is found on the guidebook console. Before the pool was added, each inquiry resulted in the guidebook actor printing a line such as:

```
Actor guidebook responding to inquiry about AD
```

Now that the actor named "guidebook" is actually a router actor, each instance of the `Guidebook` actor in the pool is assigned a different, random name. The inquiries now result in each guidebook actor printing a line such as:

```
Actor $a responding to inquiry about AD
```

Because there are five actors configured, there should be five different names for the actor in the console output, such as $a, $b, $c, $d and $e.

> **NOTE** The round-robin pool is just one of a several pool implementations that are included with Akka. You can try some of the other types just by changing the configuration file. Other choices to try include random-pool, balancing-pool, smallest-mailbox-pool, scatter-gather-pool, and tail-chopping-pool.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

In this section, you scaled an actor system by replacing a single actor with a router to a pool of identical actors. In the next section, you will apply the same concepts to distribute messages across multiple actor systems on multiple JVMs.

2.7 Scaling with Multiple Actor Systems

The router actor is responsible for keeping track of the routees that handle the messages it receives. The pool routers in the previous section handled this by creating and managing the routees itself. An alternative approach is to provide a group of actors to the router, but manage them separately. This is similar to how a traditional load balancer works. The difference is that a traditional load balancer uses a dedicated process to manage the group membership and perform the routing, whereas in the actor-based system the routing may be performed by a router actor within the client, as shown in figure 2.11.

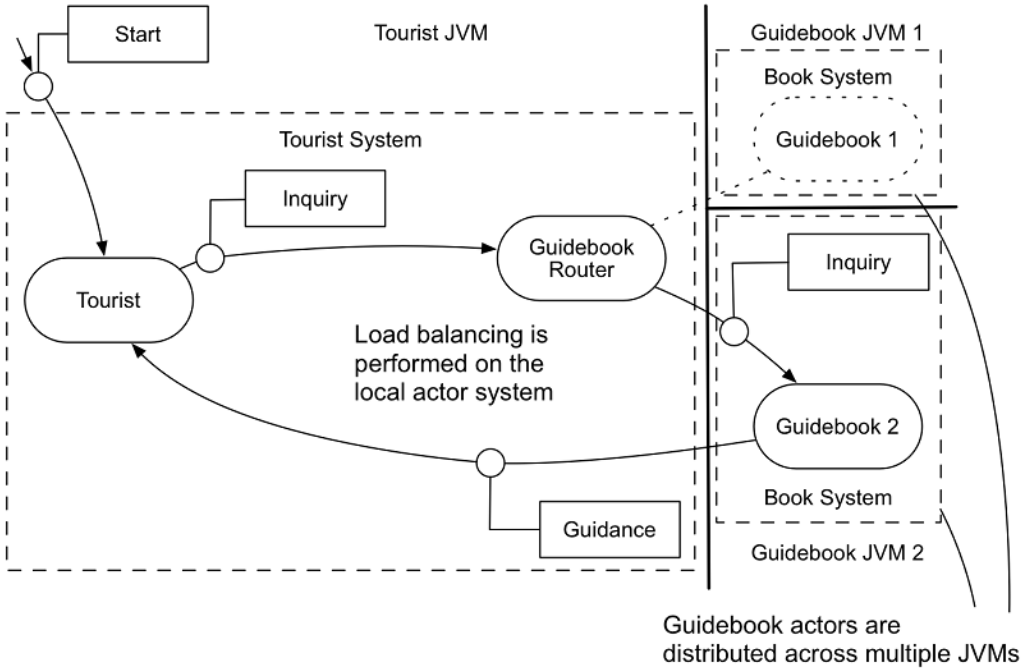

Figure 2.11 A local router can balance requests between actors on remote systems.

The two approaches are not mutually exclusive. It is reasonable to have a router on the client select which remote actor system to service the request, then have another tier of routing in the service actor system to select a specific actor instance from a pool. The response message still flows from the service actor directly to the original client actor.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

If you cloned the original example from the git repository, you can use

```
git checkout group
```

to switch to this example, which keeps the pool router on the guidebook systems and adds a group router to the tourist system.

2.7.1 Adding the Group Router

The driver for the tourist actor system is simpler with a group than it was in the initial system with a single remote actor. In the original example (listing 2.10), the driver resolved the remote actor path using a `Future[ActorRef]`, and the system waited for confirmation that the remote actor system had been verified before creating the tourist actor. With a group router, all of that is handled by the group router.

Listing 2.13 Driver for the tourist JVM with a group of guidebook systems

```
import java.util.Locale

import akka.actor.{ActorRef, ActorSystem, Props}
import akka.routing.FromConfig                          (1)

import Tourist.Start

object TouristMain extends App {
  val system: ActorSystem = ActorSystem("TouristSystem")

  val guidebook: ActorRef =
    system.actorOf(FromConfig.props(), "balancer")      (2)

  val tourProps: Props =                                (3)
    Props(classOf[Tourist], guidebook)                  (3)

  val tourist: ActorRef = system.actorOf(tourProps)     (3)

  tourist ! Start(Locale.getISOCountries)               (3)
}
```

(1) Import library to read the pool configuration from application.conf
(2) Use a different name to distinguish this router from the router pool used by the guidebook driver
(3) The remaining steps are the same as the single JVM driver shown in listing 2.6

Configuration of the router group is handled by the configuration file.

CONFIGURING THE GROUP

The group configuration for the balancer uses the round-robin-*group* rather than the round-robin-pool. Group routers expect the routees to be provided, and they are provided using `routees.paths`. Some of the other group implementations are described in Chapter 4.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

Listing 2.14 application.conf with a group of guidebook systems

```
akka {
  actor {
    provider = "akka.remote.RemoteActorRefProvider"
    deployment {
      /guidebook {                                          (1)
        router = round-robin-pool                           (1)
        nr-of-instances = 5                                 (1)
      }                                                     (1)
      /balancer {                                                 (2)
        router = round-robin-group                                (2)
        routees.paths = [                                         (2)
          "akka.tcp://BookSystem@127.0.0.1:2553/user/guidebook",  (2)
          "akka.tcp://BookSystem@127.0.0.1:2554/user/guidebook",  (2)
          "akka.tcp://BookSystem@127.0.0.1:2555/user/guidebook"]  (2)
      }
    }
  }
  remote {
    enabled-transports = ["akka.remote.netty.tcp"]
    netty.tcp {
      hostname = "127.0.0.1"
      port = ${?PORT}
    }
  }
}
```

(1) Leave alone the pool for the guidebook actor. It will continue to be used by the guidebook.
(2) Create a round-robin group router named "balancer" with three group members.

The rest of the configuration file remains the same.

2.7.2 Running the Multiple Actor Systems

The configuration in the previous section instructed the balancer to look for three guidebooks by contacting the `BookSystem` actor systems listening on ports 2553, 2554 and 2555. Open three terminal windows and start those three systems using the following commands:

```
sbt -DPORT=2553 "run-main GuidebookMain"   (1)
sbt -DPORT=2554 "run-main GuidebookMain"   (1)
sbt -DPORT=2555 "run-main GuidebookMain"   (1)
```

(1) Run each of these commands in a separate terminal

Next, run the tourist system on a fourth terminal:

```
sbt –DPORT=2552"run-main TouristMain"
```

You should see the usual guidance on the tourist console. Next, switch to each of the terminal windows running the guidebook instances. You should see that all three of them have responded to some messages, and be able to verify that they each received requests for different countries.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

The pools within each of these actor systems are still in place too, so the distributed actor system you are running now includes 16 actors: 1 tourist actor plus 3 guidebook systems that each have a pool of 5 guidebook actors. That's quite a bit for a such a small amount of code.

2.8 Applying Reactive Principles

The same tourist and guidebook actors have remained unchanged throughout this chapter. At this point, both of the driver programs take their configuration from `application.conf` rather than having anything hardcoded. The complete system requires surprisingly little code. The two actors and their drivers are less than 100 lines of Scala. Yet, this system still exhibits reactive attributes.

The four attributes that lay the foundation for reactive applications were introduced through the Reactive Manifesto in the first chapter. Reactive applications are message-driven, elastic, resilient and responsive.

The examples in this chapter are *message-driven*. Everything is in response to the same three immutable messages. All communication between actors is accomplished asynchronously, and there is never a direct function call from one actor to another. Only the actor system calls the actor to pass it a message. The message passing exhibits location transparency. The sender does not concern itself with whether the recipient of a message is local or remote.

Location transparency also allows routers to be injected into the message flow. This helps achieve the another reactive attribute. The application is also *elastic*. It is capable of applying more resources by expanding a pool of local actors, and capable of expanding remotely by adding remote actors.

In other words, you now have an elastic, message-driven system that you can scale horizontally without writing any new code. That is quite an accomplishment. Feel free to experiment with the configuration!

If you experiment, you will find that that the system has some attributes of resilience and responsiveness, but it could be better. The next two chapters will show you the additional pieces needed to create a fully reactive application. You will learn more about the design of Akka, and how the components work together to deliver the underpinnings of the system you just created.

2.9 Summary

In this chapter you learned that

- Actors have a *receive* function that accepts a message and does not have a return value.
- Actors are called by the actor *system*, not directly by other actors. The actor system guarantees there is never more than one thread at a time calling an actor's receive function. This simplifies the receive function because it does not have to be thread safe.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

- Actors are distributable by default. The same actors and the same messages were used throughout this chapter. Only the drivers and configuration was changed as the example evolved from a pair of actors exchanging messages within a single JVM all the way to 16 actors distributed across 3 JVMs.
- Immutable messages flow between actors. Immutable messages are thread safe and safe to copy, which is necessary when a message is serialized and sent to another actor system. Scala case classes offer a safe and easy way to define immutable messages.
- Senders address messages using an `ActorRef`, which is obtained from the actor system. An `ActorRef` may refer to a local or a remote actor.
- Actor systems can be scaled using a router to balance requests among multiple actors. A router may be configured as a *pool* that creates and manages the actor instances for you. A *group* router requires the actors to be created and managed separately.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

3

What is Akka?

This chapter covers:

- **The Actor model**
- **The Actor system**
- **Akka is concurrent and asynchronous**
- **Share nothing design**
- **Non-blocking design**
- **Akka Supervision and Routers**

Now that we have a good understanding of what a reactive application is and why you need it, it's time to pick a toolkit for building such systems. But, before we do, let's quickly review of the traits of a reactive system to make sure the tools we pick are up to the job. Recall from chapter one, the Reactive Manifesto lays out four key traits that identify a reactive system:

- Responsive - The system responds in a timely manner if at all possible. Responsiveness is the cornerstone of usability and utility, but more than that, responsiveness means that problems may be detected quickly and dealt with effectively.
- Resilient - The system stays responsive in the face of failure. Resiliency applies not only to highly available, mission critical systems, any system that is not resilient will be unresponsive after a failure.
- Elastic - The system stays responsive under varying workload. Reactive systems can respond to changes in load by increasing or decreasing the allocation of resources.
- Message Driven - Based on asynchronous communication where the design of sender and recipient are not affected by the means of message propagation. As a result, you can design your system in isolation without worrying about the how of transmission. Message driven communication leads to loosely coupled design that provides the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

context for responsiveness, resiliency and elasticity.

In taking a close look at these four traits, you will notice that there is an interesting distinction about one of them: Message Driven. The first three, Responsiveness, Resiliency, and Elasticity are more about how a reactive system behaves, while Message Driven operates in a supportive role, allowing the first three traits to do their job. You can think of it in terms of implementation: we implement an asynchronous message-driven architecture to promote responsive, resilient, and elastic behavior. Therefore, one of the key abilities of any toolkit we choose is: it should support asynchronous message passing.

While there are other toolkits that support message passing, the one we chose for this book is Akka[6]. The basis for this decision is the distributed nature of Akka, as well as the authors' experience with it in production. While we will dig into Akka and its many features, this chapter is not intended to be exhaustive. We'll use analogy, explanation, and some code examples to show Akka's reactive nature. The foundation in Akka we give you here will set you up to understand all the Akka we use later in the book. Later in the book, you'll learn clustering & sharding, CQRS, event-sourcing, micro-service based design, distributed testing, as well as other technologies to round out your reactive learning.

So, what is Akka and how does it qualify as a reactive toolkit?

3.1 What is Akka?

Akka is the brainchild of Jonas Bonér, CTO & Co-Founder of Typesafe. Being an experienced enterprise Java architect and competent with distributed application technologies, such as CORBA, EJB and RPC (all heavyweight protocols with concrete local and remote interfaces), Jonas grew frustrated with the older technology limitations around scalability and resilience, along with their synchronous nature. He began to realize these technologies and the standards and abstraction techniques used to approach distributed computing on the *Java Virtual Machine (*JVM) were not going to work as the level of distribution increased.

Not one to give up, Jonas began looking outside the Java enterprise space for answers, where he came upon Erlang and Erlang OTP-style (Open Telecom Platform) supervisors. He realized that the approach used by Erlang for failure management and distribution of critical infrastructure services such as Telecom could be applied to mainstream enterprises as well.

One component of Erlang in particular captured his attention: the Actor Model, a mathematical model of concurrent computation where decisions are made locally within an actor and then transmitted as a lightweight message. From the Actor Model, he realized it was possible to build loosely coupled systems with "Let-it-Crash" semantics that embrace failure and allow deterministic reasoning in a multi-threaded environment.[7] For Jonas, this was a

[6] Programming with actors - http://en.wikipedia.org/wiki/Actor_model
[7] Akka 5 Year Anniversary - http://goo.gl/WXRYxI

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

"light-bulb" moment and after several months of "hAkking." he released the first version of Akka, v0.5 on June 12th, 2009.[8]

What is hAkking?

If you've read anything about Akka, the term "hAkking" is probably familiar, if not, you can most likely guess what it means. hAkking is an affectionate quip for coding or hacking in Akka, and was first coined sometime back in 2009.

AKKA'S ENDURING DESIGN

As a testiment to the enduring design off Akka, the following snippet from the v0.5 README.md still holds true today:

The Akka kernel implements a unique hybrid of:

- The Actor model (Actors and Active Objects)
- Asynchronous, non-blocking highly concurrent components.
- Supervision with "let-it-crash" semantics. Components are loosely coupled and restarted upon failure.

What is a Kernel?

A kernel in computer science is the central part of the operating system that manages tasks of the computer and hardware, and is the most fundamental part of the system. Kernels come in two forms:

- Microkernel: contains only core functionality
- Monolithic kernel: contains core functionality and many drivers also.

Monolithic kernels are primarily useful for the core of an operating system, such as Linux, and extensible through a set of drivers. Microkernels, on the other hand, focus on specific problems such as controlling memory or CPU cycles in the case of an operating system. The term Microkernel often applies to more than one OS, as is the case with Akka. In Akka, the microkernel is a bundling mechanism that allows you to deploy and run your application without the need of an application server or launch script.

Erlang & Erlang OTP-style supervisors

Akka drew much of its inspiration from Erlang. Erlang is a programming language used in telecoms, banking, and other industries to build scalable, soft (a systemic approach for tackling real-world problematic situations), real-time systems that are meant to have high availability. It supports the traits of the Reactive Manifesto and achieves this through the Erlang OTP libraries. OTP stands for Open Telecom Platform, and is designed to be middleware that provides the tools

[8] Akka v0.5 - https://github.com/akka/akka/releases/tag/v0.5

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

necessary for development. From the Akka perspective, OTP is best known for its Actor implementation with Supervision Trees: that provides the semantics for self-healing that makes both Erlang and Akka resilient.

3.2 Akka Today

In today's world, the demand for distributed systems has exploded. As customer expectations such as an immediate response, no failure, and access anywhere increase, companies have come to realize that distributed computing is the only viable solution. In turn, the requirements placed on distributed technologies have become more and more rigorous. Things like application clustering, distributed persistence, distributed caching, and Big Data management are fast becoming expected members of any competent toolkit.

To satisfy these increasing requirements, toolkits like Akka must continually react by enhancing existing tools and adding new ones. As a result over the last couple of years, the Akka team has added some of the highest quality distributed tools available to the toolkit. The following subsections give a high-level overview of several of those tools, categorized by Reactive Manifesto trait, many of which we will discuss in detail throughout the book.

3.2.1 Message Driven

THE ACTOR SYSTEM

The core Akka library akka-actor provides all the semantics for the message passing, serialization, dispatch and routing. Through the use of location transparency (described next), the actor paradigm becomes elastic.

3.2.2 Resilient and Elastic

IMPROVED REMOTING

Akka adopts a Distributed by Default mentality through a unified programming model with referential or location transparency. Local and remote scenarios use the same API and are distinguished by configuration, allowing a more succinct way to code Message-driven applications.

AKKA CLUSTER

Akka Cluster, a loosely coupled group of systems that present themselves as a single system, provides a resilient decentralized peer-to-peer based cluster membership service with no single point of failure or single point of bottleneck. It does this through using gossip protocols and an automatic failure detector.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

CLUSTER AWARE ROUTERS

Akka's Cluster Aware Routers take Akka Routers to the next level by automatically re-balancing load whenever a new member joins the cluster.

CLUSTER SINGLETON

Sometimes we need exactly one instance of a process running in a cluster, for example, a single point of entry to an external system. Akka provides the Cluster Singleton for use-cases such as these.

CLUSTER SHARDING

Cluster Sharding allows you to interact with actors that are distributed across several nodes in a cluster by a logical identifier. This access by logical identifier is important when your actors in aggregate consume more resources than can fit on one machine. For example, a system may have

AKKA PERSISTENCE

Persistent state management is a staple for most applications. Akka Persistence not only provides this, but also automatic recovery when a system restarts due to failure or migration. In addition, Akka Persistence provides the foundation for building CQRS and Event Sourced based system.

3.2.3 Responsive

FUTURES AND AGENTS

Seamless integration with Futures (data structure used to retrieve the result of some concurrent operation) and Agents (allow safe asynchronous change at a single location). As a result, we can synchronously reason about asynchronous computations and safely modify shared state.

ROUTERS AND DISPATCHERS

The Routers and Dispatchers implementation in Akka provides a powerful mechanism for parallel programming.

AKKA STREAMS

Akka Streams is a recent addition to the Akka Toolkit. Their goal is to govern the exchange of stream data across asynchronous boundaries by managing backpressure. This technology is so profound that many of the other tools in Akka such as Akka Persistence, Spray.io (now Akka HTTP) and products like the Play framework will be updated to take advantage of Akka Streams.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

AKKA HTTP

Akka HTTP based on the Spray.io library and will provide an HTTP request/response model that incorporates streamed data on demand. Relying on Akka Streams under the covers, back pressure is implicitly managed bringing significant enhancements in Responsiveness to the HTTP paradigm.

AKKA DATA REPLICATION

Akka Data Replication is Akka's answer for CRDTs, *Conflict Free Replicated Data Types*. CRDTs allow for replicated in-memory data storage that is eventually consistent has low latency and full availability. You can think of them as a distributed cache.

3.2.4 Big Data

SPARK

Spark is a library that turns data operations on their head by moving the computation to the data rather than having the computation read the data, and for that reason is very performant for data intensive operations and transformations. Big data management is quickly becoming one of the most important requirements placed on distributed systems. Fortunately, there are technologies like Spark (a Big data streaming compute engine) a Scala application that leverages Akka. While not officially part of the Akka toolkit, feels right at home in any reactive environment due to its Akka foundation.

Today, Akka is defined as a toolkit and runtime for building highly concurrent, distributed, and resilient message driven applications on the JVM. Let's take a look at the philosophy behind this definition and establish some terminology that will help us tie Akka's features to the reactive paradigm as we move through the chapter.

Where did Akka get its name?

Akka (the Swedified form of Ahkka from the Sámi language) is a mastiff (a compact group of mountains, distinct from other groups) in the northern part of Sweden. It contains a total of ten glaciers and twelve peaks, with Stortoppen being the highest.

Akka is also the name of a goddess from Swedish mythology. She is the goddess that stands for all the beauty and good in the world. The mountain can be seen as the symbol of this goddess.

3.3 Akka terminology

Now that we have been introduced to Akka's origins, and seen some of its advanced features let’s dig into the methods employed by that Akka make it reactive, by defining some important terms.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

3.3.1 Concurrency and Parallelism

Concurrency and parallelism are sometimes confused to mean the same thing, but while related, there are some differences.

- *Concurrency* improves throughput by allowing two or more tasks to make progress in a non-determined fashion, which may or may not run simultaneously. As a result, concurrent programming focuses on the complexity that arises from that non-deterministic control flow, as we discussed in chapter one under tightly coupled middleware.
- *Parallelism*, on the other hand, occurs when the execution of the tasks happen simultaneously, parallel programming focuses improving throughput by making flow control deterministic.

The following figure will help in visualizing their differences.

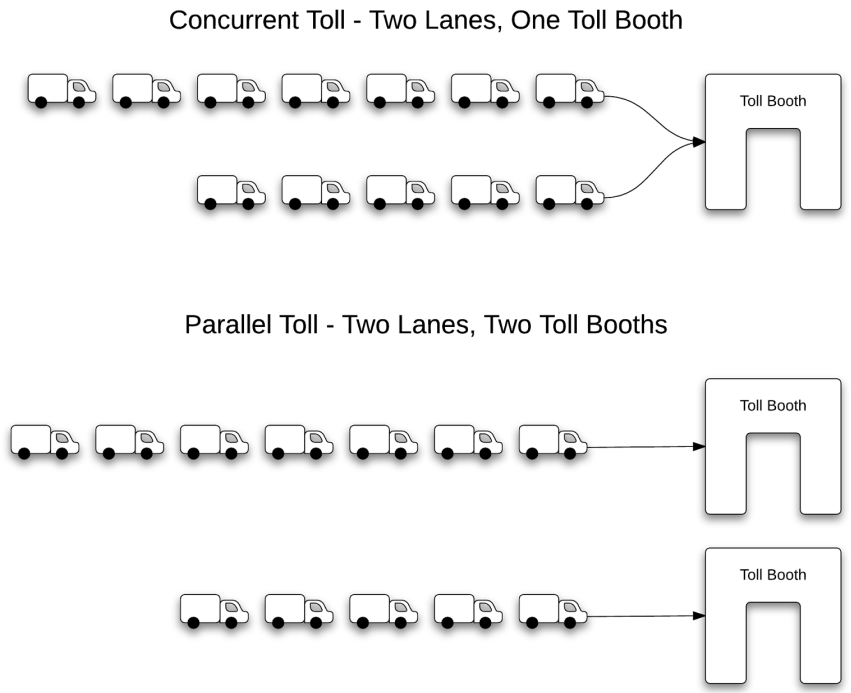

Figure 3.1 Concurrency and Parallelism Compared

The figure clearly shows how a single toll booth becomes a blocker and slows down the entire line of traffic. Adding a second toll booth removes the block and allows much greater traffic flow.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

Both concurrency and parallelism are key to reactive systems, as they promote responsiveness, support elasticity, and are part of the Akka DNA. On the flip side, they introduce non-determinism and isolated processing which require management in the aggregate, meaning that there are no guarantees as to exactly where and when a process will occur. As we work through the book, we will show you how Akka leverages these constructs, and the best way to implement and manage them through the application we will build.

3.3.2 Asynchronous and Synchronous

While concurrency and parallelism are more about the "how" of processing throughput, asynchronous and synchronous are more about accessing that throughput. A method is said to be *synchronous* if the caller of that method must wait for a return or failure condition. The opposite is true for asynchronous. An *asynchronous* method does not require its caller to wait. If the caller does wish a response, then a form of signal completion is required, such as a callback, future, or message passing. Let's explore some common forms of signal completion for asynchronous methods.

3.3.3 Blocking, Registered Callback, Future, and Message Passing

Blocking is when the sender wants to wait for the result of an asynchronous call by wrapping it in a blocking construct. This block stops all processing of not just the sender, but the entire thread which most likely has other processes running. As one might expect, this is not a good idea as it can have nontrivial performance implications and is a major road block (no pun intended) to elasticity. One simply cannot go anywhere (scale up or out) if something is blocking their way. Blocking in this context should be avoided unless absolutely necessary and is frowned upon in a message passing toolkit such as Akka.

Registered callbacks are a construct whereby the caller gives the asynchronous method with the means by which to signal completion. As a result, the caller does not have to wait for completion and may continue processing. When the asynchronous method is complete, it will use the callback to "push" the signal the caller. Akka uses callbacks with its futures construct.

A *future* is a data structure that is used to capture the result of a concurrent operation and provides a callback for its access. In regards to asynchronous methods, the future is usually used in one of two ways. The caller wraps the asynchronous method in the future, or the asynchronous method returns one. In either case, this allows the caller to continue processing and "pull" the signal when the asynchronous method is complete. Futures are a great way compose a sequence of events that are not synchronous in nature and help with responsiveness. Akka has built-in support for futures.

We will talk about message passing in detail in the next section when we discuss the Akka actors. For now, understand that message passing is at the core of any actor system, and provides actor-based semantics to call an asynchronous method, by sending the owner of that method a message. The means of that transmission are transparent to the caller, which in actor terms is the sender. Message passing is the primary way of communication in an Akka

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

system and will become second nature to you by the completion of this chapter. Following is a diagram that helps visualize the different forms of signal completion.

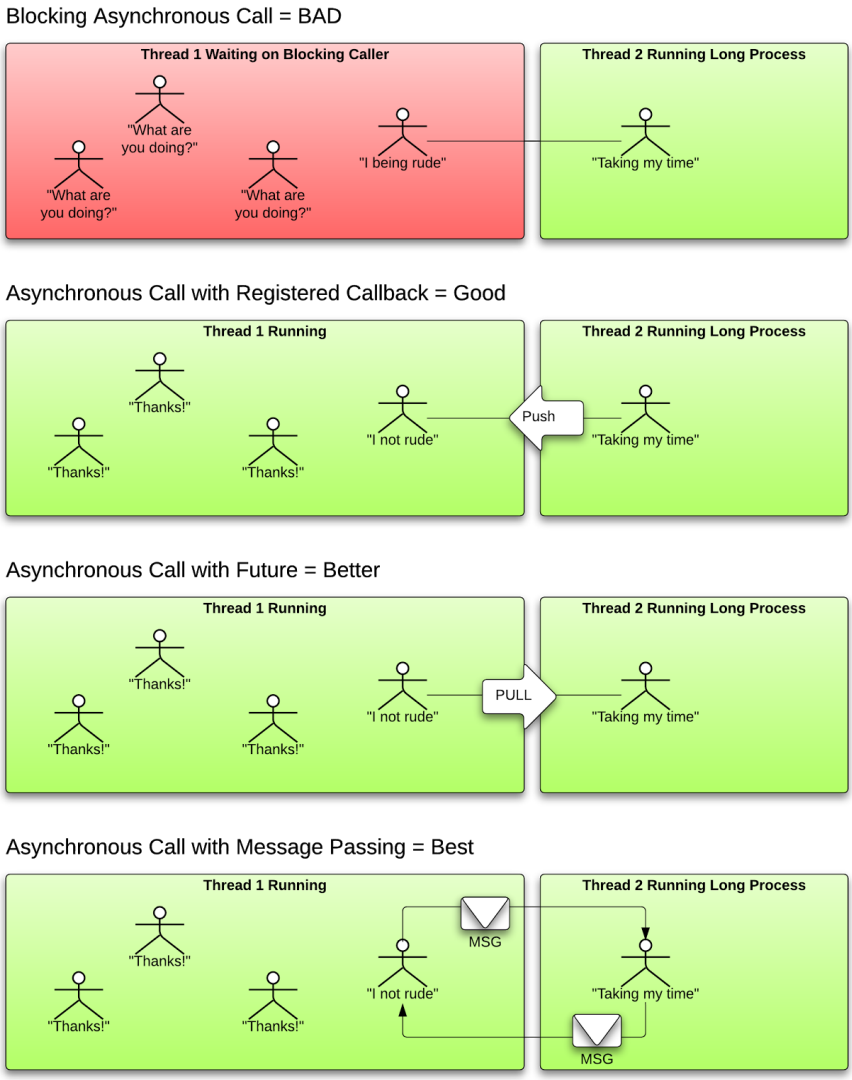

Figure 3.2 Different forms of signal completion

Asynchronous method calls provide a significant boost to responsiveness and, as a result, are key to reactive systems and part of the Akka core. The caller, or in the case of actors, the sender, does not wait for a response, therefore, is free to deal with other concerns. However,

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

using this form of communication introduces the need to reconcile the results independent its transmission. We have seen that blocking against asynchronous calls (one way to reconcile) is a major no-go as it hobbles responsiveness and kills elasticity. The good news is that we have also seen through other methods like callbacks, futures and message passing we can manage asynchronous calls without impedance.

So far, we've discussed terminology around how Akka provides the semantics for building applications that are responsive and support elasticity. But what did we mean by "support" elasticity? You will recall; elasticity is the notion that a system stays responsive under varying load through increasing and decreasing the allocated resources. Another way to say this is the responsiveness of the system is proportional to the systems ability to give and take resources. This allocation can occur both locally in the case of vertical scaling as well as horizontally in the case of distribution.

As a result of this definition, one can surmise the two concepts are interrelated in a reactive system. In addition to being interrelated, they also share a common enemy that if not managed properly, may significantly impede their abilities. Let's explore this common enemy, contention and see how can deal with it.

3.3.4 Contention

Contention, or in computer science terms, resource contention, is the conflict that arises over access to shared resources such as random access memory, cache memory, disk access and the like. You briefly saw contention earlier when two lines of cars were attempting to share the same toll booth. Contention at this level is primarily the responsibility underlying operating system, therefore, we usually don't have to deal with it. There is however another level where contention arises that we must concern ourselves with, the application level. At the application level, contention almost always results from mismanagement of shared state across multiple threads. This condition results in what we know as:

- deadlock
- livelock
- starvation
- race condition

A *deadlock* is a situation where several participants are frozen, waiting for each other to reach a specific state. We call this a "catch-22". All participants essentially stop progress and the entire sub-system stalls.

A *livelock*, similar to a deadlock is again a scenario where all participants effectively make no progress. The primary difference is that instead of being frozen, participants in a livelock situation continually change their state as to allow other participants to proceed.

Starvation, on the other hand, is different but in the aggregate has the same effect. Starvation is a scenario where some participants are always given priority over others resulting in the others never able to make progress.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

Race conditions are particularly nasty because they give the illusion of progress. A race condition occurs when multiple participants have access to unguarded shared mutable state, and the mutation of that state is assumed to be deterministic. This results in interleaved changes that may not be in the correct order, corrupting the state and can lead to unexpected results. The illusion is that while progress is being made, it's most like erroneous and in the end no good.

Figure 3.3 is a good illustration of the different types of contention scenarios.

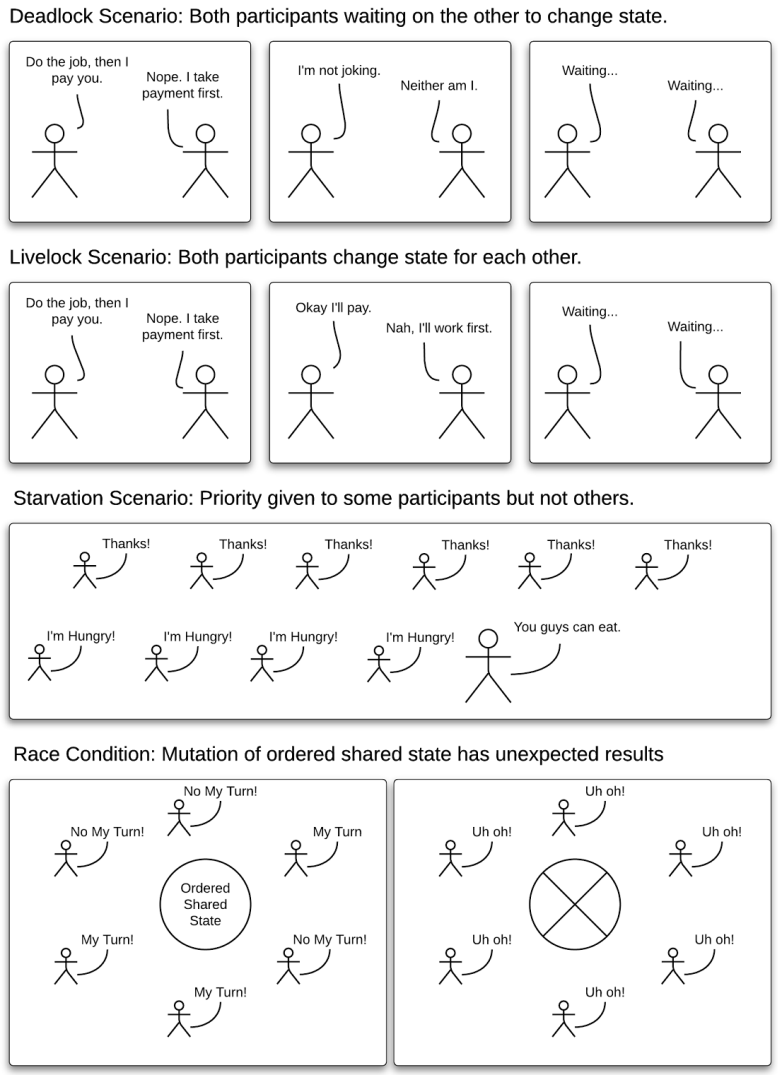

Figure 3.3 Different Types of Contention

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

As you can see, contention is a problem and one to avoid as much as possible. It has significant consequences on responsiveness and in turn constrains are capability to be elastic. So how do we get around this problem? Surely we will need concurrent processing and to manage state that in some sense is shared. We are talking about reactive systems that embrace distribution are we not? We are, and the answer is share nothing!

3.3.5 Share Nothing

Almost every application written requires some form of state and that state more often than not needs access by multiple participants. The canonical example is a concurrent counter where we track the number of visits to a given resource that updates correctly without blocking, so we stay responsive. In a language like Java, we could use construct like `AtomicInteger` to solve this problem. `AtomicInteger` provides a safe way to mutate our counter's value in a concurrent environment. The inference here is all mutations occur atomically. In other words while they occur, there is no interference from the outside. But what happens when we have a more complex state model like an employee. An Employee state model may contain multiple variables like firstName, lastName, address, as well support for differents states such as active, inactive, etc. All of these variables and state transitions will need access by are multiple participants. Unfortunately, there is no `AtomicEmployee` in Java so what do we do?

One approach is to descend into the world of concurrent programming through the use of synchronization and locking constructs. While this approach is quite common for dealing with state models in a concurrent environment, it is very tedious and notoriously difficult to debug and can cause contention. The challenges and details of this approach are beyond the scope of this book, but there are a great number of resources available if you're interested.

Another approach to managing state and the one we will stress throughout the book is to share nothing. Share nothing may sound a little extreme at first, but it's not what you might think. When we say share nothing, we are talking about three concepts:

- isolation by single source of truth,
- encapsulation of mutation, and
- immutable state transference.

Let's explain what we mean by these terms.

ISOLATION BY SINGLE SOURCE OF TRUTH

Isolation by single source of truth is a design pattern where we isolate a state model within a single authority, and that authority is solely responsible for all mutations including creation. This pattern guarantees that there is always only one record of truth and minimizes contention through encapsulation.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

ENCAPSULATION BY DEFINITION

Encapsulation by definition means we have restricted access to our state model by anyone other than the authority. As a result, one has to ask; How then do the participants use the state model? The answer is defensive copy.

Defensive copy is the notion that any time the authority receives a request; it first makes an immutable copy of its state model. In the case of mutation, this is done by creating a new instance by applying the mutation through constructor semantics that replaces the original. We will talk about constructor semantics later in the chapter. The authority also ensures that all mutations process in sequential order by reconciling or discarding ones that did not due to asynchronicity. In the case of a non-mutating request, such as a get, the authority essentially does the same but does not need to replace original due to the lack of mutation. Defensive copy allows access to state in a safe fashion and through immutable transference guarantees outsiders cannot fiddle with our state.

IMMUTABLE STATE OF TRANSFERENCE

Immutability transference is really the only safe way share state. If a state structure is immutable, then it's impossible for anyone to alter it, and we are free to share without worry of contention. In message passing, this is how we guarantee that the message we send is the same message our recipient receives. By making the message immutable, we assure its accuracy and eliminate the possibility of contention.

This approach of share nothing may seem cumbersome especially when considered in light of CRUD semantics that historically rely on object relational mappings (ORMs). That being said; we will show you an alternative in chapter five known as event sourcing that we first introduced at the beginning of the book. We believe that not only is event sourcing a great fit for reactive environments it's a superior way to reason about state and the management thereof.

We've seen that with Akka, through the use of concurrency, parallelism, and asynchronicity, we have the foundation for building responsive applications that support elasticity. We've also seen that we must manage these concepts to avoid contention by adopting a share-nothing mentality. So far, we've got a good start for establishing our toolkits credentials in regard to reactive behavior. There is a lot more to cover, especially in regard to distribution, but we'll get into that in the next chapter. Next, let's take a look at the core of Akka, actors.

3.4 The Actor Model

Actors are not new, having been around in various forms since the 1970s. In fact, Scala included an earlier more primitive version of actors before Akka fully matured and became a replacement for them. What are actors? *Actors* are small capsules of programming logic that contain behavior and state, and communicate via message passing. In Akka, actors are built

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

on top of two primary concepts: the actor model and the actor system. Let's start by taking a look at what the actor model is.

According to its inventors, the *Actor Model* is defined as a mathematical theory of computation that treats actors as the universal primitives of concurrent digital computation, inspired by the theory of general relativity and quantum mechanics.[9] Okay, we know that may sound a little scary, but in reality it's not that complicated, at least at the application level.

The primary purpose behind an actor is to provide two concepts:

- A safe, efficient way to reason about computations in a concurrent environment, and
- A common means to communicate in a local, parallel, and distributed environment.

To better understand how Akka actors support these concepts, let's take a look at the main components an actor contains:

- State
- Actor reference
- Asynchronous message passing
- Mailbox
- Behavior and the Receive loop
- Supervision

3.4.1 State

The first thing we want to look at is the state model, as we've discussed the challenges behind its management in a concurrent environment. Actors provide the semantics to encapsulate state, such that the actor becomes the single source of truth for its state model based on the share-nothing mentality.

Essentially Akka isolates each actor on a light-weight thread that protects it from the rest of the system. The actors themselves (housed inside a light-weight thread) run on a real set of threads, where a single thread may house many actors with subsequent invocations for a given actor occurring on a different thread.

Under the covers, Akka manages all the complexities of these concurrent interactions removing the need for us to use synchronization and locking semantics for concurrency. The result: we now have a safe and efficient way to reason about computation in a concurrent environment.

3.4.2 Actor Reference

In the original Scala actor implementation, actors were instantiated directly with concrete reference. This concrete reference required a different implementation and API for local and

[9] Carl Hewitt; Peter Bishop; Richard Steiger (1973). "A Universal Modular Actor Formalism for Artificial Intelligence"

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

remote actors, and in the end became unwieldy. Akka chose another approach to designate instances of actors known as `ActorRef`. Through the use of `ActorRef`, there is no need for different implementations resulting in a single API for both local and remote.

The `ActorRef` is an immutable handle to an actor that is serializable, and may be local or on a remote system. Functionally, an `ActorRef` operates as a kind of proxy mechanism for the actor it represents. This proxy behavior is important in the context of remoting, as the `ActorRef` can be serialized and sent across the wire on behalf of its actor. For the remote actor acquiring the `ActorRef`, the location of the real actor (the one the `ActorRef` represents) is transparent. This feature of Akka is called *location transparency,* which we'll talk more about in the next chapter. Suffice it to say, it's key to resilience and elasticity.

3.4.3 Asynchronous Message Passing

One of the four traits of the Reactive manifesto is that reactive architectures should be message driven, and this is exactly how actors communicate. Using asynchronous message passing semantics, actors interact with one another by sending a messages to each other's `ActorRef`. This design allows for a loosely coupled systems that support elastic scaling whether out or up. Akka actors support two operators, or in the case of Java, two methods for sending messages:

- `!` (Scala) or `tell` (Java) is used to asynchronously send a message. This is often called fire-and-forget.
- `?` (Scala) or `ask` (Java) is used to asynchronously send a message and expect a reply in the form of a `Future`.

Behind Akka's message passing semantics, there are two rules: *at-most-once* delivery and *message ordering per sender-receiver pair*. We will explain the first one here and the second in the next section on The Mailbox.

- To explain *at-most-once delivery*, we need to do in the context of delivery mechanism categories of which there are three:
- *At-most-once* delivery means that for each message handled, it will arrive at the recipient zero or one times. The inference here is that during transit, the message may get lost.
- *At-least-once* delivery means that for each message handled, multiple send attempts may occur until delivery is successful. The inference here is that the recipient may receive duplicate messages.
- *Exactly-once* delivery means that for each message handled, the recipient will receive one and only one copy. The inference here is that the message will not be lost or duplicated.

You may be asking: "Why at-most-once? I want a guarantee." Akka does support *at-least-once* delivery, through its persistence library, which we will explore in the next chapter, but to

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

address the question, let's first explain the cost of each one of these methods in Table 3.1 below.

Table 3. 1 Message Delivery Methods Comparison

Costs	Delivery Method
• Least expensive • Performs the best • Lowest implementation overhead • Does not require state management due to fire-and-forget semantics	at-most-once
• More expensive • Performance varies based on recipients acknowledgement • Medium level implementation overhead • Requires sender state management for message reconciliation • Requires recipient to acknowledge for sender reconciliation	at-least-once
• Most expensive • Wort performance • Highest level of implementation overhead • Requires sender state management for message reconciliation • Requires recipient to acknowledge for sender reconciliation • Requires recipient state management for message reconciliation	exactly-once

In a distributed environment, we want to minimize the overhead as much as possible, so we can stay responsive. One of the most costly forms of overhead is the management of communication state that the latter two delivery mechanisms require. The topic of guaranteed message delivery is complex and beyond the scope of this book, but if you're interested, there is a great post on InfoQ titled *Nobody Needs Reliable Messaging* [10]. The Akka documentation also has a great section about this under *Message Delivery Reliability* [11].

3.4.4 The Mailbox

Actors communicate by passing messages back and forth much like we do with email or when we send letters via the postal system. In both cases, we need a way to receive our messages, and just like us actors use a mailbox. Each actor has exactly one mailbox that enqueues all messages in the order received. This order of reception has an interesting implication in a

[10] http://www.infoq.com/articles/no-reliable-messaging
[11] http://doc.akka.io/docs/akka/current/general/message-delivery-reliability.html

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

concurrent environment that may not be apparent. For a given sender, the recipient will always receive the messages sent in the proper order. This is the *message order per sender-receiver pair* we mentioned above.

The same is not true for messages sent in aggregate. If a single recipient receives messages from multiple senders, due to the randomness of threading, the messages in aggregate may be interleaved. The following diagram should clear things up.

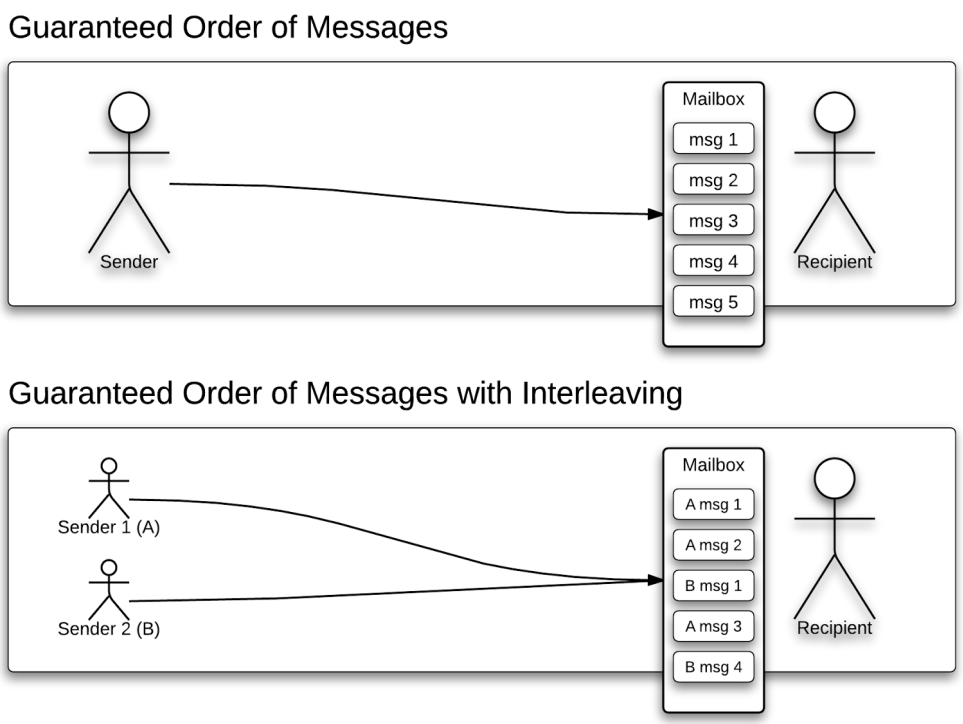

Figure 3.4 Guaranteed Order of Messages

Don't be confused by this, the recipient will always receive messages in order from a given sender, its just that when there are multiple senders, as the figure shows, that interleaving can occur.

3.4.5 Behavior and The Receive Loop

Akka also provides the semantics for actor behavior. Each time an actor receives a message, it's processed against its behavior in a method known as the `receive` loop.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

Next, we we'll explore the `receive` loop in code by giving you just a brief introduction into behavior management with actors, modelling a `Greetings` actor that supports two types of messages, `Hello` and `Goodbye` in both Java and Scala.

HELLO IN JAVA

Let's start with the `Hello` message first.

Listing 3.1 `ImmutableHello` Message in Java

```
public final class ImmutableHello {     (1) , (2)

  public ImmutableHello(String name) { (3)
    this.name = name;
  }

  public String getName() {             (4)
    return name;
  }
}
```

(1) **The class is marked final so it cannot be extended.**
(1) **Prefix the class with Immutable, this is convention, not required but good practice.**
(2) **Class variables are final, making them immutable.**
(3) **Class constructor that requires variable.**
(4) **Getters only**

Let's walk through the details of the Java version.

- The first thing you will notice is the class is marked `final`. This is so no one extends it and potentially violates our immutability requirement.
- Next we prefixed the class with the word `Immutable`. The immutable prefix is more a convention than anything else. It has no impact on the class other than making it easily identifiable as a class we should not alter.
- Next we make all class variables `final`. In Java, the keyword final enforces that once a variable is set, it cannot change for the life of the variable.
- Then we add a constructor that requires all class variables enforcing its creation as intended. Also, its the only way to set the value of variables marked final.
- Finally, we a getter without a setter. As before, this is convention. In reality, we could a setter, but they are of no value as you cannot set variables marked `final`.

There is another way to model a Java class to enforce immutability, but some regard it with disdain as it does not follow accepted practices for Java class design.

Listing 3.2 `ImmutableHello` Message in Java (Alternative)

```
public final class ImmutableHello {     (1) , (2)
  public final String name;             (3)

  public ImmutableHello(String name) { (4)
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

```
    this.name = name;
  }
}
```

1. The class is marked `final` so it cannot be extended.
2. Prefix the class with `Immutable`, this is convention, not required but good practice.
3. Class variables are `final`, making them immutable.
4. Class constructor that require variable.

The primary difference with the alternative is the lack of getters and the variables being public. This also changes how we access the classes data. In Listing 3.1 we must call a getter whereas with Listing 3.2, we use the variable directly. In the end, the effect is the same, both are immutable. Our goodbye message `ImmutableGoodbye` looks exactly the same with the only difference being the name of the class and the constructor so we won't model that for brevity.

HELLO IN SCALA

Next we'll look at the Scala version.

Listing 3.3 `Hello` Message in Scala

```
final case class Hello(name: String) (1)
```

1. Class name Hello. Don't need Immutable prefix as Scala case classes are immutable by design.

Pretty different than Java huh? If you are new to Scala, this can look surprising. A single line of code? That's correct, and the power of a Scala case class. So what is a case class and how is it immutable? A *case class* is a regular class that applies some syntactic sugar (magic) when compiled that enforces immutability. Using our example from Listing 3.3, let's walk through the steps the compiler applies:

- The class becomes final and implements scala.ScalaObject and scala.Serializable.
- The constructor arguments export as public final variables.
- The compiler adds generated toString, equals and hashCode methods.
- The compiler adds an apply method that allows creation without the new keyword.
- Other magic, that's not important for our discussion.

If you're interested in seeing what a compiled Scala case class looks like under the covers, the JVM provides a nice utility called `javap`. Simply run `javap <the name of your case class.scala>` and output will print to the console. Again, we will not model the `Goodbye` case class for brevity.

Now let's take a look at what our `GeetingsActor` looks like first modelled in Java, then in Scala.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

GREETINGSACTOR IN JAVA

This next listing sets up the GreetingsActor in Java.

Listing 3.4 `GreetingActor` with Simple `receive` Loop in Java

```
import akka.actor.UntypedActor;                    (1)
import akka.event.Logging;                         (2)
import akka.event.LoggingAdapter;                  (2)

public class Greeting extends UntypedActor {       (1)
  LoggingAdapter log = Logging.getLogger(getContext().system(), this); (2)

  public void onReceive(Object message) throws Exception {             (3)
    if (message instanceof ImmutableHello) {
      log.info("Received hello: {}", message);
      ImmutableHello ih = new ImmutableHello("Greetings Hello");
      getSender().tell(ih, getSelf());   (4) , (5) , (6)
    } else if (message instanceof ImmutableGoodbye) {
      log.info("Received goodbye: {}", message);
      ImmutableGoodbye ig = new ImmutableGoodbye("Greetings Goodbye");
      getSender().tell(ig, getSelf());   (4) , (5) , (6)
    } else
      unhandled(message);                          (7)
    }
  }
}
```

(1) Actors are implemented in Java by extending `UntypedActor`.
(2) Logging is implemented with `Logging` and `LoggingAdapter`.
(3) Actors also require the `onReceive` method which takes the `message` as a parameter.
(4) `getSender` gets a reference to the actor who sent the `message`.
(5) `tell` uses "fire-and-forget" semantics to send the `message` asynchronously.
(6) `getSelf` references the actor that is receiving the `message`.
(7) `unhandled` publishes a new `akka.actor.UnhandledMessage` to the actor's system event stream which can be configured to convert them into `Debug` messages.

GREETINGSACTOR IN SCALA

Now, the GreetingsActor in Scala.

Listing 3.5 `GreetingActor` with Simple `receive` Loop in Scala

```
import akka.actor.Actor                            (1)
import akka.actor.ActorLogging                     (2)

class Greetings extends Actor with ActorLogging { (1) , (2)

  def receive = {                                  (3)
    case msg: Hello =>
      log.info(s"Received hello: $msg")
      sender() ! Hello("Greetings Hello")          (4) , (5)
    case msg: Goodbye =>
      log.info(s"Received goodbye: $msg")
      sender() ! Goodbye("Greetings Goodbye")      (4) , (5)
    case ukm:       _ => log.info(s"Received unknown message: $ukm") (6)
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

```
  }
}
```

1. Actors are implemented in Scala by extending `Actor`.
2. Logging is implemented by mixing-in the `ActorLogging` trait.
3. Actors also require the `receive` method which defines which messages the actor handles.
4. `sender` gets a reference to the actor who sent the `message`.
5. `!`(the bang operator) uses "fire-and-forget" semantics to send the `message` asynchronously.
6. Scala _(underscore notation) is used to matches and logs any unknown messages.

COMPARING JAVA AND SCALA FOR THE GREETINGS ACTOR

Let's walk through the difference between the Java and Scala versions of our `Greetings` actor.

- The first difference is that Scala actors use `akka.actor.Actor` instead of `akka.actor.UntypedActor`. This is due to the fact its quite difficult to implement a Scala `PartialFunction` in Java 7 and below. With the advent of Java 8, this will most likely change, but for now as per the Akka documentation, this is the way it's done.
- Next you will notice that there is no need to instantiate a `Logger` as it's included by mixing-in the `ActorLogging` trait.
- The `receive` semantics are different as well, again because Scala can handle `PartialFunction`. The `receive` method matches against a series of case statements as opposed to the `if` statement in the Java version.
- Finally, if we received a handled message, we respond to the `sender` using the `bang` operator. The `sender` method is pretty much the same as `getSender` in Java. Scala also supports operator notation, thus the `!` instead of `.tell`. Any unhandled messages are matched using Scala `underscore` notation and subsequently logged.

In both of these examples, there is only one receive loop that captures all behavior that essentially is respond to a message. Within that loop, we respond with a `Hello` or `Goodbye` depending on the message we receive. But is this really behavior? At some level one could argue that it is, but ultimately in both cases the action is the same, a response. We would argue that behavior is role-based, and the actions within a role are simply the execution of that behavior.

GREETINGSACTOR CHANGES

With this idea in mind, let's change our `GreetingActor`'s role from always responding to, to one that is as follows:

- The `GreetingActor` will start in a role where it only accepts `Hello` messages.
- When the `GreetingActor` receives a `Hello` message, it will change its role to only accept `Goodbye`.
- Then when the `GreetingActor` receives a `Goodbye` message, it will change to its initial role of only accepting `Hello`.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

To do this, we will add additional `receive` loops, and swap between them with the `become` operation. The semantics behind swapping are a little complicated, but Akka uses a `Stack` to keep track of the hotswapped code as it's pushed and popped. Let's see what this looks like first in Java then Scala.

Listing 3.6 GreetingActor with become Operation in Java

```
import akka.actor.UntypedActor;
import akka.event.Logging;
import akka.event.LoggingAdapter;
import akka.japi.Procedure;                                    (1)

public class Greeting extends UntypedActor {
  LoggingAdapter log = Logging.getLogger(getContext().system(), this);

  Procedure<Object> hello = new Procedure<Object>() {          (2)
    @Override
    public void apply(Object message) {
      if (message instanceof ImmutableHello) {
        log.info("Received hello: {}", message);
        ImmutableHello ih = new ImmutableHello("Greetings Hello");
        getSender().tell(ih, getSelf());
        getContext().become(goodbye);                          (3)
      } else {
        unhandled(message);
      }
    }
  };

  Procedure<Object> goodbye = new Procedure<Object>() {        (4)
    @Override
    public void apply(Object message) {
      if (message instanceof ImmutableGoodbye) {
        log.info("Received goodbye: {}", message);
        ImmutableGoodbye ig = new ImmutableGoodbye("Greetings Goodbye");
        getSender().tell(ig, getSelf());
        getContext().become(hello);                            (5)
      } else {
        unhandled(message);
      }
    }
  };

  public void onReceive(Object message) {                      (6)
    getContext().become(hello);
  }
}
```

(1) **`Procedure` trait. Similar to a function but does not return a value.**
(2) **The `hello` procedure for the receive loop.**
(3) **`context.become(goodbye)` replaces the top of the stack, `hello` with `goodbye`.**
(4) **the `goodbye` procedure for the receive loop.**
(5) **`context.become(hello)` replaces the new top of the stack, `goodbye` with `hello`.**
(6) **Sets the default behavior for `onReceive()` to `hello`.**

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

Listing 3.7 `GreetingActor` with `become` Operation in Scala

```
import akka.actor.Actor
import akka.actor.Props
import akka.event.Logging

class GrettingsActor extends Actor {
  val log = Logging(context.system, this)

  override def receive = hello  (1)

  def hello = {                 (2)
    case msg: Hello =>
      log.info(s"Received hello: $msg")
      sender() ! Hello("Greetings Hello")
      context.become(goodbye)   (3)
    case _     => log.info("received unknown message")
  }

  def goodbye = {               (4)
    case msg: Goodbye =>
      log.info(s"Received goodbye: $msg")
      sender() ! Hello("Greetings Goodbye")
      context.become(hello)     (5)
    case _     => log.info("received unknown message")
  }
}
```

(1) Overrides the default receive loop provided by the Actor trait with the `hello`.
(2) The `hello` receive loop.
(3) `context.become(goodbye)` replaces the top of the stack, `hello` with `goodbye`.
(4) the `goodbye` receive loop.
(5) `context.become(hello)` replaces the new top of the stack, `goodbye` with `hello`.

Our `GreetingActor` is just a brief introduction into behavior management with actors. As you work through the book, we will show you more examples of how you can leverage this powerful feature of Akka.

3.4.6 Supervision

One of the most powerful features that the Akka provides for actors is the notion of supervision. We will dig into this in detail in the next section on The Actor System, as it's better understood in that context, but as a quick intro we will mention the basics here.

Akka allows any actor to create child actors for the purposes of delegating sub-tasks. In this context, the spawning actor becomes known as a supervisor and has authority over the children. Supervision is a very powerful metaphor that provides the mechanics taking Akka beyond fault tolerance to being resilient.

Resilience vs. Fault Tolerance

Of the four traits in the Reactive Manifesto, resilience is the most often misunderstood, usually being defined as fault tolerance. While resilience and fault tolerance have a lot in common, they are not one and the same.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

In computer science, fault tolerance is defined as an aspect of the system that allows continued operation in the event of failure of some of its components. So far so good; resilience covers this too, but now for the distinction.

Fault tolerance contains an implied caveat: in a fault tolerant system, the degradation of quality is proportional to the severity of the failure; meaning the larger the failure, the worse the system behaves. We often see this in the dreaded cascading failure scenario, where one server fails and the load balancer shifts traffic to another. Now the new server is not only handling its original load, but the new load as well, and on it goes, until the entire server bank craters.

Resilience, on the other hand, means reacting to failure by springing back into shape. So, rather than passing the buck to the next guy in line, a resilient system self-heals. Self-healing is achieved by repairing the failed component or spinning up a new one as a replacement.

We've seen a great many features that Akka actors bring to the table in support of the reactive paradigm. Next, we'll take a look at how Akka manages all this through the actor system.

3.5 The Actor System

In Akka, the actor system is the setting in which actors exist. It is a heavyweight construct that manages concurrency, actor lifecycles, and execution context to name a few, and forms a boundary that is both spatial and temporal. To get an overview of how an actor system works, we will review four key concepts:

- hierarchical structure that contains the bubble walker and top-level actors,
- supervision,
- actor paths, and
- the actor lifecycle.

Let's start with the hierarchical structure.

3.5.1 Hierarchical Structure

The structure of an actor system is hierarchical in nature, much like that of a modern corporation. In a corporation, there usually is a single person at the top: the Chief Executive Officer (CEO), responsible for the success or failure of the company. To manage this responsibility, the CEO will devise an overall strategy that covers both scenarios.

Underneath the CEO, there is a group executive officers who are tasked with implementing the strategy per their area of responsibility, and on down the chain it goes. This entire hierarchical structure whether corporation or actor system has one underlying means to carry out the strategy: splitting up and delegating tasks as small chunks.

Let's take a look at the executive team of an actor system in the following diagram.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

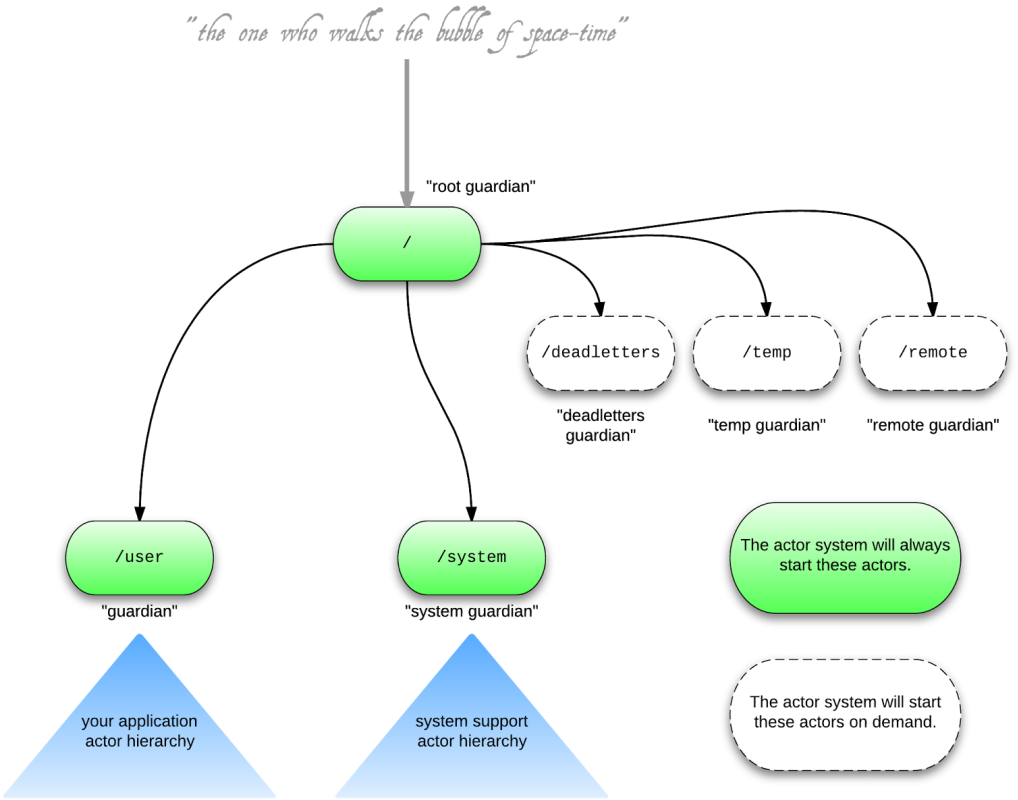

Figure 3.5 The Akka Actor System Hierarchy

BUBBLE WALKER

At the highest level of the hierarchy, there resides a unique object called a *"bubble walker"* who affectionately is *"the one who walks the bubbles of space-time."* This might sound a little crazy, but it actually makes a lot of sense. Every real actor within the actor system has to have a supervisor. We will talk about supervision in a moment, but for now follow along. The supervisor is responsible for managing exceptions thrown by the actors it supervises.

ROOT GUARDIAN

At the top of the actor stack is root guardian who, by the way, must have a supervisor as well. The problem: he is at the top, so who is his supervisor? That is the role of the *"bubble walker"* The *"bubble walker"* is a synthetic `ActorRef` which monitors and stops the root guardian at the first sign of trouble. As soon as the root guardian is terminated, the *"bubble walker"* then sets the systems `isTerminated` status to `true` and the actor system is officially shut down.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

TOP-LEVEL ACTORS

In addition to the *"bubble walker"* Akka provides a set of what a called *"top-level"* actors, as seen in Figure 3.5, that a responsible for different areas of the system. Let's take a look at what each one of these actors does.

- `/`: *The Root Guardian* - The root guardian is the CEO of the actor system and supervises all the "top-level" actors which are list below. Whenever one of the other "top-level" actors throws an Exception, the root guardian will terminate them. All other throwables escalate up to the "bubble-walker".
- `/user`: *The Guardian* - The guardian actor is the parent of all user-created actors who are created using system.actorOf(). When the guardian escalates a failure, the root guardian will respond by terminating the guardian. As a result, the guardian will terminate all user-created actors which effectively shuts down the actor system.
- `/system`: *The System Guardian* - The system guardian is a special actor that provides an orderly shutdown sequence where logging remains while user-created actors terminate. The system guardian monitors the guardian and when complete, initiates its own shutdown.
- `/deadletters`: *The Deadletters Guardian* - The deadletter guardian is a special guardian where all messages sent to stopped or non-existent actors go. It uses a best-effort basis to capture the orphaned messages as sometimes they can be lost within the local JVM.
- `/temp`: *The Temp Guardian* - The temp guardian is a "top-level" actor for short lived system-created actors like those that use `.ask`.
- `/remote`: *The Remote Guardian* - The remote guardian is for actors whose supervisors reside on a remote system.

This hierarchical structure is one of the key concepts of Akka's resilience. By having a structure that is supervised, Akka can embrace failure by delegating the appropriate action.

Now that we have a good sense of the hierarchy of Akka's actor system let's take a look at what we mean by supervision.

3.5.2 Supervision

You probably have a good idea from the hierarchy discussion of what supervision is in terms of Akka. As we mentioned above, all real actors must have a supervisor. This supervisory relationship establishes a dependency between actors, where the supervisor:

- Delegates tasks to its subordinate, and responds to their failure.
- Whenever a subordinate detects a failure, it suspends itself, all of its subordinates and reports back to its supervisor.
- The supervisor then receives the failure and based on their strategy and applies a directive.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

Let's first look at the strategies available to a supervisor.

- `OneForOneStrategy` (default): This strategy operates just as its name. The supervisor will execute one of four directives (listed below) against the subordinate, and inturn all of the subordinates children.
- `AllForOneStrategy`: When using this strategy, the supervisor will execute one of the four directive not only against the subordinate that faulted, but also against all subordinates the supervisor manages. Hence the name `AllForOneStrategy`.

The default strategy for all supervisors it `OneForOneStrategy`. You can create your own supervisor strategy by extending the `akka.actor.SupervisorStrategy` class. We will show you how strategies are implement a little later in the chapter, for now let's look at the directives the supervisor can take.

1. `Resume` the subordinate, keeping its accumulated internal state.
2. `Restart` the subordinate, clearing out its accumulated internal state.
3. `Stop` the subordinate permanently.
4. `Escalate` the failure, thereby failing itself.

The supervision strategy and directives semantics provide a powerful mechanism for the transparent handling of faults that enables the self-healing of resilience. We should note that communication used between supervisors and their subordinates come in the form of special messages managed in mailboxes separate from the actors. The implication is that the ordering of supervision related messages is not relative to normal ones.

3.5.3 Actor Paths

So far we've seen what an actor is, the hierarchical system they exist in, and how to manage actors through supervision. Now let's take a look at how we access actors within the hierarchy.

In any good hierarchy, there exists a unique trail of names that one can recursively follow. From the root guardian on down through the hierarchy, one can reach any actor in the system known as the *actor path*. The actor path consists of a root that is the actor system, followed by a concatenated sequence of elements, delineated with a forward slash. You can liken it to a URL.

Let's see what they look like in this next code listing.

Listing 3.7 Local and Remote Actor Paths

```
"akka://bookstore/user/order/worker-1"                              // local
"akka.tcp://bookstore@host.bookstore.com:5154/user/order/worker-2" // remote
```

- The first part of the path establishes the system being accessed: "`akka`", and in the case of the remote path, the transport as well: "`akka.tcp`". Akka supports both TCP and UDP for remote transports.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

- The next part establishes the server. In the local case its "bookstore" and the remote is "bookstore@host.bookstore.com" on port "5154".
- Lastly, we enter the actual actor system in both cases through the root guardian, the "/" before "user" down to the actual actor, "worker-1" or in the remote case "worker-2".

The result of accessing an actor by its path is an ActorRef, and this is obtained in one two ways: by looking them up or creating them. To lookup an actor, Akka supports both relative and absolute paths through the use of ActorSystem.actorSelection and ActorContext.actorSelection. In addition to obtaining and ActorRef, you can use actorSelection to send a message to an actor directly.

The following listing is an example of using actorSelection to send an actor a message using both a relative and absolute path.

Listing 3.8 Using actorSelection with Relative and Absolute Paths

```
context.actorSelection("../worker-1) ! msg          // relative
context.actorSelection("/user/order/worker-1) ! msg  // absolute
```

To create an actor, you use ActorSystem.actorOf or ActorContext.actorOf. ActorSystem.actorOf is generally used when starting the system, while the latter is used from within actors already created. Unlike actorSelection, actorOf requires Props which is an immutable class for specifying options for the creation of an actor.

Next is an example of using ActorContex.actorOf with Props for creation.

Listing 3.9 Using actorOf with Props to Create an Actor

```
class Manager extends Actor {
  context.actorOf(Props[Worker], “worker-1”)
  // other code …
}
```

In Listing 3.9 we are setting the type of Props to be our actor class, in this case Worker. In the actorOf method, we first pass in the Props object, followed by the name of our actor “worker-1”.

While this approach is certainly acceptable, its best practice to establish a Props factory within your actor as follows.

Listing 3.10 Using actorOf with Actor Props Factory

```
object Worker {
  def props = Props[Worker]
}

class Worker extends Actor {
  def receive = {
    case order:Order => // do something ...
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

```
    // other code …
  }
}

class Manager extends Actor {
  context.actorOf(Worker.props, “worker-1”)
  // other code …
}
```

3.5.4 Actor Lifecycle

An actor has essentially three phases which it runs through: starting, restarting and stopping, which tie back into the supervisor directives we discussed above. Let’s talk about starting first.

STARTING

When the actor system starts an actor with `actorOf` the following sequence of events occur:

- `actorOf` is called
- the actor path is reserved
- a random UID is assigned for the actor instance
- the actor instance is created
- the `preStart()` method is called on the actor instance
- the actor instance is started

This sequence of event can occur in one of two conditions. When an actor is started, or when an actor is restarted by the supervisor. Before we talk about restart, let's first talk about stop as stop is included in restart.

STOPPING

When an actor is stopped the following sequence of events occur:

- the actor instance is asked to stop with `Stop`, `context.stop()` or `PosionPill`
- the `postStop()` method is called on the actor instance
- the actor instance is terminated
- the actor path is now usable again

RESTARTING

Now let’s talk restart because it's an interesting combination of the two. When a supervisor restarts an actor the following sequence of events occur:

- the `postRestart()` method is called on the old actor instance
- the old actor instance is asked to stop with `Stop`
- the `postStop()` method is called on the old actor instance
- the actor path of the old instance is reserved for the new instance
- the UID of the old instance is reserved for the new instance
- the old actor instance is terminated

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

- the new instance is created
- the `preStart()` method is called on the new instance
- the new actor instance is started

When the actor system executes the restart directive, it actually replaces the faulty actor with a new one. This may sound a little odd, but that's the way it works. The important thing to note is during restart, the `postRestart()` method is called on the old instance rather than the new.

3.5.5 Microkernel Container

Software as a tool should probably hold the title of "most used and versatile invention ever created." There are tens of millions of software applications running in the world today, solving a wide range problems from the mundane to the extremely complex.

Software in many ways can be likened to the human body. The human body is an incredible machine that can be trained to do just about anything. But there is one thing the human body must have in order to operate: an oxygenated environment. In software lingo, we call this the runtime. Every application ever written requires one and without it is nothing more than a bunch of bits. Fortunately, when it comes to Akka, the runtime is included. The runtime in Akka, the microkernel, is designed and optimized to offer a bundling mechanism that allows single payload distribution atop the JVM without the need for an application container or startup script.

PUTTING OUR TOOLKIT TO WORK

We've spent a good deal of time establishing the rationale behind using Akka as a reactive toolkit; now it's time to put it to use and have some coding fun. Starting in the next chapter, we'll establish an analogy that that will help us reason through the process of building a reactive application. We will start small and then throughout the book expand our analogy, while we in parallel apply the techniques we discover in code.

3.6 Summary

To close out this chapter, let's review what we've learned.

- The history of Akka and how it evolves to keep pace with the ever growing demands of computing.
- Akka Terminology
  - Concurrency and parallelism and their differences.
  - Asynchronous and synchronous semantics.
  - How do deal with asynchronous communication through blocking, callbacks, futures and message passing.
  - The challenge of contention in a concurrent environment.
  - A share nothing mentality through isolation and immutability.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

- The actor model
  - State management within an actor.
  - What an actor reference is.
  - How actors asynchronously message.
  - What the mailbox is.
  - Actor behavior with the receive loop.
  - Actor supervision
  - The difference between resilience and fault tolerance.
- The actor system
  - The hierarchical structure of the actor system.
  - Supervision strategies and directives.
  - What actor paths are.
  - How the actor lifecycle operates.
  - What the Akka microkernel is.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

4

Akka Basic Toolkit

This chapter covers:

- Analogical reasoning
- Establishing an analogy for code
- Implementing message driven architecture
- Implementing elasticity
- Implementing resilience

4.1 Bookstore Analogy

We believe one of the best ways to learn is through the process of analogical reasoning, or comparing the known to the unknown. An analogy is like a handle on a heavy bucket. While you may still be able to pick the bucket up without one, your chore will be more difficult. The use of analogies in graduate school is quite common, especially in the disciplines of science and engineering. In these disciplines, analogies are referred to as a *succedaneum*, which means to use something in place of another. The poet William Wordsworth captured it best[12]:

> *Science appears as what in truth she is,*
> *Not as our glory and our absolute boast,*
> *But as a succedaneum, and a prop*
> *To our infirmity.*

[12] The Prelude: Book 2: School-time (lines 212-15)

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

To begin our learning by succedaneum, we will use a simple analogy to reason about and model an Akka-based reactive application starting with what we have learned so far. Then as you work your way through the book, the complexity of the analogy will increase step-by-step. With each increase in complexity, you will learn a new reactive concept that will be used to evolve the application layer-upon-layer, until the whole idea is complete. As a result of this learning style, you'll discover how to craft your design, such that the theory translates into succinct maintainable code that is enjoyable to write. Let's get started!

BUILDING A BOOKSTORE

Imagine a scenario where an ambitious librarian with a specialty in books of antiquity decides to start a question and answer service. Being an expert on antique books, our librarian quickly obtains two customers who are delighted to have found a specialist. To facilitate communication with his clients, the librarian decides to use a whiteboard in the back room of a local coffeehouse. He sets a schedule where each customer on alternating days will post a question by noon that he will answer later that day. Then the customer who posted the question will return by 8 PM to retrieve their response.

Not long after starting this process, the librarian encountered a problem. His regularly scheduled customer came in and posted their question at 9am as usual. Then unbeknownst to the librarian, the second customer being confused about the date came in and overwrote that question and with their own. When the librarian came in that afternoon, not realizing what had happened he posted his response. Later that evening, the first customer returned to retrieve the librarian's answer, which they would later learn was incorrect, and then this customer promptly filed a complaint. Realizing what had happened, the librarian knew he had to come up with a solution.

SOLUTIONS

The first solution the librarian considered was to hire a guard to lock the door after a customer posted their question. Locking the door would prevent another customer from getting access to the board before the first question was answered. While this solution would most likely work, it was quickly dismissed, because it required additional resources and was costly. Additionally, the librarian realized that this would more tightly couple him to the use of the back room, which would prevent his ability to scale in the future.

After much thought, the librarian realized he did not need the whiteboard at all. A much better approach would be to use the services of the post office. By relying on a written question sent through the mail, he was assured the request would be the correct one for a given customer as Federal law prohibits the tampering of mail in transit. In essence, the letter is immutable. Furthermore, neither he nor his customers would be bound by the use of the coffeehouse, thus establishing a loosely coupled relationship that would help him grow.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

To get his new business started our librarian files the appropriate paperwork to do business in his location starting a new company called Rare Books, LLC. In addition, he hires one worker, another librarian who will handle the day to day processing of customer requests whom he will manage. By freeing himself from the daily grind, not only will he be able to meet his current customer's needs, but he will also have the time to focus on growing his business.

Now that we have baselined the analogy, let's break it down in down in Akka terms in a simple table and then start by coding the messages used by our librarian and customer.

Table 3.1 Analogy Construct to Akka Translation Table

Analogy Construct	Akka Translation
Post Office	Messaging framework
Customer question sent by mail	Immutable case class
Librarian answer sent by mail	Immutable case class
RareBooks	Top level actor representing the owner
Librarian	Librarian actor who is a child of RareBooks
Customer	Top level actor representing the customer

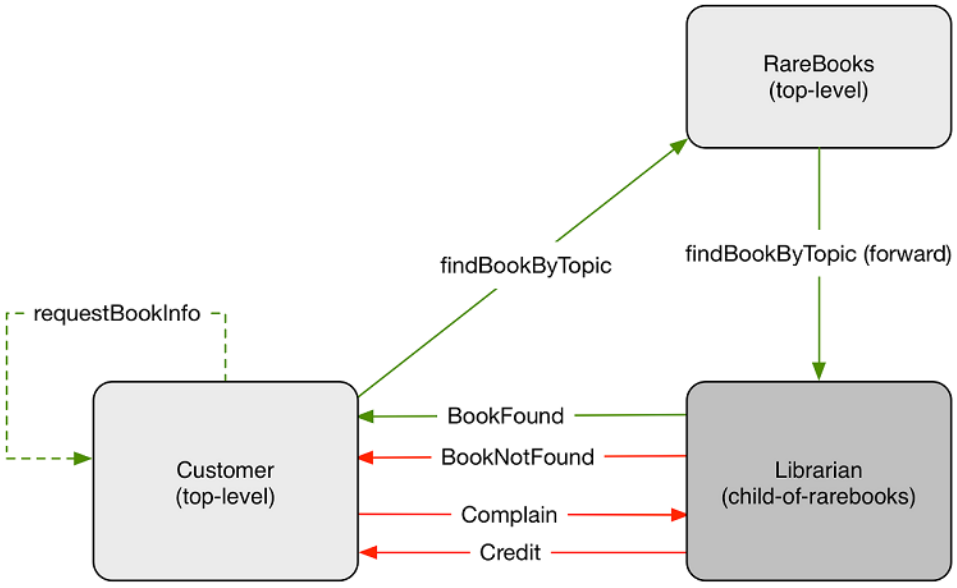

Figure 4.1 The RareBooks Actor System

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

4.2 Message Driven

4.2.1 Immutable Messages

One of the challenges our librarian faced was when he responded to a question where the second client's question overwrote the first. In this case, because of scheduling confusion both customers contextually (during the same day), tried to ask a question at the same time, which resulted in a non-deterministic (one not expected based on the input) outcome. In Akka, this translates into a concurrency fault caused by a race condition.

The librarian first considered hiring a guard to lock the door but quickly dismissed the notion. The guard could be equated to the synchronization and locking strategies we mentioned above in the concurrency discussion. While this approach would work, through the analogy we see that it enforces tight coupling that is something we want to avoid in a reactive system. Additionally, it has the potential to introduce other concurrency problems such as deadlocks, livelocks, and starvation mentioned above as well. In the end, the librarian chose a reactive approach, by establishing a base of operation, Rare Books LLC, hiring another librarian, and using a message-driven service that implements immutable messaging, the post office.

CODING THE MESSAGE PROTOCOL

We will start modeling our analogy with the following code listing of the types of questions a customer will ask. These "questions" are a form of communication that we will model as an immutable message. For brevity sake, we will model two message that we expect Rare Books to answer. From a programming point of view, you can think of the messages as a form of API for a given actor. In general, its is considered a best practice to define the messages an actor receives in either a protocol object (as seen below) or the actors companion object.

What is a Protocol Object?

In computer science, a protocol is defined as the rules that allow two entities to communicate with one another. These rules or standards are the syntaxes and semantics of acceptable communication. Following this idea, it is quite common to define a "protocol" object that contains the messages that are used within the actor system.

Listing 4.1 RareBooksProtocol.scala

```
package com.rarebooks.library ❶

import scala.compat.Platform  ❷

/**
 * This file contains the messages for the rare books info service.
 */
object RareBooksProtocol {    ❸
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

```
sealed trait Topic
case object Africa extends Topic
case object Asia extends Topic
case object Gilgamesh extends Topic
... other topics

/**
 * Card trait for book cards.
 */
sealed trait Card {          ❹
  def title: String
  def description: String
  def topic: Set[Topic]
}

/**
 * Book card class.
 *
 * @param isbn the book isbn
 * @param author the book author
 * @param title the book title
 * @param description the book description
 * @param dateOfOrigin the book date of origin
 * @param topic set of associated tags for the book
 * @param publisher the book publisher
 * @param language the language the book is in
 * @param pages the number of pages in the book
 */
final case class BookCard(     ❺
    isbn: String,
    author: String,
    title: String,
    description: String,
    dateOfOrigin: String,
    topic: Set[Topic],
    publisher: String,
    language: String,
    pages: Int)
  extends Card

/** trait for all messages. */
trait Msg {     ❻
  def dateInMillis: Long
}

... other message implementations

/**
 * Find book by isbn message.
 *
 * @param isbn isbn to search for
 * @param dateInMillis date message was created
 */
final case class FindBookByIsbn(     ❼
    isbn: String,
    dateInMillis: Long = Platform.currentTime) extends Msg {
  require(isbn.nonEmpty, "Isbn required.")
}
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

```
  /**
   * Find book by topic.
   *
   * @param topic set of topics to search for
   * @param dateInMillis date message was created
   */
  final case class FindBookByTopic(     ❽
      topic: Set[Topic],
      dateInMillis: Long = Platform.currentTime) extends Msg {
    require(topic.nonEmpty, "Topic required.")
  }

  ... other message implementations
}
```

❶ The package structure for this object
❷ Provides the current system time
❸ Protocol object establishing base messages for all actors
❹ Trait for book card information
❺ Case class for book card information
❻ Common trait for all messages
❼ Case class for requesting books by ISBN
❽ Case class for requesting books by Topic

In Listing 4.1 we show a partial listing of the `RareBooksProtocol` object which is a singleton, and contains the majority of messages and abstractions that our actors will use. One of those abstractions is the marker `trait Msg` that is used for matching. Marker traits are an abstraction used to generalize or group messages or structures under a certain type. We will explore this generalization more with the notion of commands and events in Chapter 6. Another thing to note about our messages thus far is they both have a `dateInMillis` of type `Long` parameter. For the purpose of our domain, `dateInMillis` uses `Platform.currentTime` for a creation timestamp.

With this immutable message structure, we've taken a small but important step toward a reactive design. By adopting the share-nothing mentality, in the sense that our messaging structures are immutable, we avoid the many pitfalls of concurrent programming, while moving toward a loosely coupled message-driven workflow.

Next, let's look at our message-passing implementation by modeling our actors.

4.2.2 Asynchronous Concurrency with Actors

The first thing we need to do is see which parts of our analogy would suit the role of an actor. Recall that *actors* are lightweight objects that encapsulate state, behavior, and solely communicate through asynchronous message passing. In looking at our participants, we see three potential candidates:

- The librarian
- the customers
- the post office

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

In the context of our analogy, let's compare each member to the features of an actor in a table checklist and validate our assumptions.

Table 3.2 Actor Checklist

Participant (Actor)	State?	Behavior?	Async Messaging?	
RareBooks (Owner)	Manages workflow and encapsulates state as system owner.	Exhibits behavior by managing workflow.	Asynchronously message direct to librarian	√
Librarian	Encapsulates state as the single source of truth for the answer.	Exhibits behavior by responding to a customer's question.	Asynchronously messages by using the post office.	√
Customers	Encapsulates state as the single source of truth for their question.	Exhibits behavior by requesting an answer.	Asynchronously messages by using the post office.	√
Post Office	Encapsulates state as the single source of truth for message delivery.	Exhibits behavior by delivering the mail.	Provides the delivery mechanism. Does not directly message participants	X

Table 3.2 shows us that the three fitting candidates for actor modeling are the Rare Books owner, the librarian, and the customers as all three represent distinct identity, behavior, and potential state management. While we could model the post office as an actor too, in the context of our analogy, the role of message delivery mechanism is more suitable as that is the primary responsibility of the postal service.

CODING THE CATALOG

Before we model our actors, we need to design the content they will discuss. In our case, the content is information related to rare books which in library terms is known as the *card catalog*. For the purpose of our analogy at this point, we will start with an in-memory representation of the *card catalog* in the form of the `Map` implemented in a `singleton` object `Catalog` in `Catalog.scala`. As we progress through the chapters, we will move this content to a `NoSQL` persistent store, but for now an in-memory singleton will work just fine.

Our `Map` will consist of a key of type `String`, the ISBN which points to a `BookCard`, an immutable class we saw earlier in `RareBooksProtocol` that contains pertinent information relating to a book. Following is a partial listing of what the Catalog looks like:

Listing 4.2 The Catalog

```
package com.rarebooks.library

object Catalog {     ❶
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

```
  import RareBooksProtocol._     ❷

  val phaedrus = BookCard(       ❸
    "0872202208",
    "Plato",
    "Phaedrus",
    "Plato's enigmatic text that treats a range of important ... issues.",
    "370 BC",
    Set(Greece, Philosophy),
    "Hackett Publishing Company, Inc.",
    "English",
    144)

  val theEpicOfGilgamesh = BookCard(     ❹
    "0141026286",
    "unknown",
    "The Epic of Gilgamesh",
    "A hero is created by the gods to challenge the arrogant King Gilgamesh.",
    "2700 BC",
    Set(Gilgamesh, Persia, Royalty),
    "Penguin Classics",
    "English",
    80)

  ... other BookCard instances

  val books: Map[String, BookCard] = Map(     ❺
    theEpicOfGilgamesh.isbn -> theEpicOfGilgamesh,
    phaedrus.isbn -> phaedrus,
    theHistories.isbn -> theHistories)

  ... other code used to fetch BookCard items from the map
}
```

❶ **The singleton Catalog**
❷ **Import for RareBooksProtocol.BookCard**
❸ **Instance of phaedrus BookCard**
❹ **Instance of theEpicOfGilgamesh BookCard**
❺ **The state map representing the BookCards in the catalog**

CODING RAREBOOKS (THE OWNER)

Now that we've outlined how messages should look and identified our state model for `BookCard` items, the catalog, let's see how our `RareBooks` actor will look. In Chapter 2, we introduced the concept of a top-level actor. In Akka terms, this means any actor created by using `system.ActorOf()`, and by designation are children of the *User Guardian*. While this may sound ominous, there is nothing particularly unusual. It merely implies the *User Guardian* is the parent and supplies the default supervision strategy. We will talk in more detail about supervision later in the chapter as it relates directly to resilience. Lets start with an outline of what `RareBooks` will look like.

`RareBooks.scala` - This is the source file that contains `RareBooks` related code.

- `object RareBooks` - As mentioned above, this is the singleton companion object. It

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

contains local imports, local message definitions, a properties factory and other local functions for processing.
    - `case object Close` - messages `RareBooks` is closed
    - `case object Open` - messages `RareBooks` is open
    - `case object Report` - messages `RareBooks` daily report
    - `def props: Props = Props(new RareBooks)` - properties factory for creation
- class RareBooks
    - `extends Actor with ActorLogging with Stash`
    - `import context.dispatcher` - used for simulation
    - `import RareBooks._` - import local companion object
    - `import RareBooksProtocol._`
    - `private val openDuration: FiniteDuration` - used for simulation
    - `private val closeDuration: FiniteDuration` - used for simulation
    - `private val findBookDuration: FiniteDuration` - used for simulation
    - `private val libararian` - local state for librarian child actor
    - `var requestsToday: Int` - threadsafe local mutable state
    - `var totalRequests: Int` - threadsafe local mutable state
    - `override def receive: Receive = ready` - set default receive behavior
    - `private def open: Receive = {` - behavior when open
    - `private def close: Receive = {` - behavior when closed
    - `protected def createLibrarian()` - factory for creating Librarian child actor

Listing 4.3 The RareBooks. scala File

```
package com.rarebooks.library          (1)

import akka.actor.{ Actor, ActorRef, ActorLogging, Props, Stash }    (2)
import scala.concurrent.duration...    (2)
```

(1) **Package containing the RareBooks.scala file**
(2) **File level imports shared by the companion object and actor**

The `RareBooks.scala` file is the container for for both the companion object and actor.

Listing 4.4 The RareBooks Companion Object

```
object RareBooks {      (1)

  case object Close     (2)
  case object Open      (3)
  case object Report    (4)

  def props: Props =    (5)
    Props(new RareBooks)
}
```

(1) **Singleton companion object for messages, properties factory and local functions**
(2) **private Close immutable message**

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

❸ private Open immutable message
❹ private Report immutable message
❺ Properties factory for actor creation

As mentioned above, the companion object must have the same name as the a class in scala, which in our case is the actor `RareBooks`. A companion object is where you will want to put much of your processing code that does not require actor integration. This will allow you to unit test your code without having to spin up the actor system.

Listing 4.5 The RareBooks Actor

```
class RareBooks extends Actor with ActorLogging with Stash {     ❶

  import context.dispatcher      ❷
  import RareBooks._             ❷
  import RareBooksProtocol._     ❷

  private val openDuration: FiniteDuration = ...      ❸

  private val closeDuration: FiniteDuration = ...     ❸

  private val findBookDuration: FiniteDuration = ... ❸

  private val librarian = createLibrarian()           ❹

  var requestsToday: Int = 0     ❺
  var totalRequests: Int = 0     ❺

  context.system.scheduler.scheduleOnce(openDuration, self, Close)        ❻

  /**
   * Set the initial behavior.
   *
   * @return partial function open
   */
  override def receive: Receive = open     ❼


  /**
   * Behavior that simulates RareBooks is open.
   *
   * @return partial function for completing the request.
   */
  private def open: Receive = {     ❽
    case m: Msg =>                  ❾
      librarian forward m           ❿
      requestsToday += 1
    case Close =>                   ⓫
      context.system.scheduler.scheduleOnce(closeDuration, self, Open)     ⓬
      log.info("Closing down for the day")
      context.become(close)         ⓭
      self ! Report
  }

  /**
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

```
   * Behavior that simulates the RareBooks is closed.
   *
   * @return partial function for completing the request.
   */
  private def close: Receive = {     14
    case Open =>                     15
      context.system.scheduler.scheduleOnce(openDuration, self, Close)     16
      unstashAll()                   17
      log.info("Time to open up!")
      context.become(open)           18
    case Report =>                   19
      totalRequests += requestsToday
      log.info(s"$requestsToday ... requests processed = $totalRequests")
      requestsToday = 0
    case _ =>
      stash()     20
  }

  /**
   * Create librarian actor.
   *
   * @return librarian ActorRef
   */
  protected def createLibrarian(): ActorRef = {     0
    context.actorOf(Librarian.props(findBookDuration), "librarian")
  }
}
```

1. Actor definition with logging mixin
2. Local imports
3. Local private immutable values for open, close and find book simulation durations
4. Local private immutable value for librarian actor
5. Local private mutable variables for state of request today and total requests
6. Bootstrap the opening of rare books
7. Set the initial actor behavior
8. Actor behavior when open
9. Match against marker trait Msg for open behavior
10. Forward the message to the Librarian actor
11. Match against Close message to simulate closing for the day
12. Schedule once the next time RareBooks will be open
13. Change actor behavior to close
14. Actor behavior when closed
15. Match against Open message to simulate opening for the day
16. Schedule one the next closing
17. Unstash all messages stashed while simulating closed
18. Change actor behavior to open
19. Match against Report message for showing how requests processed report
20. Stash all messages not processed by close state
0. Factory method for creating the Librarian child actor

The `RareBooks` actor is now complete, at least for this stage of our analogy. That being said, however, there is a lot going on here, so let's break it down. For the purposes of demonstrating reactive concepts, we have chosen to use analogical reasoning as we explained at the beginning of this chapter. In so doing, we must build into our model the realities that

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

mimic our analogy. In other words, we must algorithmically simulate impediments that `RareBooks` will face, such as the hours of operation, managing employees (the `Librarian`) to name a few. These difficulties will allow us to reason through the various obstacles one runs into when building an application and see how we apply reactive patterns to overcome.

We will start with a simple table that lays out the impediments `RareBooks` will face, and the technology used for simulation.

Table 3.3 Analogy - RareBooks Impediment Explanation

Analogy - RarerBooks Impediment	Technology
Rare Books LLC, has hours of operation that we must account for. This is known as the librarian's behavior, and comes in two forms: • RareBooks is open for business. • RareBooks is closed for the day.	The actor has two `akka.actor.Receive` (behavior) loops: • `open` - the actor processes all messages except for `Done`. • `close` - the actor will only process the `Open` and `Report` message and `stash` all others.
When open, `RareBooks` must process messages until it is time to close for the day	`scala.concurrent.duration.FiniteDuration`: • simulates the amount of time until the next opening. • `com.rarebooks.library.RareBooks.Close`: • `Close` is the message that is used to `scheduledOnce` the next opening and trigger the change in actor behavior • `akka.actor.Scheduler.scheduleOnce`: • simulates the amount of time until the next opening. • `akka.actor.become`: • changes the actors behavior by hot-swapping their receive loop from `open` to `close`.
When closed `RareBooks` is not actively processing book information requests but cannot lose them.	`scala.concurrent.duration.FiniteDuration`: • simulates the amount of time until the next closing. • `com.rarebooks.library.RareBooks.Open`: • `Open` is the message that is used to `scheduledOnce` the next closing, unstash all messages and trigger the change in actor behavior. • `akka.actor.Scheduler.scheduleOnce`: • simulates the amount of time until the next closing. • `akka.actor.StashSupport.unstashAll`: • Un-stash all messages received while closed. • `akka.actor.become`: • changes the actors behavior by hot-swapping their receive loop from

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

	`close` **to** `open`.
	`akka.actor.StashSupport.stash`:
	• **stashes the messages received other than** `Open` **and** `Report` **while closed.**

CODING THE LIBRARIAN

Now that we've seen the `RareBooks` actor, let's take a look at what our child `Librarian` actor will look like. We will start with an outline of what the actor structure will contain and then model the actor in scala.

`Librarian.scala` - This is the source file that contains librarian related code.

- `object Librarian` - As mentioned above, this is the singleton companion object. It will contain local imports, message definitions, a properties factory and other local functions for processing.
  - `import Catalog._`
  - `import RareBooksProtocol._`
  - `case class Done` - local message used for simulation
  - `def props(findBookDuration ...): Props` - properties factory for creation.
  - `private def optToEither[...` - function for converting an option to an Either.
  - `private def findByIsbn`
  - `private def findByAuthor`
  - `private def findByTitle`
  - `private def findByTopic`
- `class Librarian`
  - `constructor (findBookDuration ...)` - used for simulation
  - `extends Actor with ActorLogging with Stash`
  - `import context.dispatcher` - used for simulation
  - `import Librarian._`
  - `import RareBooksProtocol._`
  - `override def receive: Receive = ready` - set default receive behavior
  - `private def ready: Receive = {` - behavior when ready
  - `private def busy: Receive = {` - behavior when busy
  - `private def research(d: Done)` - used for simulation
  - `private def process(...` - private method for processing response.

Listing 4.6 The Librarian. scala File

```
package com.rarebooks.library                          (1)

import akka.actor.{ Actor, ActorRef, ActorLogging, Props, Stash }    (2)
import scala.concurrent.duration.FiniteDuration   (2)
```

(1) **Package containing the Librarian.scala file**
(2) **File level imports shared by the companion object and actor**

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

The `Librarian.scala` file is the container for for both the companion object and actor.

Listing 4.7 The Librarian Companion Object

```
object Librarian {          ❶

  import Catalog._
  import RareBooksProtocol._

  final case class Done(    ❷
      e: Either[BookNotFound, BookFound],
      customer: ActorRef)

  def props(findBookDuration: FiniteDuration): Props =    ❸
      Props(new Librarian(findBookDuration))

  /**
   * Convert option to either function.
   *
   * @param v input for function
   * @param f function to match against
   * @tparam T type for Either
   * @return on success return Right[BookFound]
   *         otherwise return Left[BookNotFound]
   */
  private def optToEither[T](    ❹
      v: T,
      f: T => Option[List[BookCard]]): Either[BookNotFound, BookFound] =
    f(v) match {
      case b: Some[List[BookCard]] =>
        Right(BookFound(b.get))
      case _ =>
        Left(BookNotFound(s"Book(s) not found based on $v"))
    }

  ... other private defs

  /**
   * Convert option to either for validation.
   *
   * @param fb find book command
   * @return either list of books or error
   */
  private def findByTitle(fb: FindBookByTitle) =    ❺
    optToEither[String](fb.title, findBookByTitle)

  /**
   * Convert option to either for validation.
   *
   * @param fb find book command
   * @return either list of books or error
   */
  private def findByTopic(fb: FindBookByTopic) =    ❺
    optToEither[Set[Topic]](fb.topic, findBookByTopic)
}
```

❶ **Singleton companion object for messages, properties factory and local functions**

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

2. private Done immutable message
3. Properties factory for actor creation
4. private method for converting Option to Either
5. Private method for finding book by ...

Listing 4.8 The LIbrarian Actor

```
class Librarian(findBookDuration: FiniteDuration)
    extends Actor with ActorLogging with Stash {     1

  import context.dispatcher      2
  import Librarian._             2
  import RareBooksProtocol._     2

  /**
   * Set the initial behavior.
   *
   * @return partial function ready
   */
  override def receive: Receive = ready     3

  /**
   * Behavior when ready to receive a find book request
   *
   * @return partial function for completing the request
   */
  private def ready: Receive = {     4
    case m: Msg => m match {
      case c: Complain =>            5
        sender ! Credit()
        log.info(s"Credit issued to customer $sender()")
      case f: ... match against other messages     5
      case f: FindBookByTopic =>     5
        research(Done(findByTopic(f), sender()))
    }
  }

  /**
   * Behavior simulating the need research the customer's request.
   *
   * @return partial function for completing request
   */
  private def busy: Receive = {     6
    case Done(e, s) =>              7
      process(e, s)                 8
      unstashAll()                  9
      context.become(ready)         10
    case _ =>     11
      stash()     12
  }
  /**
   * Simulate researching for information by scheduling completion of
   * the task for a given duration.
   *
   * @param d simulation is done
   */
  private def research(d: Done): Unit = {     13
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

```
    context.system.scheduler.scheduleOnce(findBookDuration, self, d)     (14)
    context.become(busy)     (15)
  }


  private def process(r: Either[BookNotFound, BookFound]): Unit = {      (16)
    r fold (
      f => {
        sender ! f          (17)
        log.info(f.toString)
      },
      s => sender ! s)      (17)
  }
}
```

(1) Actor definition with logging mixin
(2) Local imports
(3) Set the initial actor behavior
(4) Actor behavior when ready to receive book request
(5) Match against marker trait Msg for ready behavior
(6) Actor behavior for book research simulation
(7) Match against local Done message when simulation is complete
(8) Private helper method for processing results of simulation
(9) Unstash all messages stashed while simulating research
(10) Change behavior back to ready
(11) Match against all other messages received when busy
(12) Stash all messages not processed by busy state
(13) Simulate being busy by doing research
(14) Schedule for when simulation complete
(15) Change behavior to busy
(16) Private helper method for validating message and response.
(17) `!` (the bang operator) uses “fire-and-forget” to send the message asynchronously

!, ->, =>, etc.

Typically in a language operator constructs such as +, -, *, etc. are native to the language. In other words, they are reserved keywords that implement a specific behavior. Such is not the case in Scala. What we normally would consider a reserved keyword such as + is a method on a given class. For example, if you look at the Int class in the Scala docs, we find that + is an overloaded method that takes a single numeric parameter such as Int, Float, Long, etc.

The reality of this construct is that in Scala when you see 2 + 2 what it translates to is 2.+(2). The recommended practice is to use dot notation with non-operator based method invocation and infix notation with operator based methods. This infix notation is also the case with Akka when using scala. For example, we use "!" (bang) which equates to the tell method for fire-and-forget, and "?:" which equates to ask when expecting a response.

The `Librarian` actor is now complete. As with `RareBooks`, there is a lot going on here as well. Again, in keeping the spirit of our analogy, let's look at the impediments that our librarian will face.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

Table 3.4 Analogy - Librarian Impediment Explanation

Analogy - Librarian Impediment	Technology
The librarian can only do one thing at a time. This is known as the librarian's behavior, and comes in two forms: • Librarian is ready to open letters and process them. • Librarian has opened a book information request letter and is busy researching.	The actor has two `akka.actor.Receive` (behavior) loops: • `ready` - the actor processes all messages except for Done. • `busy` - the actor will only process the `Done` message and `stash` all others.
Researching a book information request takes time.	`scala.concurrent.duration.FiniteDuration`: • simulates the amount of time it takes to research a book information request.
When ready, the librarian opens a letter for a book request and begins researching	`akka.actor.Scheduler.scheduleOnce`: • this simulates the temporal boundaries for researching. `akka.actor.become`: • changes the actors behavior by hot-swapping their receive loop from `ready` to `busy`.
While researching, librarian can not process other requests that pile up in their inbox.	`akka.actor.StashSupport.stash`: • stashes the messages received other than `Done` while `busy`.
When the librarian is done researching, he is once again ready to process new requests.	`com.rarebooks.library.Librarian.Done`: • `Done` is the message that was `scheduledOnce` to simulate completion. `akka.actor.StashSupport.unstashAll`: • Un-stash all messages received while `busy`. `akka.actor.become`: • changes the actors behavior by hot-swapping their receive loop from `busy` to `ready`.

CODING THE CUSTOMER

Now, let's take a look at the `Customer` actor. The `Customer` is a top-level actor just like `RareBooks` and follows a similar pattern, as the `Librarian`, but with a new twist, the management of internal state. Let's start with an outline:

`Customer.scala` - This is the source file that contains customer related code.

- `object Customer` - The singleton companion object. It will contain local imports, message definitions, a properties factory and other local functions for processing.
  - `import RareBooksProtocol._`

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

    - `def props(rareBooks: ActorRef, ...): Props` - props factory for creation.
    - `case class CustomerModel` - immutable class modeling the customer.
    - `private case class State(...)` - immutable state container.
- `class Customer(rareBooks: ActorRef, ...) extends Actor with ActorLogging`
    - `import Customer._`
    - `import RareBooksProtocol._`
    - `private var state = State(...)` - threadsafe internal mutable state
    - `override def receive: Receive = ...`
    - `private def requestBookInfo()` - method for requesting book
    - `private def pickTopic: Topic` - used for simulation

Listing 4.9 The Customer.scala

```
package com.rarebooks.library     (1)

import akka.actor.{ ActorRef, Actor, ActorLogging, Props }     (2)
import scala.util.Random          (2)
```

(1) **Package containing the Librarian.scala file**
(2) **File level imports shared by the companion object and actor**

The `Customer.scala` file is the container for for both the companion object and actor.

Listing 4.10 The Customer Companion Object

```
object Customer {                    (1)

  import RareBooksProtocol._

  def props(rareBooks: ActorRef, odds: Int,
        tolerance: Int): Props =     (2)
    Props(new Customer(rareBooks, odds, tolerance))

  /**
   * Customer model.
   *
   * @param odds the customer's odds of finding a book
   * @param tolerance the customer's tolerance for BookNotFound
   * @param found the number of books found
   * @param notFound the number of books not found
   */
  case class CustomerModel(          (3)
    odds: Int,
    tolerance: Int,
    found: Int,
    notFound: Int)

  /**
   * Immutable state structure for customer model.
   *
   * @param model updated customer model
   * @param timeInMillis current time in milliseconds
   */
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

```
  private case class State(model: CustomerModel, timeInMillis: Long) {     ❹
    def update(m: Msg): State = m match {
      case BookFound(b, d) =>
        copy(model.copy(found = model.found + b.size), timeInMillis = d)
      case BookNotFound(_, d) =>
        copy(model.copy(notFound = model.notFound + 1), timeInMillis = d)
      case Credit(d) =>
        copy(model.copy(notFound = 0), timeInMillis = d)
    }
  }
}
```

❶ Singleton companion object for messages, properties factory and local functions
❷ Properties factory for actor creation
❸ Immutable case class for CustomerModel representing current state
❹ Private immutable state manager class

Listing 4.11 The Customer Actor

```
/**
* Customer actor.
*
* @param rareBooks reference to RareBooks actor
*/
class Customer(rareBooks: ActorRef, odds: Int, tolerance: Int)
    extends Actor with ActorLogging {  ❶

 import Customer._                     ❷
 import RareBooksProtocol._            ❷

 // initialize the customer state
 private var state = State(CustomerModel(odds, tolerance, 0, 0), -1L) ❸

 // bootstrap customer requests
 requestBookInfo()

 override def receive: Receive = {     ❹
    case m: Msg => m match {
      case f: BookFound =>             ❺
        state = state.update(f)        ❻
        log.info(f"{} Book(s) found!", f.books.size)
        requestBookInfo()
      case f: BookNotFound             ❺
          if state.model.notFound < state.model.tolerance =>          ❼
        state = state.update(f)        ❻
        log.info(f"{} Book(s) not found! My tolerance is {}.",
          state.model.notFound, state.model.tolerance)
        requestBookInfo()
      case f: BookNotFound =>          ❺
        state = state.update(f)        ❻
        sender ! Complain()            ❽
        log.info(
          f"{} Book(s) not found! Reached my tolerance of {}. Sent complaint!",
          state.model.notFound, state.model.tolerance)
      case c: Credit =>                ❺
        state = state.update(c)        ❻
        log.info("Credit received, will start requesting again!")
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

```
        requestBookInfo()
      case g: GetCustomer =>              ❺
        sender ! state.model              ❾
    }
  }

  /**
   * Method for requesting book information by topic.
   */
  private def requestBookInfo(): Unit = ❿
    rareBooks ! FindBookByTopic(Set(pickTopic))

  /**
   * Simulate customer picking topic to request. Based on the odds they will randomly pick a
       viable topic, otherwise
   * they will request Unknown to represent a topic that does not exist.
   *
   * @return topic for book information request
   */
  private def pickTopic: Topic =          ⓫
    if (Random.nextInt(100) < state.model.odds)
      viableTopics(Random.nextInt(viableTopics.size)) else Unknown
}
```

❶ Actor definition with logging mixin.
❷ Local imports.
❸ private local mutable state.
❹ Receive loop for processing messages
❺ Match against marker trait Msg for ready behavior
❻ Update state based on message
❼ Guard statement against match
❽ Complain message sent to sender when notFound exceeds tolerance
❾ Customer current state sent to sender
❿ Method used for requesting book information
⓫ Method used for simulating the picking of a book topic

As mentioned above, the `Customer` actor is similar to the `Librarian`, with one distinct difference: the management of local mutable state. For now, we track the number of books found, not found, the odds of requesting viable book information and the customer's tolerance for error. As the analogy progresses, we will add to this notion of state management by adding more properties, and in Chapter 5 we will introduce the persistence of this state with Akka Persistence.

What is a Match Guard?

In Scala, a `match` guard is syntactic sugar that provides more readable syntax for `case` statements. You can think of them as a filter. The reason they work is that unlike a Java `Switch` statement, Scala `case` statements do not fall through. As a result, you can place a "guard" against a `match` and be assured that if `true`, the case will exit.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

Before we move to the next topic, elasticity, let's review the impediments included in our customer and view from a reactive perspective what we have achieved thus far.

Table 3.5 Analogy - Customer Impediment Explanation

Analogy - Customer Impediment	Technology
The Customer impediments come in three forms: • **Initiating the request process.** • **Occasionally requesting information on a book that does not exist.** • **Stop requesting information when tolerance for books not found is exceeded.**	**In the Customer constructor we initiate the request process, randomly request book information that does not exist and stop requesting information when the customer's tolerance is exceeded.**
Customer initiates request process	`requestBookInfo`: • **helper method that initiates the cycle.**
Customer occasionally requests information on a book that does not exists.	`pickTopic`: • **uses** `scala.util.Random` **and** `customer.model.odds` **to simulate occasionally requesting erroneous book information**

WHAT WE'VE ACHIEVED REACTIVELY SO FAR

- Responsive
  - The participants communicate asynchronously through message passing. This removes delays due to synchronous processing.
  - The immutable message structure avoids delays that otherwise could occur due to concurrency faults.
- Resilient
  - The share nothing philosophy through isolation of state avoids concurrency faults that could occur due to shared mutable state.
  - The immutable message structure removes the possibility of mutations to the state that is messaged avoiding potential concurrency faults.
- Elastic
  - The loosely coupled design through message passing support vertical and horizontal scaling.
- Message Driven
- The actor model provides the message passing semantics for message-driven architecture.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

4.2.3 Running the Application

To run our application, we will use Simple Build Tool or `'sbt'` for short. We are not going to go into detail about `'sbt'` as that is beyond the scope of the book for now, but you can find the full details and documentation at http://www.scala-sbt.org/. From a terminal window, we change directory to the root application `'reactive-application-development-scala'` and type `'sbt'` where you will see the following output:

```
[info] Loading global plugins from ...
[info] Loading project definition ...
[Info] Set current project to ...
>
```

From the `'>'` we type `'project chapter3_001_messaging'` this will change the project to the one we just laid out. Now we type `'run'` and you should see the following:

```
[info] Running com.rarebooks.library.RareBooksApp
...
```

Enter commands [`q` = quit, `2c` = 2 customers, etc.]:

Enter `'2c'` for two customers and the application will start processing messages between `RareBooks`, `Librarian` and `Customers`. Since we have not discussed output as of yet, specifically UI related, all results are logged to `'rarebooks.log'` in the root directory. A good way to watch what is going on is to tail the log file while the app is running like so:

```
$ tail -f rarebooks.log
```

In the log file, you should see something similar to the following which shows the actor system starting up, the creation of the `RareBooks`, `Librarian`, and `Customer` actors. In addition, you will see a sequence of messages flowing back and forth, and the simulation of `RareBooks` opening and closing. One thing to note is the throughput. Depending on your machines specifications, performance may vary, but you should see around 10 or 11 messages processed per day.

Congratulations! You have created and run beginning of your first reactive application.

4.3 Elasticity

We have spent a great deal of time laying out our analogy and establish our overall system in a reactive format. As a result, we have the groundwork for moving forward and exploring more complex concepts like elasticity in detail. Let's first review what it means to be elastic by reviewing our definition from the Reactive Manifesto.

ELASTICITY DEFINITION

The system stays responsive under varying workload. Reactive systems can respond to changes in load by increasing or decreasing the allocation of resources.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

Now that we have reviewed what being elastic means let's consider the current state of our analogy and see if it is elastic. As of now, our owner has one employee, the Librarian, who is responsible for processing all requests. We have built into our analogy through simulation that processing (researching a request for book information) takes time. As you can imagine when the number of customers increases (load), the overall response time will increase as well. In fact, the increase is proportional to the number of customers requesting. If I have two customers and it takes two seconds to process each request, then to satisfy both customers, it will take a total of four seconds. Essentially we have a first come first serve model. If you are late to the game, you may have to wait a long time to play. The question is: How do we best solve this problem? In other words, how do we make our application elastic? From our owners perspective, it is pretty obvious he needs to hire more librarians. In computational terms, he needs to process requests in parallel.

Before we discuss the implementation, Let's review the difference between concurrency and parallelism from chapter two. Concurrency and parallelism are sometimes confused to mean the same thing, but while related, there are some differences.

- *Concurrency* improves throughput by allowing two or more tasks to make progress in a non-determined fashion, which may or may not run simultaneously. As a result, concurrent programming focuses on the complexity that arises from that non-deterministic control flow, as we discussed in chapter one under tightly coupled middleware.
- *Parallelism*, on the other hand, occurs when the execution of the tasks happen simultaneously, parallel programming focuses improving throughput by making flow control deterministic.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

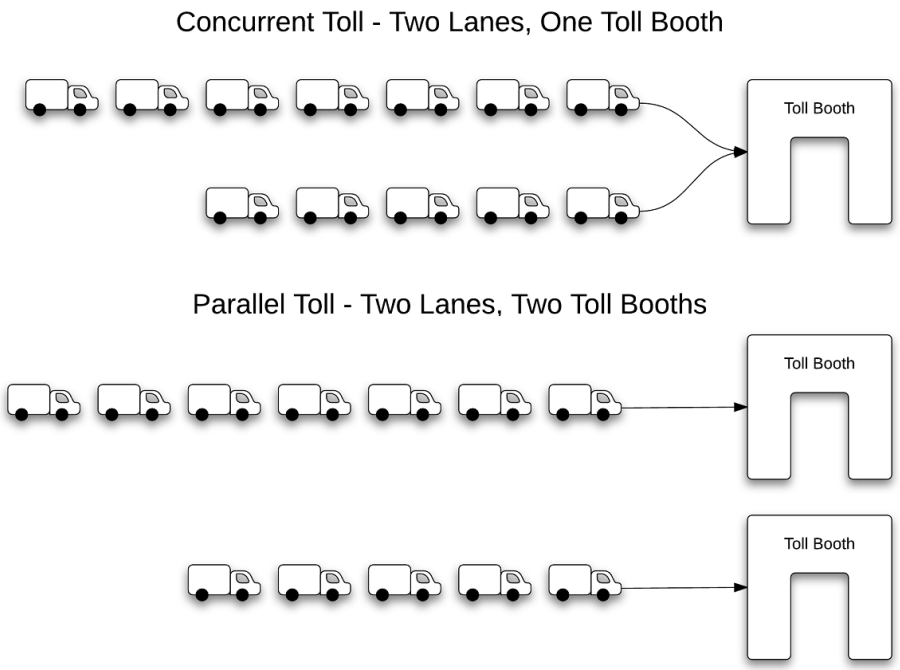

Figure 4.1 Concurrency and Parallelism Compared

The figure clearly shows how a single toll booth becomes a blocker and slows down the entire line of traffic. Adding a second toll booth removes the block and allows much greater traffic flow.

Now that we reviewed what parallelism means and how it provided elasticity let's code it for our analogy.

4.3.1 Akka Routing

From our analogy, we have clearly realized that the owner needs to hire more librarians. In doing so, the owner will be able to reduce the overall response time for customer request by processing them simultaneously. As we mentioned above, computationally, we call this parallelism. To achieve parallelism in Akka, we will use a router. In Akka, you can think of a Router as an actor that proxies messages from the set of other actors whom we call routees.

In order to manage the "proxying" of these messages, the router will use one of the several routing logic patterns that we implement by way of code or configuration.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

ROUTING LOGIC

The Router implementation in Akka is very powerful as it provides a variety of routing logic strategies that can be applied based on an application's needs as well as the possibility of creating your own. The Following table lists the routing logic that ship with Akka:

Table 3.6 Akka Routing Logic

Routing Logic	Description
`RoundRobinRoutingLogic`	A pattern of ordering where items that are encountered (messages) are processed sequentially in a circular manner.
`RandomRoutingLogic`	Uses a random number with a range bound by the number of routees to select the routee to employ.
`SmallestMailboxRoutingLogic`	Attempts to send to a non-suspended routee with the least number of messages in their mailbox.
`BroadcastRoutingLogic`	Broadcasts the message to all routees.
`ScatterGatherFirstCompletedRoutingLogic`	Broadcasts the message to all routees and replies with the first response.
`TailChoppingRoutingLogic`	Sends the message to a random picked routee and waits for a specified interval and then sends to another random picked routee until one cycle is complete.
`ConsistentHashingRoutingLogic`	Uses consistent hashing to select a routee based on the message received.

As you can see, there are several out-of-the-box routing strategies available. The routees themselves are ordinary child actors wrapped with `ActorRefRoutee`, which denotes it is a routee. `ActorRefRoutee` also provides value members for referencing the routee, `ref`, and sending messages, `send`.

Now let's take a look at the type of Routers Akka supports which come in two flavors, Pooled routers, and Group Routers.

POOLED ROUTERS

Pooled routers are where the router creates and manages the routees as children and is responsible for their supervision. The settings for pooled routers are either based on configuration or code, with the requirement that the creation of a router is done programmatically. Let's take a look at the two different ways of defining settings for a pooled router.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

Listing 4.12 Pool Router Settings via Configuration

```
akka.actor.deployment {          (1)
  /rarebooks/librarian {         (2)
    router = round-robin-pool    (3)
    nr-of-instances = 10         (4)
  }
}
```

(1) Deployment configuration section of Akka conf.
(2) Path of router.
(3) Type of router.
(4) Number of routers in the pool

As you can see, the configuration is straightforward for router settings. Using this configuration, let's look at how we would create this router.

Listing 4.13 Pooled Router Settings via Configuration

```
val librarian: ActorRef =              (1)
  context.actorOf(FromConfig.props(    (2)
    Librarian.props(findBookDuration), "librarian"))
```

(1) Immutable value reference to the Actor.
(2) Create the actor based on configuration properties.

One thing you will immediately notice, is there are no actual changes to the actor that we intend to use as a router. If you stop and think about this for a moment, it is quite amazing. By simply modifying our configuration file and making a slight change to the way we create our actor, we have implemented parallelization.

Before we move onto group routes let's look at the final scenario of configuring and creating a pooled router programmatically as this is similar to how we will do it in our analogy.

Listing 4.14 Pooled Router via Programmatic Configuration and Creation

```
val librarian: ActorRef =                    (1)
  context.actorOf(RoundRobinPool(5).props(   (2)
    Librarian.props(findBookDuration)), "librarian")
```

(1) Immutable value reference to the Actor.
(2) Programmatic configuration and creation.

GROUP ROUTERS

Group routers differ from pooled routers in that they allow the routees to be created externally and then associate to a router by their specified actor path. The key difference here is that the router is not the parent of the routee and is not responsible for their supervision. As with a pooled router, a group router's settings are defined in either configuration or code and must be created programmatically. Let's look at these two scenarios in code.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

Listing 4.15 Group Router Settings via Configuration

```
akka.actor.deployment {                    (1)
  /rarebooks/librarian-router {            (2)
    router = round-robin-group             (3)
    routees.paths = [
        "/user/rarebooks/librarian-1",     (4)
        "/user/workers/librarian-2",       (4)
        "/user/workers/librarian-3"]       (4)
  }
}
```

(1) Deployment configuration section of Akka conf.
(2) Path of router.
(3) Type of router.
(4) Path to actors who will be part of the group.

Listing 4.13 Group Router Creation via Configuration

```
val librarian: ActorRef =                  (1)
  context.actorOf(FromConfig.props(        (2)
    Librarian.props(findBookDuration), “librarian”))
```

(1) Immutable value reference to the Actor.
(2) Create the actor based on configuration properties.

Again, with no change to the actor code for the routee, we have achieved parallelization in our system. Now let's look at how to create a group router with settings programmatically.

Listing 4.14 Group Router via Programmatic Configuration and Creation

```
val paths = List(                          (1)
    "/user/rarebooks/librarian-1",
    "/user/workers/librarian-2",
    "/user/workers/librarian-3")
val librarian: ActorRef =                  (2)
  context.actorOf(RoundRobinGroup(paths).props(     (3)
    Librarian.props(findBookDuration)), "librarian")
```

(1) List of paths for routees
(2) Immutable value reference to the Actor.
(3) Programmatic configuration and creation.

DISPATCHERS

At the heart of Akka's messaging system is what are called Dispatchers. They are the underlying gears that make the Actors operate. You can think of them kind of like an engine. Essentially they coordinate the communication between actors implementing `scala.concurrent.ExecutionContext` and registering an actor's mailbox for execution. They work in concert with an Executor (essentially a grouping of threads) and provide the execution time and context in which actors operate. In doing so, they provide the foundation for

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

parallelism (via the Executor) in that they can be "tuned" to the underlying cores within a machine. Akka provides four types of dispatchers:

- Dispatcher: This is the default dispatcher if none are defined
  - Every actor has its own mailbox
  - The dispatcher can be shared amongst any number of actors
  - Can be backed by the "fork-join-executor", "thread-pool-executor"
  - Good for Bulkheading, optimized for non-blocking
- PinnedDispatcher: Dedicates a unique thread for each actor
  - Every actor has its own mailbox
  - The dispatcher cannot be shared
  - Backed by "thread-pool-executor"
  - Good for heavy I/O blocking operations
- BalancingDispatcher: Designed to redistribute work from busy actors to idle ones
  - All actors share a single Mailbox
  - Only actors of the same type can share the dispatcher
  - Can be backed by the "fork-join-executor", "thread-pool-executor"
  - Good for work sharing
- CallingThreadDispatcher: Runs on the current thread only, and does not create new threads
  - Every actor has it's own mailbox per thread on demand
  - The dispatcher can be shared amongst any number of actors
  - Can be backed by the calling thread
  - Good for testing

4.3.2 Librarian Router

Now that we have a basic understanding of how parallelization works in Akka by the way our routers, let's go back to our analogy and address our problem of throughput. You will recall our owner realizes as the number of customers increase, his ability with only one librarian to service those request decreases proportionally to the number of customers. As a result, our owner realized that he must hire more librarians or face the potential of wrath of his customers. As you can guess, this translates into use routers.

In order to achieve this in our code, we will need to change two files, our `application.conf` and `Librarian.scala`. Let's look at the application.conf file changes first.

Listing 4.15 application.conf Changes

```
rare-books {
  open-duration = 20 seconds
  close-duration = 5 seconds
  nbr-of-librarians = 5     ❶
  librarian {
    find-book-duration = 2 seconds
  }
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

```
}
```

1. Setting for the number of librarian routees to create.

Now you might be asking the question, *"Wait a second, I thought the router settings thing went under `akka.actor.deployment`?"*. Well, you are correct, but in our case we will be programmatically configuring and creating our routees. This `nbr-of-librarians` setting is merely a convenience, for adjust the number of routees we will create in the code. Now let's look at the changes we need to make to our `RareBooks` actor.

Listing 4.16 RareBooks Changes

```
import akka.actor.{ Actor, ActorLogging, Props, Stash }                    (1)
import akka.routing.{ ActorRefRoutee, Router, RoundRobinRoutingLogic }     (2)
...
  private val nbrOfLibrarians: Int =
      context.system.settings.config
        getInt "rare-books.nbr-of-librarians"     (3)
...
  var router: Router = createLibrarian()          (4)
...
  private def open: Receive = {
    case m: Msg =>
      router.route(m, sender())                   (5)
...
  /**
   * Create librarian as router.
   *
   * @return librarian router reference
   */
  protected def createLibrarian(): Router = {     (6)
    var cnt: Int = 0                              (7)
    val routees: Vector[ActorRefRoutee] = Vector.fill(nbrOfLibrarians) {     (8)
      val r = context.actorOf(
        Librarian.props(findBookDuration), s"librarian-$cnt")                (9)
      cnt += 1
      ActorRefRoutee(r)                           (10)
    }
    Router(RoundRobinRoutingLogic(), routees)     (11)
  }
```

1. Remove ActorRef import.
2. Add routing ActorRefRoutee, Router and RoundRobinRoutingLogic imports.
3. Get the number of librarian routees to create from configuration.
4. local mutable reference is a Router instead of an ActorRef.
5. We route the message instead of forwarding.
6. createLibrarian returns a Router instead of an ActorRef.
7. local mutable variable for indexing routees.
8. Vector of ActorRefRoutee.
9. Creation of individual librarian.
9. Wrap librarian in ActorRefRoutee
11. Create Router with RoundRobinRoutingLogic and the list of routees

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

Our owner is now happy as he can satisfy many more customers in parallel. If you are wondering why we chose to programmatically configure our router, it's because we want to have more control over our supervision of our routees, which is coming up next!

4.3.3 Running the Application

Assuming you are still in `'sbt'` you will want to change your project to `'chapter3_002_elasticity'` and run it again. This time, however, let's enter more customers by typing `'5c'` for five customers. Tail the log file again and take note of the significant increase in performance! Even with more than double the number of customers, we get almost five times the performance! That is pretty amazing for a few lines of new code.

4.4 Resilience

So far, we have explored the basis of reactive programming via message-driven communication (the base) and elasticity (a pillar). Now we will take a look at the other pillar, resilience. Before we get started, however, we need to update our analogy to simulate a fault that in our code will reflect the real world interactions our owner would have with his employees and customers.

Successful business owners usually put their heart and soul into their endeavours. In so doing, quality is of the utmost importance. However, as the company grows and additional resources come on board sometimes a compromise in quality can occur. One can expect this to some extent as new employees many times do not have the experience, and there can be a lengthy learning curve. Such is the case for our owner that we will simulate as a fault in Akka and show how we can overcome these problems through the use of supervision.

When our business owner was a sole proprietor, he prided himself in always responding back to the customer. When a customer reached, their tolerance for "books not found" the owner would send them a credit, which in turn would resolve the issue. He would never allow the number of complaints to stress him, as he considered it a standard part of doing business (think "let it fail"). Unfortunately, this is not the case with the newly hired librarians. While they are committed to doing a good job, they do not have the owner's stamina for grievances. As a result, after a certain number of complaints, our employees become frustrated. While frustrated, they must take a break and are unable to process requests, but even worse, they lose track of the current complaint resulting in the customer never receiving a credit. This is a serious problem. While the employee will recover and start working again, the customer will not. Since they never received a credit, of which the worker lost track, eventually all customer requests will stop.

4.4.1 Faulty Librarian Actor

In Akka terms, we call this a fault and will model it as an exception. When one of our librarians exceeds their tolerance for complaints, they will become frustrated and throw a `ComplainException`. When this exception occurs, Akka will pause the `Librarian` routee and

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

escalate to the parent who is `RareBooks`. Since `RareBooks` uses the default supervision strategy (which we will talk about shortly), it will restart the paused librarian effectively resetting their `complainCount` to zero. At this point, the customer, who complained, is in a state where they will no longer request information as they did not receive a credit.

Let's start by making the number of tolerable complaints a property of librarian in the `application.conf` and add it as a parameter to the properties factory.

Listing 4.17 Faulty Librarian Actor application.conf Changes

```
...
rare-books {
  open-duration = 20 seconds
  close-duration = 5 seconds
  nbr-of-librarians = 5
  librarian {
    find-book-duration = 2 seconds
    max-complain-count = 2     (1)
  }
}
```

(1) Maximum number of tolerable complaints for a given librarian

Listing 4.18 Faulty Librarian Actor Changes

```
object Librarian {
  ...
  final case class ComplainException(c: Complain, customer: ActorRef) extends
       IllegalStateException("To many complaints!")              (1)
  ...
  def props(findBookDuration: FiniteDuration, maxComplainCount: Int): Props =
    Props(new Librarian(findBookDuration, maxComplainCount))    (2)
  ...
class Librarian(findBookDuration: FiniteDuration, maxComplainCount: Int)    (3)
    extends Actor with ActorLogging with Stash {
  ...
  private var complainCount: Int = 0            (4)
  ...
  private def ready: Receive = {
    case m: Msg => m match {
      case c: Complain if complainCount == maxComplainCount =>  (5)
        throw ComplainException(c, sender())    (6)
      case c: Complain =>     (7)
        complainCount += 1    (8)
        sender ! Credit()     (9)
        log.info(s"Credit issued to customer $sender()")
  ...
```

(1) Immutable ComplainException case class
(2) Properties factory takes maxComplainCount as parameter
(3) Constructor takes maxComplainCout as parameter
(4) Local mutable state for number of complaints received
(5) Scala guard on match if current complaint count equals max complaint count
(6) Throws ComplainException
(7) Otherwise process the Complain messages as normal

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

8. Update the complainCount
9. Sends customer a Credit message

Now we have a small change we need to make to our RareBooks actor to fetch the max-complain-count and pass it to the librarian on creation:

Listing 4.19 RareBooks Actor Changes

```
...
class RareBooks extends Actor with ActorLogging with Stash {
  ...
  private val maxComplainCount: Int =      (1)
    context.system.setting.config getInt
      "rare-books.librarian.max-complain-count"
  ...
  protected def createLibrarian(): Router = {
    ...
    val r = context.actorOf(Librarian.props(
      findBookDuration, maxComplainCount), s"librarian-$cnt")     (2)
  ...
```

1. Fetch the max-complain-count from application.conf
2. Pass maxComplainCount upon creation of Librarian routee

4.4.2 Running the Faulty Application

As before, make sure you are at the `'sbt'` prompt and set the project to `'chapter3_003_faulty'`. Run the application with two customers but this time let's set their tolerance for books not found to two as follows:

```
Enter commands [`q` = quit, `2c` = 2 customers, etc.]:
2c2
```

Tail the log file and let the application run for a while. You will notice that after the one of the `Librarians` throws the `ComplainException` that the `Customer` the `Librarian` was serving will stop requesting information, and eventually all customers will stop. The application will continue running, but the number of customer requests processed per day will be zero!

4.4.3 Librarian Supervision

From our analogies perspective, the owner will implement a simple rule to overcome this small but serious issue. Whenever a librarian becomes frustrated and needs a break, they must first inform the owner of their situation. That way the owner can proxy for the librarian and issue the credit in their behalf. In Akka terms, we will achieve this by implementing the appropriate supervision process. You will recall we detailed Akka supervision in Chapter 2 but let's do a quick review. There are two strategies you get out of the box:

- `OneForOneStrategy` (default): This strategy operates just as its name. The supervisor will execute one of four directives (listed below) against the subordinate, and in turn all of the subordinates children.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

- `AllForOneStrategy`: When using this strategy, the supervisor will execute one of the four directives not only against the subordinate that faulted, but also against all subordinates the supervisor manages.

Along with theses strategies, you also get four directives:

1. Resume the subordinate, keeping its accumulated internal state.
2. Restart the subordinate, clearing out its accumulated internal state.
3. Stop the subordinate permanently.
4. Escalate the failure, thereby failing itself.

In our case, the default strategy (`OneForOneStrategy`) and the directive of `Restart` is fine. However, we will need to intercept the exception and extract the customer so we can send them a credit while the faulty librarian is recovering. Let's take a look at what his looks like in code.

Listing 4.20 RareBooks Supervision Implementation

```
...
class RareBooks extends Actor with ActorLogging with Stash {
  ...
  override val supervisorStrategy: SupervisorStrategy = {          (1)
    val decider: SupervisorStrategy.Decider = {                    (2)
      case Librarian.ComplainException(complain, customer) =>      (3)
        customer ! Credit()                                        (4)
        log.info(s"RareBooks sent customer $customer a credit")
        SupervisorStrategy.Restart                                 (5)
    }
    OneForOneStrategy()(decider orElse super.supervisorStrategy.decider)     (6)
  }
...
```

(1) Override the default supervision strategy
(2) Create the Decider for handling ComplainException
(3) Match against Librarian.ComplainException
(4) Send the customer a Credit message
(5) Invoke the Restart directive
(6) Return the OneForOneStrategy with the created Decider or apply the default strategy

What is a Decider?

A decider is just as the name says, it decides what to do in the case of failure. From an Akka point of view, it represents a `PartialFunction[Throwable, Directive]` that applies at the time of failure. The `Decider` maps the child actors fault to the `Directive` that is taken. We should note, as per the Akka documentation if you declare the strategy inside the supervising actor (as opposed to within a companion object) its decider has access to all internal state of the actor. In addition, this is thread-safe, including obtaining a reference to the currently failed child (available as the sender of the failure message).

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

This seemingly simple but powerful model of hierarchical supervision is the key to resilience behind Akka. As mentioned in the sidebar "Resilience vs. Fault Tolerance" in Chapter 2, this style of oversight is much more than fault tolerance. While fault tolerant systems imply the degradation of quality proportional to the severity of the failure, such is not the case with resilience. Resilience embraces failure through expectation and in turn self-heals in the face of failure.

4.4.4 Running the Application

Now that we have fixed our faulty `Librarian` problem let's run the application once more. From the `'sbt'` prompt and set the project to `'chapter3_004_resilience'`. Run the application with two customers and a low tolerance as before:

```
Enter commands [`q` = quit, `2c` = 2 customers, etc.]:
2c2
```

Tail the log file and let the application run for a while and you will see that when one of the `Librarians` throws the `ComplainException`, `RareBooks` steps in and issues a credit to the complaining `Customer`.

4.5 Summary

This has been a somewhat lengthy but foundational chapter. We have learned the basics of reactive programming by way of analogy and created our first reactive application that implements the tenets of the Reactive manifesto. Next, we will take a slight break from code and discuss Domain Driven Design and what it means in reactive computing. Before we end this chapter, let's outline of what we learned.

- Succedaneum and the power of analogical reasoning.
- Used analogical reasoning to layout an analogy
- Translated our analog to Akka terms
- Reasoned about and implement the four tenets of the Reactive Manifesto
- Message Driven
  - Immutable Messages
  - Protocol Objects
  - Async Concurrency with Actors
- Elasticity
  - Concurrency vs. Parallelism
  - Akka Routing
  - Routing Logic
  - Pooled Routers
  - Group Routers
  - Router Settings via Configuration
  - Programmatic Router Settings

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

- Resilience
  - Embracing Failure by Expectation
  - Akka Supervision
  - Supervision Strategies
  - Supervision Directives

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

5

Domain-Driven Design

This chapter covers

- Domain-driven design
- Modeling a flight domain
- The Saga Pattern
- Modeling domain aggregates with Akka
- The Akka Process Manager pattern

Sometimes we humans require some distance to fully comprehend a thing. An example of this is an impressionist painting. When we look at one of these paintings up close, all we see is colored dots and small brush strokes; we don't see art. When we back away from the painting it all becomes clear, and we can see a child on a swing, a park, or a lake. This is exactly the case with a large architecture, a reactive architecture being no exception. In this spirit we will take a some steps back and look at a reactive application from a distance, starting with the next item in the reactive toolbox, *Domain-Driven Design*.

In building our reactive architectures, we need not venture into completely unknown territory. Helpful patterns and technologies already exist that can make programming the reactive way a lot less effort than you think, if you learn to use and embrace some well-known "standard" items in your toolbox. Much like the construction of a house, if one knows how to build a wall, a floor and a roof, it's not too much of a stretch to build the entire house. We'll explore domain-driven design in this chapter; with this approach to domain modeling you can begin building systems that adhere to the Reactive Manifesto, giving you message-driven, elastic, responsive, and resilient capabilities.

Domain-driven design (DDD) is a set of domain development terms, tools, and ideas formalized by Eric Evans in the early 2000s to simplify complex domain modeling; these tools will help you build reactive applications, incorporating the key traits of the Reactive Manifesto:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

- elasticity (react to load): with an easily distributable and individually scalable domain model
- message-driven: immutable, one-way messages to reduce side effects between domains
- resilient: failing sub-domains will not degrade the system behavior on the whole
- responsive: well organized and subdivided domains are more performant than monolithic domains

This chapter will discuss these tools and methodologies and describe both how they work in a traditional sense, as well as their application in the reactive world.

5.1 What is domain-driven design?

Domain-driven design (DDD) is a valuable tool for fleshing out domains and their behavior, a *domain* being a discrete area of functionality designed to solve a software problem. DDD applies to many domains and programming languages, (you'll see it applied to Scala and Akka here). The phrase "Domain-Driven Design" was first referred to in a book bearing the same name by Eric Evans, the father and pioneer of the domain-driven design philosophy.

DDD is the practice of modeling complex systems, with the goal of mapping real-life domain behavior to system behavior. Many of us have been practicing these techniques without knowing they had a name; some might call the use of DDD simply "good architecture." *Immutability,* which we discussed in chapter 2, describes objects that are created only once and never mutated or changed. *Immutable* domain design means *domain entities*, or meaningful items in a domain model, are created only once, and all attributes are set on creation, allowing easy sharing across multiple computer threads and processes. Immutable objects have fewer side effects, as there are no setters (mutator functions that update the state) on any of the attributes on the object. This doesn't mean a domain entity cannot be changed, but the structure of doing so is different than a traditional setter type update; instead of mutating the current state of the object, the object provides a copy of itself with the desired change.

DDD ADDS STRUCTURE AND VOCABULARY

Domain-driven design is a structure of practices and a set vocabulary, called the ubiquitous language, that maps software to a real-life domain. Using DDD practices results in an evolving domain model patterned as closely as possible to the actual functions of the physical domain, whether that is a postal service like we explored in chapter 2 or an airport, which we'll use as a domain example in this chapter.

DDD is not meant for trivial architectures, but what reactive architecture is trivial? One important prerequisite of DDD is access to domain expertise. It is possible to apply the practice by becoming an expert in the domain yourself, but that is difficult and carries some amount of risk due to substitution of actual domain expertise for a developer's best guess. For instance, in the absence of business expertise in an airport domain model, an attribute

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

belonging clearly to ground control may inadvertently be added to tower. Each of these mismatches of the domain vs. real life counterparts obscures the design and makes the overall application of domain-driven design less valuable, so in short do DDD, but do it right or not at all.

DDD sets up a solid foundation at the very beginning of a distributed software design because the domain is divided up, not only into more easily digestible pieces, but also easily distributable pieces, helping satisfy the important elastic aspect of going reactive.

One of the first steps in creating an accurate domain model is to divide the functions of the domain into discrete parts, known as *bounded contexts*, so each can be understood as independent pieces, and the interactions among these pieces can be mapped into the model of the entire domain.

We'll look at bounded contexts in just a bit, but first let's look at what happens in the absence of such domain-driven design: incorrectly planned and monolithic designs, characterized in the following section as a *big ball of mud.*

5.1.1 The big ball of mud

The absence of proper application design has been called a "Big Ball of Mud" (Brian Foote and Joseph Yoder http://www.laputan.org/mud/), meaning a domain design that is is "*haphazardly structured, sprawling, sloppy, duct-tape and bailing wire, spaghetti code jungle.*" This is due to the fact that monolithic domain designs seldom if ever work and quickly become unmanageable. These types of designs are constantly in need of quick repairs, that are not well thought out as part of the architecture as a whole, and as these accumulate the system starts its inevitable decline.

WHAT CREATES A BALL OF MUD?

The forces that contribute to this ball of mud are:

- **Time**

  This is the expectation that there isn't enough time to do the best job or some mad rush to

  get software out to market or before a given season of the year.

- **Experience**

  Insufficient programmer aptitude, inexperience or lack of supervision contribute to a ball of mud.

- **Cost**

  This is the perception that higher quality software will bear too high a cost, which is

  interesting because most times inferior software costs substantially more than doing it right

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

in the first place. Sometimes there just isn't enough money to fund anything more than a

hastily rushed project for a startup only concerned with fiscal survival.

- **Visibility**

  Software, especially back-end software cannot be directly seen or touched. A messy user

  interface would draw criticism and immediate correction, whereas the backend can be built

  in shadows just waiting to fail.

- **Complexity**

  This is a killer. Sometimes software needs to be somewhat complex to solve a problem, but

  when the software houses multiple complexities (bad encapsulation design) or is overly

  complex, it becomes confusing and unmanageable. Complex code is hard to look at, and

  discouraging to maintain.

- **Change**

  Software changes; requirements change. If software is built in a tightly coupled fashion,

  without expectation of change there will be mud.

WHAT DOES THE MUDBALL LOOK LIKE?

The big ball of mud is characterized by the following problems:

- **Throwaway code**

  Solving the immediate problem without regard to the overall design, because of perceived simplicity of a change or to make the change as non-invasive as possible. Throwaway code is usually never really thrown away, as time is never taken to refactor it the right way because of the same short-term thinking that caused it in the first place. This is also known as the *sunk cost fallacy,* where there is an overwhelming and irrational feeling in management that so much money has been spent on a bad project that it's too expensive to throw away.

- **Piecemeal growth**

  This is growth over time and evolution. New York City is an excellent example of piecemeal growth of a city over time. The city started out organized, and then spread inward and outward from what used to be New Amsterdam, the present day Canal Street area up through Harlem. The cities streets are disorganized from downtown to

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

midtown; Broadway even goes diagonal at some point. The city expanded according to need and not according to some grand plan. This may be characterized by the urban sprawl of your codebase. Los angeles, another example of uncontrolled urban sprawl, is pictured below.

Figure 5.1 Los Angeles urban sprawl (http://www.lasmogtown.com/?tag=urban-sprawl)

On one hand, a grand master plan might seem like it would result in a more organized city and code, but the reality is that all things change and planning on a moving target is setting yourself up for failure. On the other hand, growth without planning will end up as a mess, so what to do? The solution is *atomicity* of design, which means designing closely related sections of your system compartmentalized from other parts of the system. Keep the system up to date, by relentlessly refactoring locally.

- **Keep it Working**

  The software is important; your customers, workers, and indeed your money depends on it. Necessary improvements are desired, but not done for fear of breaking the system. Everyone from the top down is either afraid of having the code touched or touching it themselves. Very often the resulting code is throwaway, returning us to the throwaway code issues we just explored.

- **Sweeping it under the rug**

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

If you can't make the dirt go away, you can hide it from plain sight. Unrealistic deadlines, insufficient requirements, and the feeling that just a little more hidden, dirty code won't hurt anything and nobody will notice anyway right? It's usually thought that this type of code is cost prohibitive, but the cost of maintaining such a codebase should not be ignored.

HOW CAN YOU AVOID THE BALL OF MUD?

- **Control fragmentation with continuous integration**

  When large teams work on a domain (even a team of 3 may be considered large), there is a potential for fragmentation due to the divergence of ideas within the team. Since the domain is continuously *discovered*, different developers may come up with different and divergent ideas. Why not solve this problem by further breaking down the domain? This is not the way to solve the problem. The domain is broken down to a level that still maps to a real life domain problem and when attempts are made to break it down further and artificially, that domain loses cohesion. So what to do?

  Continuous integration to the rescue. Developers working on any given domain should come together at least daily, merging their code and ideas and most importantly keeping the ubiquitous language document updated with their evolved ideas. When code is merged often, emerged divergence is quickly guarded against and removed. As you can see, continuous integration allows close collaboration within the team.

- **Avoid anemic domain models**

  An anemic domain model is one that has not been thoroughly thought through in terms of domain-driven design and is an anti-pattern. This type of domain model at first glance might appear to seem reasonable, and map somewhat into real life objects, but when examined closely, it has no real behavior. An example of an anemic domain object would resemble a data structure with getters and setters rather than some entity with behavior and complex characteristics. These models do not fully benefit from object-oriented design in that the data and behavior is not encapsulated together, it's more of a half step, and really matches a procedural style more than object oriented.

- **Design shearing layers**

  A ball of mud can be mitigated or prevented entirely by designing *shearing layers*. If we look at a building as an analogy to software, there is an argument that there isn't anything that is really a building, but many layers of components such as foundation, wiring, roof, rafters, walls, etc. These components have their own rates of change and their own lifespans. This implies that software components should be grouped according to their own rates of change and lifespan. This applies to discrete areas of domain as well as areas of abstractions because abstractions change much slower than most other logic and should therefore exist in and of itself and be maintained

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

separately. A big ball of mud is the attempt to build the building without regards to layering.

- **Reconstruction**

  The only real cure for a ball of mud is *reconstruction*. Your system has declined to the point that it has unfortunately become a big ball of mud. Sometimes it is best to just throw it all away and start over. This is always a hard pill to swallow and is required by:

- Obsolescence of the tools and technology
- Long absence of original maintainers
- Real requirements emerged over time during the building of the throwaway system
- Such drastic design change as to render the original architectural assumptions useless

ARCHITECT'S MOST USEFUL TOOLS

"An architect's most useful tools are an eraser at the drafting board, and a wrecking ball at the site.
- Frank Lloyd Wright [13]

Now that we've seen the problem with improper domain design, let's take a closer look at the right way to design your domains in a domain-driven way, so your domain "cityscape" ends up looking like the one in Figure 5.2 below, starting with bounded context.

Figure 5.2 Utopian Cityscape (http://www.theatlantic.com/international/archive/2011/04/sustainable-cities-what-makes-urban-areas-around-the-world-successful/237668/)

[13] http://www.laputan.org/mud/mud.html #reconstruction

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

5.1.2 Bounded Context

A *bounded context* describes a discrete area of functionality or a domain within a domain. You begin modeling a domain by dividing up major areas of the application into a set of bounded contexts.

An example of this approach would be the modeling of aircraft and airports; let's call this the Flight domain. Rather than attempting to model everything together in a large, cross-contaminated mishmosh, the an application built for the flight domain would be divided into three major areas of behavior:

- Aircraft, with the concern of flying,
- Tower, concerned with multiple aircraft arriving, departing, and in the air, and finally
- Ground control, which concerns itself with moving multiple aircraft and vehicles around the airport.

The airport is implicit in this example, as no behavior has been identified around the airport as a whole, although it may be defined in the *ubiquitous language*, a set of terms shared and understood by both the domain experts and the developers, which we'll explain in the next subsection.[14] There are a great many

sub-domains that may be identified as part of an airport, but for this example we'll keep it simple and work with the three defined above. A bounded context is a DDD term describing a sub-domain; in this case there are three contexts or sub-domains: the aircraft, the tower, and ground control, each having very different concerns.

The aircraft is only concerned about flying, safety, weather and the mechanics of flight. The aircraft does communicate with external bounded contexts, such a control tower, but would do so in the simplest manner necessary to carry on the business of flying. The inner workings of the how and why of control would be left to the control tower bounded context. If communication is required between bounded contexts, such as the case of a control tower contacting the aircraft by radio instructing the plane to descend 2000 feet, in our system this would translate to the control tower context communicating some message it understands to the aircraft context.

You may now be seeing the connection between our flight domain and modeling any complex domain in the business world. You can see how we have broken up the complex business of flight into digestible pieces, and in doing so we naturally attained making the domain more distributed as well, because the domain can be split into three completely independent pieces:

- aircraft,

[14] To attempt to define the airport as an *aggregate root*, or container of domain functionality as we'll discuss very soon, would result in combining the ground control and tower contexts, this is undesirable as these are separate concerns and the domain models would clash and cause confusion such as ground control's treatment of a plane as just another vehicle whereas the tower concerns itself with aircraft.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

- tower, and
- ground control.

Each piece can exist at runtime on separate hardware and across different physical locations.

ANTI-CORRUPTION LAYER

As we mentioned above, the aircraft context or sub-domain does communicate with the other contexts: tower and ground control, and when it does, it uses what DDD calls an anti-corruption layer to translate the incoming messages. An *anti-corruption* layer is an outer layer that sits inside a bounded context to convert and validate data to and from other contexts and external systems. For example, a command might be sent by an external party for a reduction of altitude by 2000 feet, it's up to the aircraft context to determine if this command is reasonable and will not drive the craft into the ground.

Just as the aircraft context is concerned with the complex workings of flight, so the control tower and ground control contexts contain their own complexities around controlling aircraft and vehicles in all aspects of flight as well as on the ground. To help discover and identify our domains we use a bounded contexts diagram. Figure 5.3 illustrates the use of a bounded-context diagram for flight.

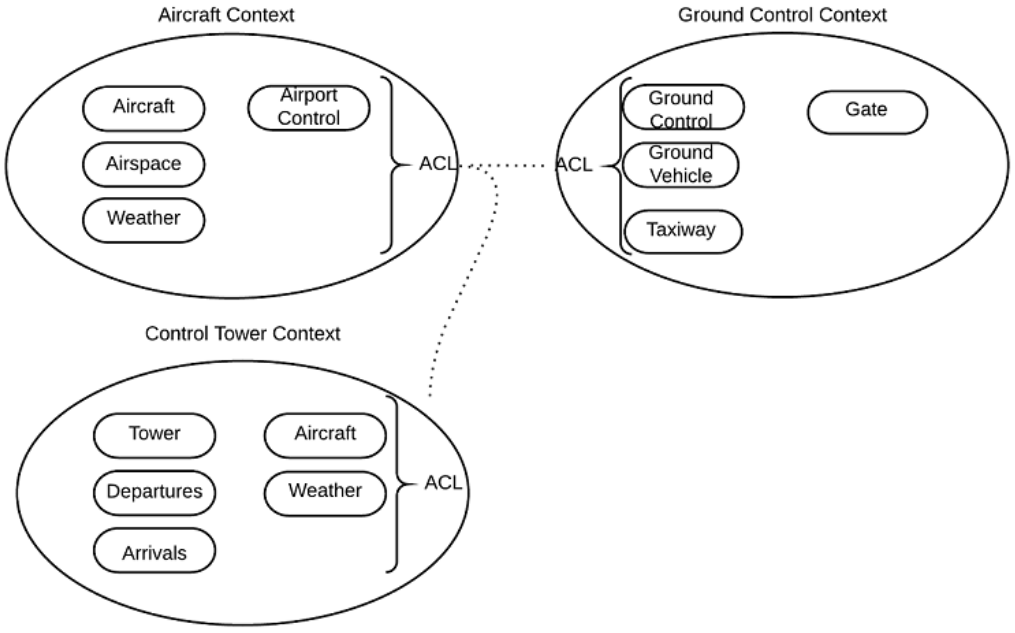

Figure 5.3 Flight domain contexts to show the entire system from a bird's eye view

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

You can see in the figure above that each context can be thought out and developed in complete isolation from the other contexts, and eventually hooked together through their anti-corruption layers, denoted with *ACL*. Development of such subdomains is much simpler to tackle than the entire thing at once.

You can already get a glimpse of how domain driven design has made a large application more distribution friendly: the overall domain is completely decoupled into various bounded contexts, which can operate independently and in different application spaces, only communicating through messages with the outside world.

Next, let's look in greater detail at the ubiquitous language, which is effectively a glossary of domain terminology that serves as a reference for the bounded contexts.

5.1.3 Ubiquitous language

The use of the terms "aircraft," "vehicle," "ground control," "departures," etc. represent the use of *ubiquitous language*, which is the software terminology matching the real-life domain, and shared by the development team and the domain experts. Figure 5.3 illustrates this design.

The ubiquitous language is a set of terms describing all aspects of the domain that are understood by both the domain experts and the developers. This language is expressed in a living document that should be created as early as possible and continuously referred to by functional and technical teams. The ubiquitous language facilitates discussions with the domain experts and discovers key domain concepts used in the domain model. The figure below illustrates a partial document describing the ubiquitous language of the overall flight domain from our example.

- **Aircraft - Usually an airplane, but not necessarily so. Aircraft fly to and from destinations and must deal with external control within airspace and airports, but must also be concerned with weather and the operation and health of the aircraft and those aboard.**
- **Airspace - A geographical area in which an aircraft is under control of a tower.**
- **Arrivals - The area of the tower primarily concerned with aircraft arriving into the airport.**
- **Departures - The area of the tower primarily concerned with aircraft taking off from the airport.**
- **Ground Control - This operation controls and moves aircraft and other vehicles around the tarmac as well as to and from gates.**
- **Tower - The control tower controls aircraft during departure, arrival, and within its airspace.**
- **Weather - Weather in a concern everywhere, from the aircraft in the air as well as all vehicles on the ground, including plows and snow melters.**

Figure 5.4 Ubiquitous language document clearly defining flight domain terminology

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

With the terminology firmly in hand, thanks to our ubiquitous language document, it's time to design our domain entities: the building blocks of the domain that map to the document above, as well as real life. These building blocks are:

- *Entities, which have identity and are persistent objects, and*
  - can take the form of *Aggregates,* in order to encapsulate other entities
  - and utilize *Value Objects.*

We'll explain each of these in turn, next.

5.1.4 Entities, Aggregates and Value Objects

Entities in domain driven design are any objects representing a meaningful area of the domain, and most importantly have identity. Each entity's identity is always unique among all other entities in the entire domain. This uniqueness is established by the attributes within the entities, which can never be completely duplicated. An example of such an entity might be a person: a person can have the same first and last names but never the same social security number.

Sometimes unique ids are a fabricated thing, such as a generated id at the time of an entity's creation, but often some meaningful attribute is naturally unique such as a customer id or serial number. In our flight domain example, the *id* attribute uniquely identifies aircrafts. The id would never be reused for another aircraft, and allows the reuse of the callsign for other flights. In practice, we ourselves have always utilized a generated id for all domain entities for consistency's sake, as well as the ease of use within a framework. For example, in your framework somewhere you handle some common functionality around creating new entities and want to guard against duplicates; when all entities consistently use id, this becomes simple. This does not however prevent duplicate business keys. For example, in an employee domain each employee has a unique id generated but employees also have a social security number that must be unique. This uniqueness must be maintained somewhere in the framework, such as a service layer. See the services section below.

Entities are named in some meaningful way and map directly to the ubiquitous language; they are not defined by their attributes, but have real-life meaning in and of themselves, i.e. you don't need to understand the inner behaviors of an aircraft to know what an aircraft is. In Figure 5.5 below, only the aircraft and passengers are entities.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

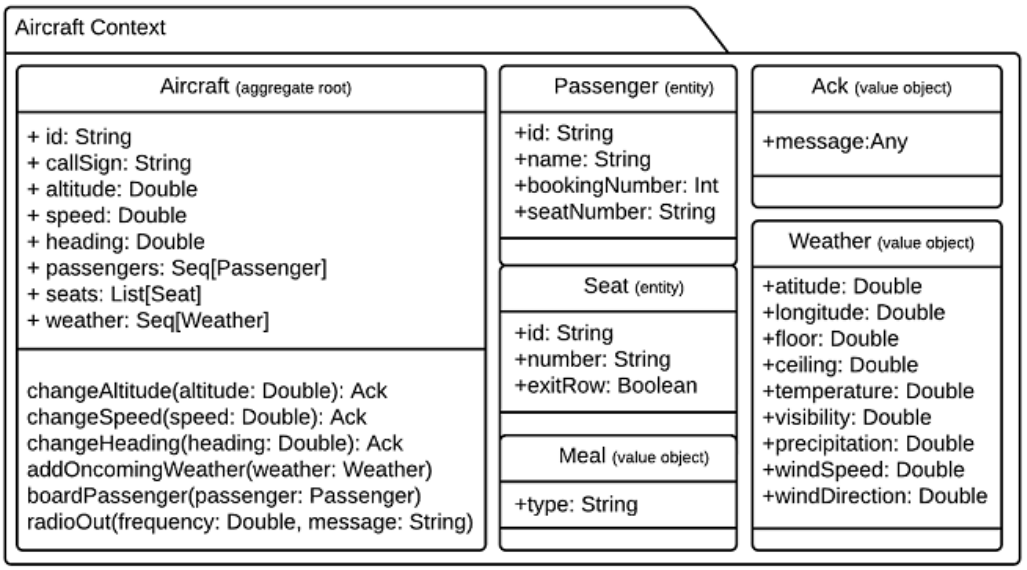

Figure 5.5 Example Aggregate with Value Objects

Above we can see that the ack, meals and weather are value objects, and as such have no identity as we'll explain below. The aircraft, passengers and seats are all entities and have unique identity as well as persistent state. Since the aircraft encapsulates the other entities and value objects it is considered an *aggregate root*.

AGGREGATE ROOTS

Aggregates roots, also known as *aggregates,* are domain entities that represent the top of a hierarchy of two or more entities. It is highly recommended to identify your aggregated roots as single entities and not to try to immediately identify an entity hierarchy within. This falls under the classification of premature optimization, which usually considered a bad design choice. There hierarchies will naturally present themselves in due course as your design evolves. Let's assume here that when we were designing the aircraft we did so knowing that we would need some domain object to model the aircraft behavior in the sky and on the ground and later determined that the passengers affect that behavior due to the overall weight of the aircraft, etc. The passenger entity is then identified and determined to be part of the aircraft aggregate. Aggregate roots map directly to the ubiquitous language, such as the term *aircraft*. Aggregates are entities first and as such of course carry a unique identity. Access to any encapsulated entities such as passengers, would only be through the aggregate. Another example of an aggregate root might be a car. The car would be made up of many entities contained within, such as the engine, electrical system, chassis, interior, etc. but none of those would exist in any capacity outside the car, although they might be modeled in other

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

domains differently, for example an assembly line domain could also have the notion of an engine but with different meaning and behavior.

In our example bounded context diagram in the figure above, the aircraft is the only aggregate root, encapsulating domain behavior. The aircraft aggregate has various states and many complex capabilities. In fact an aggregate root can be considered to be a bounded context in itself. The aircraft aggregate encapsulates the many areas of behavior it contains and is the root access to all that it holds.

STRONG CONSISTENCY WITHIN AGGREGATES

Aggregates are always strongly consistent with regard to their encapsulated data, in that any change to the aggregate is guaranteed to reflect the very latest state of all data within. For example, the addition of a passenger to the plane would result in a new version of the aggregate, including the very latest speed and heading of the aircraft.

Interactions between aggregates are always eventually consistent, i.e. an Aircraft will eventually receive a radio message from a tower, but the state of the tower may be changing before the aircraft even has a chance to respond. This in no way hampers the plane from making its maneuver and eventually the aircraft will get more instructions.

VALUE OBJECTS

In our flight domain the ack, meals and weather are considered value objects. A *value object* represents a meaningful construct within the domain and also maps to the ubiquitous language, but does not have the concept of identity and therefore is not unique. It could be possible that the aircraft has a collection of completely duplicated meals, which is usually the case and one of those meals is considered completely equivalent to another without breaking DDD rules for value objects. These value objects would never exist in an of themselves but only in the context of the encapsulating aircraft.

As we mentioned, the meals on the aircraft are value objects. Each meal has no identity or behavior in its own right, and are always completely immutable in that they are created once and never updated. One meal is distinguishable from another only by comparison to another by its type, chicken vs beef because there is no identity. In short we don't care much about one meal vs another, except for the ability to delivery chicken type meals to passengers wanting chicken and beef to the others. Classification does not equal identity!

An aggregate, its entities and its value objects all have consistent state and are stored as a single persistent unit. In the figure you can see the aircraft aggregate modeled with some basic characteristics, including the collections of the passenger and seat entities, as well as meal and weather value objects.

- The **ack** (*acknowledge)* is a value object at its most simplistic; it is a data structure designed to carry a radio message from the aircraft to the tower and has no behavior or identity, but it maps to the ubiquitous language and is therefore a value object, even though it is not contained within airplane as any attribute or collection.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

- A **passenger** would never exist outside of this aircraft (in our model). Passengers may be added to an aircraft, but only by going through the aircraft aggregate; the passengers would never be accessed directly.

Next we'll look at the concept of services, repositories, and factories, which are best described as all the rest of the application's DDD functionality that is not contained within aggregates.

5.1.5 Services, Repositories and Factories

Services are best described as processes that don't "belong" to any domain entity, but may operate on those entities. A service operates on one or many aggregates at once from the outside, in order to achieve some overall goal. You might have noticed that we did not find it necessary to employ services in our flight domain. Most times in our experience you will find that all of your domain logic correctly lives inside your domain aggregates, which is a common object oriented encapsulation technique. This is not to say that services are wrong; they are valuable when necessary, but we recommend they are used sparingly as a rule. The reason for this: developers over time have made services the "go to" area for functionality, when they should have placed that functionality within aggregates. Services are often used for functionality when developers are unfamiliar with aggregates and domain-driven design. A helpful rule of thumb is if it is a behavior of a thing, include in in an aggregate, if not, use a service.

An example where we *have* needed a service is a situation we had in an energy optimization application. We needed to reach out to various buildings and temperature zones to pull real-time temperature readings and associate those readings with buildings, which we modeled as aggregates. Figure 5.6 shows how a building reading service might operate to retrieve all building readings each minute, then send them to each building aggregate.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

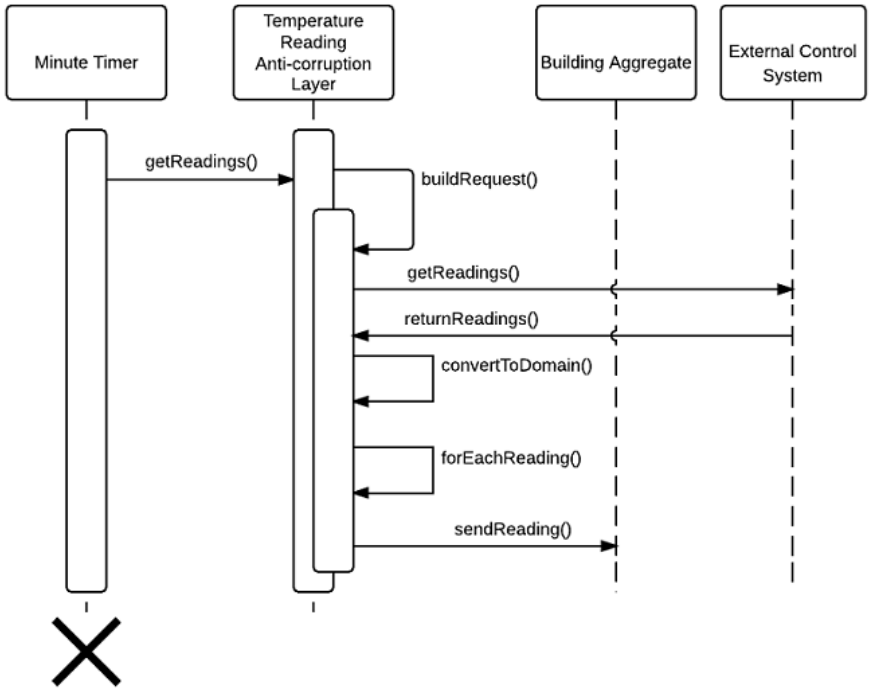

Figure 5.6 Building reading domain service to operate on many building entities at once

As the figure shows, the design dictated we make a request to the outside building control service each minute for all readings, across all buildings. Obviously this functionality could not then be within any given building, and therefore we needed to do it in a service. As it happened, this service was also an anti-corruption layer, which as we explained earlier, is a protective and communicative layer between our domain and the outside world. The reason we used an anti-corruption layer here is that the services needed to call out to an external building control system for data, that data needing validating and transformation before being allowed to enter our domain.

Repositories are similar to services, because they are also outside the domain, but they are used to retrieve and instantiate domain objects from the data store.

REPOSITORIES AND FACTORIES

Repositories are the way you can "get at" your domain aggregates. Aggregates are persistent and DDD applications are long-lived, meaning the systems will start and stop many times; it is the data that drives the system and gives in ongoing life. Imagine a scenario in which flight 300 has been created in the system and is in the air, and the system is shut down. When the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

system restarts, there must be some facility to instantiate domain aggregates from the database in order to start working with them again. This is exactly what a repository does.

Typical repository behavior might include:

- a `get` of a single domain aggregate by id,
- retrieval of all aggregates, or
- some sort of find operation.

In the Scala programming language, putting aside the concept of actors for now, it's best and most expressive to implement repositories as companion objects.

- A *companion object* is a singleton with the same name as an implementation class and with special access to private functionality in that class, and is commonly used as factories.
- A *factory* is a way of creating objects and is very close in behavior to our repository pattern, but differs because the repository returns aggregates that already exist in some persisted state, whereas the factory is only used for the initial creation.

The very idea of flight 300 is first conceived by utilizing the create factory function in the *Aircraft* companion object. You'll notice in the code listing below that we are able to collapse the behaviors of both repository and factory in the aircraft companion object, rather than create a separate object, such as *AircraftRepository*, though it would be ok to do it that way as well. For convenience we'll just put everything in a single object.

In the following code example, the *create* is the factory function, and all others are repository functions. You'll notice that the aircraft only needs a minimum amount of data in order to be created: the unique id and the callsign. Listing 5.1 shows how the repository/factory would normally be modeled in many languages; shown using plain Scala here.

Listing 5.1 Aircraft repository and factory companion object

```
object Aircraft {

  def create(id: String, callsign: String): Aircraft = ... ❶

  def get(id: String): Aircraft = ... ❷

  def getAll(): List[Aircraft] = ...  ❸

  def findByCallsign(callsign: String): Aircraft = ...      ❹
}

case class Aircraft (
  id: String,
  callsign: String,
  altitude: Double,
  speed: Double,
  heading: Double,
  passengers: List[Passenger],
  weather: List[Weather]
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

```
)
```

1. The create factory function
2. The get repository function issues a database operation to retrieve a single aircraft
3. The getAll repository function issues a database operation to retrieve all aircrafts
4. The findByCallsign repository function issues a database operation to find an aircraft by callsign

The code block above shows how an aircraft repository and factory would traditionally be modeled, we'll look at an alternative, message-driven way, building on what you learned in chapter 2 with Akka actors, in section 4.2.

At various points in time, flight 300 must communicate with the outside world; the flight needs to know about incoming weather and traffic control on both the ground and in the air. To do this we use an anti-corruption layer, which we briefly covered earlier when describing the aircraft's communication with tower and ground control, and which we'll now explain in more detail.

5.1.6 Anti-corruption Layers

Anti-corruptions layers sit atop a bounded context and convert communications and representational data to and from outside systems. This allows a level of purity within the main part of the bounded context that knows nothing of the outside system's requirements or its inner workings. In turn, the outside system has no knowledge of the inner workings of the bounded context. Maintaining purity in is important because garbage into the domain will result in garbage out or in short: corruption.

The anti-corruption layer, shown below, translates inbound external logic to constructs understandable to the inner domain, in this case external weather data is validated and converted to the domain representation of weather.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

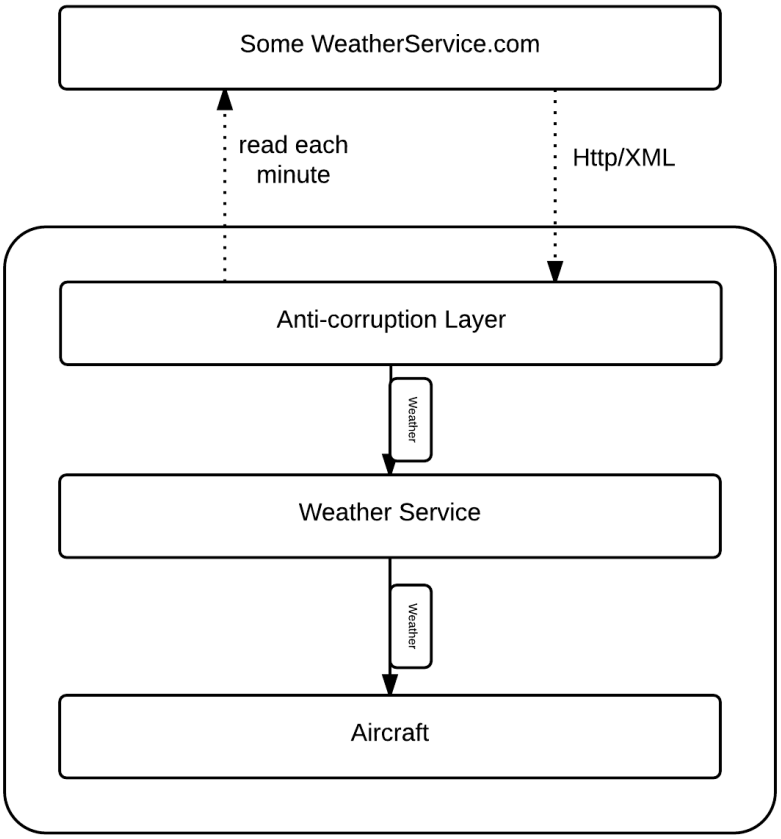

Figure 5.7 Weather anti-corruption translating incoming XML to domain *weather*

Conversely, the anti-corruption layer translates domain logic for the outbound flows to the external constructs when sending data to external contexts or parties. The weather is requested each minute by the weather service within the domain and arrives over HTTP in an XML format known and dictated by the external weather provider, *Some WeatherService.com*. The anti-corruption layer is is knowledgeable of both the format of the XML coming in and of course the Weather domain value object within. This layer will not allow bad or invalid data to enter the domain.

A use case relating to the figure above would be the receipt aircraft of new weather by the aircraft for airspace it's about to fly through. As the figure shows, The weather comes in to the aircraft's anti-corruption layer in some format, let's say XML. The anti-corruption layer takes this XML format and transforms it into the aircraft's idea of weather, which is the Weather value object contained in the aircraft context. As you can see in the figure, the aircraft context

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

utilizes an anti-corruption layer to communicate with an external weather service. The anti-corruption layer accepts incoming weather in the XML format of the weather service provider and translates that Weather value objects.

Next, we'll wrap up domain-driven design by looking at state transitioning and layered architectures, and then finishing with the valuable *Saga* pattern for modeling long-lived domain transactions.

STATE TRANSITION

In additional to a typical anti-corruption layer, anti-corruption can be augmented by domain state transitioning. An aircraft on the ground has different behaviors from an aircraft in the air. Requesting an aircraft on the taxiway to bank left is nonsensical and therefore having a function in place to take that command is also nonsensical. To express this we support the aircraft in two flavors:

- Aircraft and
- GroundedAircraft.

Each command issued to a grounded aircraft would result in an updated instance of the aircraft in ground mode. Only upon a successful takeoff command would that grounded aircraft morph into an aircraft in the air and at that point all of the commands having to do with flying would be supported.

There are various ways to achieve this in terms of implementation, and we won't exhaust them here, but pay special attention to the become/unbecome section we looked at in Chapter 3, because that is a very handy way to model this. With an actor domain implementation, the actor representing the aircraft "becomes" a handler of a different set of messages for each state it handles and is a very elegant solution.

5.1.7 Layered Architecture

It is common in muddy designs to mix UI, database and other non-domain logic with the domain itself. Applications should be built in layers, separating the concerns as much as possible. It should be very easy to look at a domain object and quickly understand the behavior of that object. As other code is mixed in, it becomes hard to separate the wheat from the chaff, making maintenance of the code difficult as well as putting unnecessary barriers in place that make it hard for new staff and domain experts to understand it all.

Using UI as an example, changes would cause possible changes and corruption to domain logic, because the UI person may not have a good enough understanding of the domain, and could break the domain rather than fulfill the UI change requirement.

In practice, be sure to isolate domain logic from all application underpinnings so that the domain model is as expressive and apparent as possible.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

5.1.8 Sagas

A *saga* is a long-running business process, but don't get fooled by the words "long running," as a saga might complete its job in an instant. Sagas are not strictly considered part of domain-driven design, but are an important pattern you'll likely find necessary in your reactive applications. Sagas are not domain entities, but may operate upon them and have no sort of domain identity, although they might carry a transaction id while they do their job, but what is that job?

A good example of a business saga is a bank transfer. A bank transfer cannot take place solely within a single bank account, and one would not want to tie up a bank domain entity to wait for funds and acknowledgments for a single customer's transfer.

For this situation, we would create a saga to model the bank transfer process from start to finish. The bank transfer saga would have a number of steps, or more accurately states.

1. Withdraw from the *from* account, and await the acknowledgement
2. Deposit to the destination account and await the acknowledgement
3. Notify the customer of the completed transfer
4. Kill itself

There are as many ways to model sagas as there are computing languages, and later in this chapter you'll see how it's done the Akka way.

5.1.9 Shared Kernel

This is a shared context among domain teams that contains shared code in order to be DRY (don't repeat yourself), and should be handled with care. This kernel should be kept as small as possible and not allowed to bloat with premature optimizations. Addition of any code into this context should only be done in close collaboration among all teams, because breakage here would have a ripple effect.

Shared kernels typically take the form of libraries named *core* or *common* and may easily lead to code smell so tread carefully.

RATCHETING IT UP A NOTCH

You now have a good grasp of domain-driven design and how much of a helpful tool it is in designing software that is simplest to reason about and distribute: reactive friendly. Now we'll show new ways (compared to a traditional Java or Scala class with getters and setters) of implementing the flight domain to make it more reactive friendly, by applying what you've learned about Akka in chapter 2 and 3.

We're going to stop short of implementing any sort or storage mechanism to persist the domain to a database, and leave that as implied for now as there will be a deep dive on that subject in chapter 5 as we combine the domain-driven design patterns with Command Query Responsibility Segregation and Event Sourcing to enable your application to be completely reactive.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

5.2 An Actor-based domain

We'll design a rudimentary Akka actor-based domain model, which, when when coupled with Akka clustering and persistence as you'll see in later chapters, will allow unbounded elasticity, as well as resilience, responsiveness, and to be message driven.

5.2.1 A Simple Akka Aircraft Domain

In the listing below we lay the groundwork for with a Scala case class, like the ones we applied in chapter 2 (Listing 3.3), representing an aircraft state for our flight domain. This case class is is a construct we will use strictly for message passing. The aircraft protocol includes all the messages that are handled by an aircraft actor, which we will model a just a bit later.

Listing 5.2 The Aircraft Case Class and Protocol

```
final case class Aircraft ( ❶
  id: String,
  callsign: String,
  altitude: Double,
  speed: Double,
  heading: Double,
  passengers: List[Passenger],
  weather: List[Weather]
)

object AircraftProtocol {    ❷
  sealed trait AircraftProtocolMessage ❸
  final case class ChangeAltitude(altitude: Double) extends AircraftProtocolMessage ❹
  final case class ChangeSpeed(speed: Double) extends AircraftProtocolMessage
  final case class ChangeHeading(heading: Double) extends AircraftProtocolMessage
  final case class BoardPassenger(passenger: Passenger) extends AircraftProtocolMessage
  final case class AddWeather(weather: Weather) extends AircraftProtocolMessage
  final case object Ok
}
```

❶ **The aircraft class has no behavior now, it is just reflects current state**
❷ **The messaging protocol for an aircraft**
❸ **It is best practice to seal the messages, which will result in Scala match errors when a message is not implemented**
❹ **These messages are immutable and will not result in the direct return of any data**

WHERE IS THE BEHAVIOR?

The code above lays the groundwork for interaction with an aircraft but where is the behavior? The behavior is all encapsulated with an actor, as we'll show next when we wrap the aircraft domain functionality within an actor. We talked about layered architectures earlier, which dictate that the domain logic should stand out as much as possible, and there are some camps that desire a full domain object with functions and attributes.

The argument here is that we're doing the same thing, but with a slight paradigm shift.

- Instead of functions named *changeAltitude()* or *changeSpeed(),* we use message

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

handlers within the actor.

- We also contend that modeling out the domain independent of actors, and then wrapping that domain within the actor is a waste of time, because we consider actors an extension of either Scala or Java, and as such we assume there will be no other implementation of *aircraft* in this architecture. This is something we are willing to live with and bears mentioning, but follow your own heart and do what feels right in your own designs.

The following listing will show an aircraft actor that communicates using the aircraft protocol for messaging. You'll see that because this is an actor, each message is a one-way communication, and each message is handled in the order received. The actor stores current state in memory, and each message received results in a modified replacement of that state.

5.2.2 The actor

The Akka actor will encapsulate all behavior of the aircraft and will do so with messaging rather than traditional function calls.

- Each processing of a message results in the new state of the aircraft within the actor.
- The sender is simply sent an OK reply upon processing of the message, but
- that reply may be expanded upon to return either the OK, or a list of validation errors or similar.

We'll keep it simple here.

Listing 5.3 The Aircraft Actor

```
import akka.Actor

class AircraftActor(          ❶
  id: String,
  callsign: String,
  altitude: Double,
  speed: Double,
  heading: Double,
  passengers: List[Passenger],
  weather: List[Weather]
) extends Actor (

  import AircraftProtocol._ ❷

  var currentState: Aircraft = Aircraft(id, callsign, altitude, speed, heading, passengers,
       weather)               ❸

  override def receive = {
    case ChangeAltitude(altitude)    =>
      currentState = currentState.copy(altitude = altitude) ❹
      sender() ! OK            ❺

    case ChangeSpeed(speed)          =>
      currentState.copy(speed = speed)
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

```
      sender() ! OK

    case ChangeHeading(heading)      =>
      currentState = currentState.copy(heading = heading)
      sender() ! OK

    case BoardPassenger(passenger)   =>
      currentState = currentState.copy(passengers = passenger :: passengers)
      sender() ! OK

    case AddWeather(incomingWeather) =>
      currentState = currentState.copy(weather = incomingWeather :: weather)
      sender() ! OK
  }
```

1. The actor constructor arguments initialize the current state; may be used in creation or read from a database
2. The protocol is brought into scope
3. A var is ok here and can only be accessed by the actor itself when instantiated
4. The current state is updated to contain the new value
5. The reply of OK so the sender of the message knows the message was processed

The example above is completely thread safe, but not really free of side effects. The problem here is even though one atomic update of the aircraft can occur at any given time, an updater may be making that update based upon stale state. This may be guarded against using *versioning*, a method of ensuring domain consistency as we'll show next.

5.2.3 The Process Manager

We talked about the saga pattern earlier in a functional way, so let's look at an implementation using an actor approach. We'll implement the bank transfer example we discussed earlier in section 4.1.8. We won't go as far as modeling the accounts; just assume each bank account is represented by an actor that accepts messages using an account protocol. We take the liberty of simply calling the actor the *BankTransferProcess,* and drop the word manager for brevity.

The next listing shows the account and process manager protocols and companion objects.

Listing 5.4 The Process Manager and Account Protocols and Companion Objects

```
import akka.actor.{ReceiveTimeout, Actor, ActorRef}
import scala.concurrent.duration._

object BankTransferProcessProtocol { (1)
  sealed trait BankTransferProcessMessage

  final case class TransferFunds(
    transactionId: String,
    fromAccount: ActorRef,
    toAccount: ActorRef,
    amount: Double) extends BankTransferProcessMessage
}

object BankTransferProcess {          (2)
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

```
  final case class FundsTransfered(transactionId: String)
  final case class TransferFailed(transactionId: String)
  final case object Acknowledgment
}

object AccountProtocol {                    ❸
  sealed trait AccountProtocolMessage
  final case class Withdraw(amount: Double) extends AccountProtocolMessage
  final case class Deposit(amount: Double) extends AccountProtocolMessage
}
```

❶ The bank transfer process manager protocol
❷ The companion object with positive and negative acknowledgment messages
❸ The account protocol

In the next listing, we'll utilize the objects we just created in the account transfer process manager actor.

Listing 5.5 The Process Manager Actor

```
class BankTransferProcess extends Actor {

  import BankTransferProcess._
  import BankTransferProcessProtocol._
  import AccountProtocol._

  context.setReceiveTimeout(30.minutes) ❶

  override def receive = {                  ❷
    case TransferFunds(transactionId, fromAccount, toAccount, amount) =>
      fromAccount ! Withdraw(amount)
      val client = sender()
      context become awaitWithdrawal(transactionId, amount, toAccount, client)
  }

  def awaitWithdrawal(transactionId: String, amount: Double, toAccount: ActorRef, client:
       ActorRef): Receive = {               ❸
    case Acknowledgment =>
      toAccount ! Deposit(amount)
      context become awaitDeposit(transactionId, client)

    case ReceiveTimeout =>
      client ! TransferFailed(transactionId)
      context.stop(self)
  }

  def awaitDeposit(transactionId: String, client: ActorRef): Receive = { ❹
    case Acknowledgment =>
      client ! FundsTransfered(transactionId)
      context.stop(self)                    ❺

    case ReceiveTimeout =>
      client ! TransferFailed(transactionId)
      context.stop(self)
  }

}
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

1. **The receive timeout will allow a total of 30 minutes for any step in the process**
2. **The initial request to transfer funds includes all information necessary to do the job, including the sender actor reference, which we copy here to *client* in order to enable the reply to the initial requestor of the transfer across receive boundaries.**
3. **Here we await the withdrawal or fail the transfer on a receive timeout message (included in Akka framework)**
4. **Here we await the deposit or fail the transfer on a receive timeout**
5. **Finally the process self destructs**

The process above babysits the entire transfer from start to finish, over a limited amount of time. At any point failure is always a possibility, due to inability to access an account, insufficient balance, etc. When this happens the client will be sent the "transfer failed" message in order to perform some compensating action or retry the transfer.

5.3 Summary

In this chapter you have learned:

- How to divide and conquer a domain using domain driven design
- To define an ever evolving ubiquitous language to describe a domain
- The concepts of entities, aggregate roots and value objects, which form the building blocks of your domain
- When and how to utilize services, repositories and factories to work with your domain entities
- How you might model aspects of an airport/flight domain
- The use of anti-corruption layers to communicate with other domains and the outside world
- The saga pattern for long running transactions
- How you might design and aircraft aggregate as an Akka actor
- How you would implement a saga using the process manager pattern.

When domain-driven design is used in combination with Akka and *Command Query Responsibility Segregation and Event Sourcing (*CQRS-ES), a set of concepts that dictates the separation of read and write application concerns and the persistence of events rather than state, which we'll cover next in chapter 5, you'll have all the tools necessary to build successful, reactive applications.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

6

Using Remote Actors

This chapter covers

- **Structuring reactive applications using sbt**
- **Configuring Akka with** `application.conf`
- **Remoting with Akka**
- **Reliability guarantees in distributed systems**

Historically, a distributed system is defined as a software system where components that reside on networked computers work together on a common goal. While this definition, albeit somewhat generalized, captures the gist, it has led to some unfortunate side effects. The primary is the belief that the programming model used for a remote object can be generalized to match that of a local one. In a distributed system, remote objects require different latency metrics, memory access models, concurrency constructs, and failure handling. The local model cannot be generalized into a distributed model. Rather, you need treat them separately and optimize both for their intended environment. For more on this subject, see the classic paper *A Note on Distributed Computing* (Sun Microsystems, 1994), http://dl.acm.org/citation.cfm?id=974938, which explains why the generalization approach is bound to fail.

6.1 Refactoring to a distributed system

Akka's philosophy is one of distribution by default. Through its toolkit and runtime design, all functionality is equally available whether on a single JVM or a cluster of machines. The API for elasticity and scalability, whether horizontal or vertical, is one and the same.

Akka uses optimization instead of generalization. It provides the semantics for distributed interaction translated into a common API. Hard problems are managed under the covers,

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

leaving you to focus on your application. The result is that the system can run in a local or distributed environment with almost no change to the code.If you are new to distributed computing, it may not be apparent how difficult this is.

6.1.1 Splitting into multiple actor systems

In chapters 3 and 4, you built an actor system with a simple supervision hierarchy, as shown in figure 6.1. In that hierarchy, the `customer` and the `rarebooks` actors both are supervised by the `user` actor, which is supplied automatically by Akka. The two actors you created represent separate ideas that each contain state and identity. Each could run isolated and independent of the other, providing elasticity and scalability.

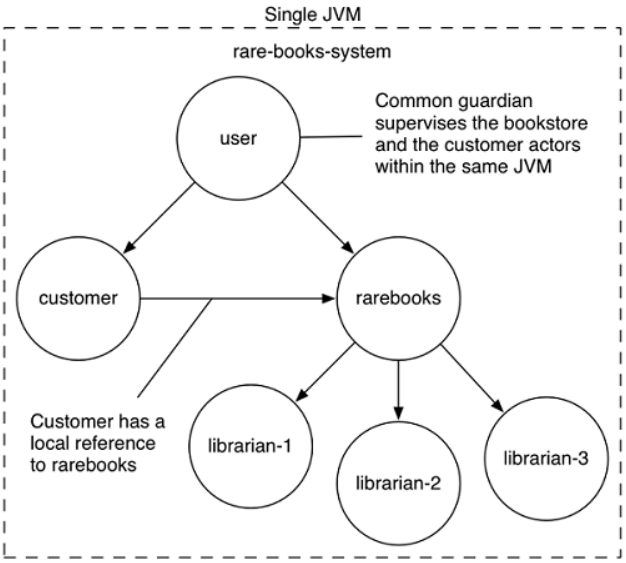

Figure 6.1 Initial state of the RareBooks application

To transform this into a distributed system that runs in multiple JVMs, a few changes are necessary, as shown in figure 6.2. The reference from the `customer` actor to the `rarebooks` actor will be a remote reference rather than a local reference. Akka uses a common API and is distributed by default, so the remote reference has the same type as the local reference, an `ActorRef`. Instead of requiring code changes, distributed actors require configuration changes. You will learn more about remote references in section 6.3.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

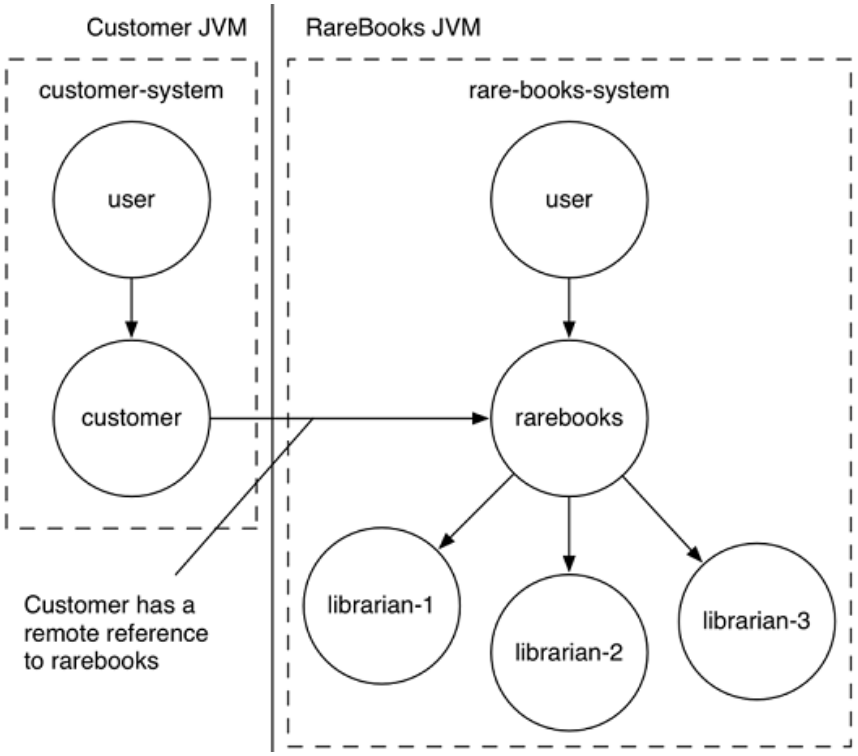

Figure 6.2 Target state of the RareBooks application

The example system needs a bit of refactoring before it is ready for remoting. Each actor system will run in a separate JVM. Each JVM in turn requires its own application containing a `main()` driver.

6.1.2 Structuring with SBT

The first time you split a single actor system into two, which was back in section 2.4, the actor systems were left together in a single project and the one project had two `main()` drivers. That becomes unmanageable in real-world projects. Before making additional code changes, you will refactor the project to match the domain, as shown in figure 6.3.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

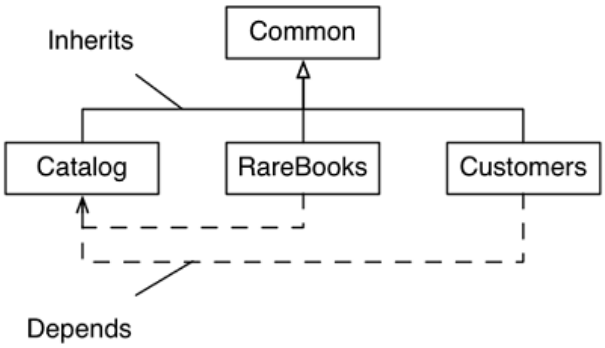

Figure 6.3 Refactoring the RareBooks application into a parent SBT project and three subprojects

SBT makes it easy to define the dependencies. Start by defining the common parent project that declares the three subprojects. A simplified version follows. The complete version is available in the book source repository at https://github.com/ironfish/reactive-application-development-scala.

Listing 6.1 Root build.sbt with sub projects

```
lazy val common = project                                    (1)

lazy val chapter6_001_catalog = project.dependsOn(           (2)
[CA]common % "test->test;compile->compile")                  (2)

lazy val chapter6_001_customer = project.dependsOn(          (3)
[CA]common % "test->test;compile->compile",                  (3)
[CA]chapter6_001_catalog % "compile->compile")               (3)

lazy val chapter6_001_rarebooks = project.dependsOn(         (4)
[CA]common % "test->test;compile->compile",                  (4)
[CA]chapter6_001_catalog % "compile->compile")               (4)
```

(1) Declare the common parent that is used by all the examples in this book. It contains the Akka dependencies and other build parameters
(2) Declare that the catalog is a project that depends on the common parent
(3) Declare that customer is a project that depends on the common parent and on the catalog
(4) Declare that rarebooks is a project that depends on the common parent and on the catalog

The catalog subproject is a static, compile-time dependency referenced by both the Customers and RareBooks subprojects. The actors in the Customers subproject will need a reference to the RareBooks actor. It would have been possible to handle this with another static dependency, but it is preferable not to compile the location of an actor into the code. Instead, the reference should be provided using run-time configuration.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

6.2 Configuring the applications

If you been around the JVM for a while, the notion of configuration is not new. Many frameworks like Spring and Hibernate use configuration to ease the assembly of framework objects and services in a loosely coupled fashion.

Akka configuration uses the Lightbend Config Library, which is a general library for managing configuration on the JVM. It is a robust library that can be used for Akka or any JVM-based application. Through this configuration library, Akka provides the means to establish logging, enable remoting, define routing, tune dispatching and other things, all with little or no code.

Now you just need an application to configure.

6.2.1 Creating the first driver application

The next step of our remoting exercise is to implement RareBooks as an independent application. RareBooks must be able to bootstrap independently so that it can stand on its own.

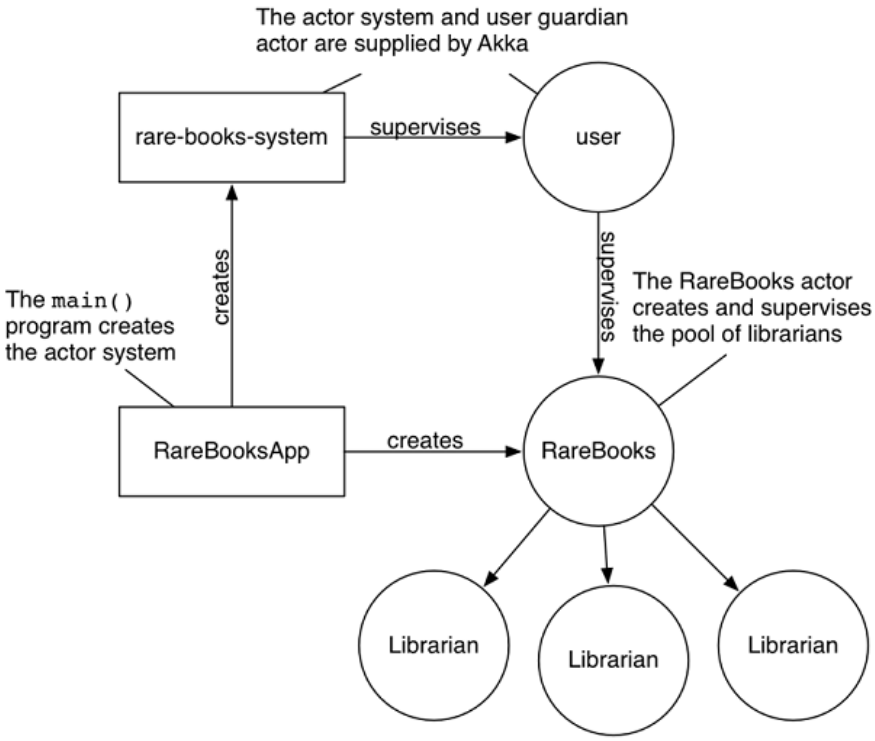

Figure 6.4 The RareBooksApp driver and RareBooks and Librarian actors

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

The standalone version of RareBooks is shown in figure 6.4. The RareBooks `main()` method is defined in the companion object and is responsible for creating the actor system, as follows:

Listing 6.3 Companion Object for RareBooksApp.scala

```
package com.rarebooks.library

... (1)

object RareBooksApp {

def main(args: Array[String]): Unit = {
    val system: ActorSystem = ActorSystem("rare-books-system") (2)
    val rareBooksApp: RareBooksApp = new RareBooksApp(system)  (3)
    rareBooksApp.run()                                         (3)
  }
}
```

(1) Imports removed
(2) Create the actor system
(3) Instantiate and run the application

The`RareBooksApp`class is a command-line application that creates the `RareBooks` actor, as follows:

Listing 6.4 RareBooksApp.scala class

```
class RareBooksApp(system: ActorSystem) extends Console {  (1)

  private val log = Logging(system, getClass.getName)
  createRareBooks()   (2)
                                                            (2)
protected def createRareBooks(): ActorRef = {              (2)
  system.actorOf(RareBooks.props, "rare-books")             (2)
  }                   (2)

  def run(): Unit = {
commandLoop()         (3)
    Await.ready(system.whenTerminated, Duration.Inf)        (4)
  }

  @tailrec
  private def commandLoop(): Unit =                         (5)
    Command(StdIn.readLine()) match {
      case Command.Customer(count, odds, tolerance) =>
commandLoop()
      case Command.Quit =>
        system.terminate()
      case Command.Unknown(command) =>
        log.warning(s"Unknown command $command")
        commandLoop()
    }
}
```

(1) Console is a common superclass used by drivers in this book

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

2. Create the RareBooks actor, which takes care of creating the librarians
3. Start the command loop
4. Wait while the application runs, so it does not exit immediately
5. Accept simple commands from the terminal

Next, you might expect make code changes to the RareBooks actor. There are none! Instead, use the actor you created back in chapter 4 (listing 4.3, listing 4.4, and listing 4.5). It is the same.

6.2.2 Introducing application.conf

On startup, Akka looks for a file called application.conf. Listing 6.2 is a starting point for a custom application.conf for an Akka application. You should find it easy to understand the syntax, although some of the parameters will be new:

Listing 6.2 Sample application.conf for an Akka Application

```
akka {

  loggers = ["akka.event.slf4j.Slf4jLogger"]                    (1)

  loglevel = "ERROR"                                            (2)
  stdout-loglevel = "ERROR"                                     (3)

  logging-filter = "akka.event.slf4j.Slf4jLoggingFilter"        (4)

  actor {
    provider = remote                                           (5)

    default-dispatcher {
      throughput = 10                                           (6)
    }
  }

  remote {
    log-remote-lifecycle-events = off
    netty.tcp.port = 4711                                       (7)
  }
}
```

1. Loggers to register at boot time
2. Log level used by the configured loggers as soon as they have been started. Options: OFF, ERROR, WARNING, INFO, DEBUG
3. Log level for the very basic logger activated during ActorSystem startup, before the configured loggers have been initialized. Sends log messages to stdout
4. Filter of log events that is used by the LoggingAdapter before publishing log events to the eventStream
5. Configure Akka for remote references. Options: local, remote, cluster
6. Throughput for default Dispatcher, set to 1 for as fair as possible
7. The port clients should connect to. Default is 2552.

The format for application.conf should look familiar. If you are thinking JSON, then you are close. It is a superset of JSON called HOCON (Human-Optimized Config Object Notation).

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

6.2.3 Using HOCON

The primary goal for HOCON is to stay close to JSON format while providing convenient, human-readable configuration notation. To be both machine-readable and human-readable, the format should be:

- *A JSON superset*—all valid JSON should be valid and should result in the same in-memory data that a JSON parser would have produced
- *Deterministic*—the format is flexible, but it is not heuristic. It should be clear what is invalid, and invalid files should generate errors
- *Parsed with minimal look-ahead*—the file can be tokenized by looking at only the next three characters. Right now, the only reason to look at three is to find comments that start with //. Otherwise, it would require only two characters.

Paraphrased from the HOCON documentation, the following features are also desirable, to support human usage:

- Less noisy/less pedantic syntax
- Ability to refer to another part of the configuration, so you can set a value to another value
- Import/include another configuration file into the current file
- Map to a flat properties list such as Java's system properties
- Get values from environment variables
- Ability to write comments

Independent configuration is a powerful notion, and part of what allows Akka to be distributed by design.

6.2.4 Configuring the driver application

You have one final step to complete remote setup for RareBooks and that is, as you might expect, the configuration. RareBooks needs to be configured to accept messages from other actor systems. In addition, some of the hard-coded parameters move from the domain model into the configuration file where they easier to manage. Following are the changes required for the application.conf file:

Listing 6.5 RareBooks application.conf

```
akka {
  loggers  = [akka.event.slf4j.Slf4jLogger]
  loglevel = DEBUG

  actor {
    debug {
      lifecycle = on
      unhandled = on
    }

    provider = remote                                    ❶
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

```
  }

  remote {
    enabled-transports = ["akka.remote.netty.tcp"]      ❷
    log-remote-lifecycle-events = off

    netty.tcp {                                         ❸
      hostname = localhost                              ❸
      port = 2551                                       ❸
    }
  }
}

rare-books {                                            ❹
  open-duration = 20 seconds                            ❹
  close-duration = 5 seconds                            ❹
  nbr-of-librarians = 5                                 ❹
                                                        ❹
  librarian {                                           ❹
    find-book-duration = 2 seconds                      ❹
    max-complain-count = 2                              ❹
  }                                                     ❹
}                                                       ❹
```

❶ **Enable remote providers**
❷ **Remote transport added**
❸ **Specify the listener hostname and port number**
❹ **Use HOCON to specify parameters for the domain model**

Consider what you have accomplished. You started with an actor that was embedded in a self-contained simulation. RareBooks is now a standalone application that is prepared to accept messages from other systems. Now that the store is in business, you can give it some customers. Like any store in the real world, it is important for customers to be able to find the store. From the perspective of the customer, RareBooks is a remote actor.

6.3 Remoting with Akka

Location transparency is a fundamental concept in computer science where a unique logical identifier is used to represent the physical location of a distributed resource. You can liken location transparency to the Internal Revenue Service (IRS) use of social security numbers (SSN), where the SSN is the logical identifier, and your name is the physical address. Regardless of what you change your name to (assuming you do it legally), you always have the same SSN that the IRS uses for tax purposes.

Akka takes the idea of location transparency and pushes it to its limit by making it purely driven by configuration. The result is few or no code changes when moving from a local to distributed environment.

In Akka, this notion is foundational to remoting and significant for resilience. It is quite common due to fault conditions that the physical location of our resources changes over time. For example, in a clustered environment when a node crashes the coordinator may spin up a

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

replacement on a different machine. The result is that the new node's physical address has changed and, if that physical address were in the code, the application would be broken. You will learn more about clustering Akka in the next chapter.

Location Transparency vs. Transparent Remoting

Location transparency should not be confused with another concept, transparent remoting. Transparent remoting is a pattern whereby a remote proxy is created that conforms to the interface of a remote object. Instead of executing the methods of the remote object locally, the parameters are serialized and sent over the network. The remote object then deserializes the parameters, executes and marshals a return. Transparent remoting is an attempt to generalize the model for both local and remote objects, whereas location transparency is an optimization for local and remote communication.

6.3.1 Constructing remote references

You learned in section 3.5 that an Actor System is hierarchical in nature and provides a unique trail of actor names that can be navigated recursively. You also saw how to construct a URI to reference an actor in the local actor system. Figure 6.5 shows how to extend that to a remote actor. The differences are:

- The protocol changes from `akka` to `akka.tcp`. Akka supports both TCP and UDP for remote transports.
- The hostname or IP address is included after the actor system name.
- The listener port is included after the hostname or IP address. The default is 2552, but it is better to include it explicitly.

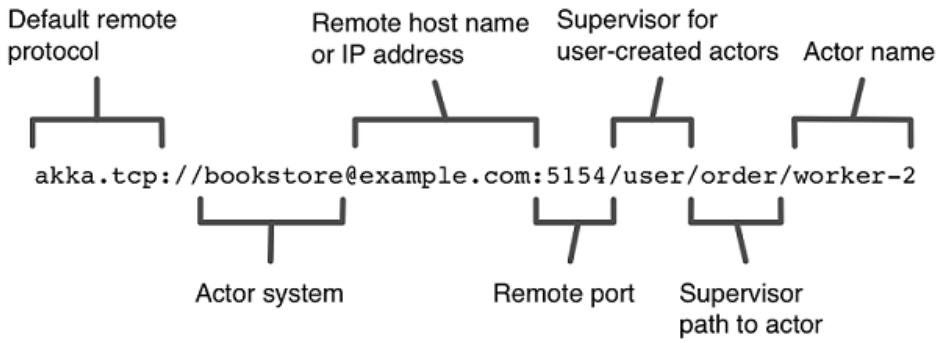

Figure 6.5 Remote actor references use a different protocol than local references, and adds the hostname and port to locate the actor system. The path of supervisors leading from /user to the named actor remains the same.

Notice that the protocol for remoting is *not*HTTP.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

6.3.2 Resolving the references

Once you have a URI, the next step is to convert it into an ActorRef. The steps are often combined into a single expression, but we will go through them individually. The first step, showing in the following listing, is to get the physical base address of the remote system:

Listing 6.6 Configurable Hostname and Port

```
import akka.actor.Address

private val hostname: String =                                    (1)
[CA] system.settings.config.getString("rare-books.hostname")      (1)

private val port: Int =                                           (2)
[CA] system.settings.config.getInt("rare-books.port")             (2)

val rareBooksAddress: Address =                                   (3)
[CA]Address("akka.tcp", "rare-books-system", hostname, port)      (3)
```

(1) Get the hostname for the rare-books remote actor system
(2) Get the port number for rare-books remote actor system
(3) Create the base address

The values for hostname and port are configurable and reside in the Customer's application.conf, which will be reviewed in the next section. The next step is to create an ActorSelection that identifies the complete path to the remote actor. The path begins with the root, which comes from the base address that was just constructed. It is followed by "user", which is the built-in guardian actor, then by "rare-books", which is the name of the actor being selected, as follows:

Listing 6.7 Selecting the path for the remote actor

```
import akka.actor.{ActorSelection, RootActorPath}

val selection: ActorSelection  =
[CA]system.actorSelection(                (1)
[CA]RootActorPath(rareBooksAddress) /     (2)
[CA]"user" / "rare-books")                (3)
```

(1) Use the actor system to perform the selection
(2) Generate a root path using the previously created rareBooksAddress
(3) Extend the selection path through the guardian to the rare-books actor. RootActorPath conveniently declares the / operator to make paths easier to build and read

The final step is to resolve the actor selection into an ActorRef. As part of resolving the selection, Akka exchanges messages with the remote actor system to verify that the reference is valid. Because messages are asynchronous, the resolution takes some time. Therefore, the result of the resolution is a Future[ActorRef] rather than just an ActorRef, and there is a timeout specified. When the Future completes, the ActorRef is then used to construct a Customer actor in the local system, as follows:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

Listing 6.8Resolving the actor selection

```
selection.resolveOne(resolveTimeout).onComplete {                    (1)

case scala.util.Success(rareBooks) =>                                (2)

system.actorOf(Customer.props(rareBooks, odds, tolerance))           (3)

case scala.util.Failure(ex) =>
log.error(ex, ex.getMessage)
}
```

(1) Akka address created based on hostname and port for rare-books
(2) On successful completion, the rareBooks ActorRef is returned
(3) When the actor resolution completes, use it create a customer

No changes to the customer actor are needed other than the minimal side effects of refactoring. You have created a new application that can be invoked remotely without changing the core domain. This notion is one of the key aspects of the Akka toolkit.

Creating actors remotely

Actor selection is used to find remote actors that already exist. It is also possible to create new actors on remote systems. This could be used, for example, to increase the number of librarians available to the RareBooks store in the example application. To create actors remotely with Akka, add the remote actor path to the *deployment* section of application.conf:

```
akka {
  actor {
    deployment {
      /worker-2 {
        remote = "akka.tcp://remoteWorkerSystem@127.0.0.1:2556"
      }
    }
  }
}
```

You can set up the deployment with code rather than configuration too. The documentation for akka.actor.Props.withDeploy() is a good starting point.

Once the deployment configuration is in place, use ActorSystem.actorOf or ActorContext.actorOf as usual to create actors remotely, like this:

```
context.actorOf(Props[Worker], "worker-2")
```

ActorSystem.actorOf is used when starting the system, and ActorContext.actorOf is used from within actors already created. Unlike actorSelection, actorOfrequires Props, which is an immutable class for specifying options for the creation of an actor, and the class needs to be available on the classpath of the remote actor system.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

6.3.3 Replacing clients with peers

If you are used to working with HTTP, you are familiar with the idea of having a request and a response. In reactive systems, messages are one-way. A response is nothing more than another one-way message back to the sender, as shown in this simple Echo actor:

Listing 6.8 Echo the original message back to the sender

```
import akka.actor.Actor

class Echo extends Actor {
  def receive = {
    case msg =>
      sender() ! msg            (1)
  }
}
```

(1) **Akka makes the sender available as an `ActorRef` through `Actor.sender()`**

This arrangement works because a reference to the sender is included in every message, and Akka makes it available to the receive method.

Actors are equal peers. When a response is returned, it is just another one-way message from one actor to another, in this case from the current actor to the sender. An actor may be a client for one interaction, a service to another, and any other role as needed by the domain model. One consequence is that an actor sending a message does not necessarily expect a single message in response. Whatever responses are produced may come from the same actor or a different actor. There could be one message, multiple messages, or perhaps none!

> **WARNING** Calls to the `sender()` function are valid only within the scope of the receive method. Be careful not to make the function accessible to other threads. This error happens most often when the sender is used in a Future. If you need to pass the sender, resolve it to value first. Akka guarantees that an actor only processes one message at a time. This is essential because if the receive method had two simultaneous callers then it would have no way to decide which ActorRef to return.

Another consequence is that, like all actors, an actor in the role of a client needs to run within an actor system too. In the RareBooks example, separating Customer into another JVM requires creating another actor system for it, as shown in figure 6.6.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

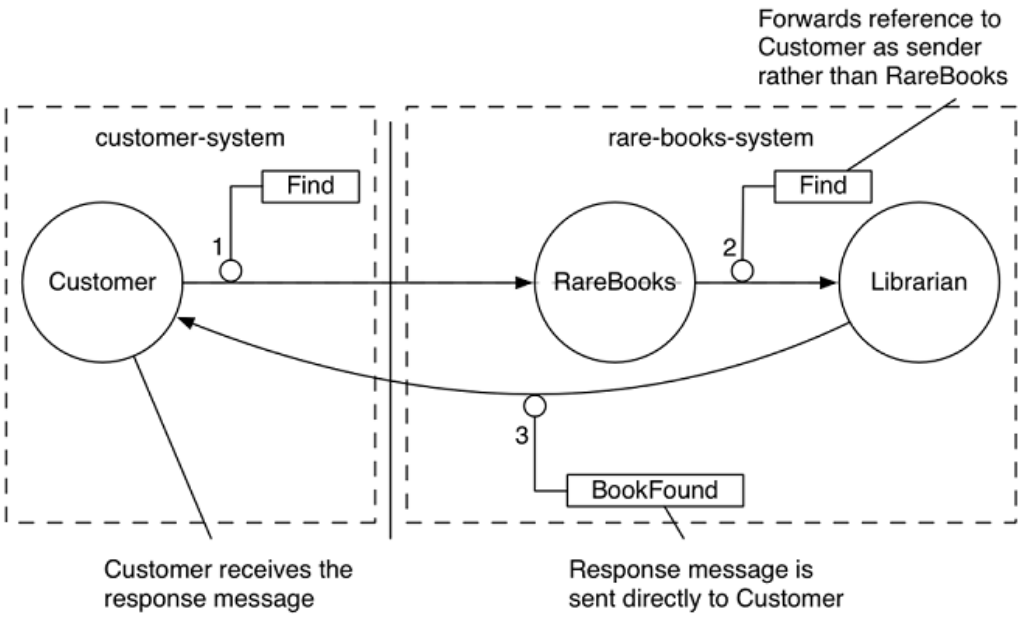

Figure 6.6Every actor lives inside an actor system. When an actor sends a message, responses may come from the same actor, a different actor, not at all

In this example, a Customer actor creates a remote reference to RareBooks and sends a Find message. The RareBooks actor then *forwards* the Find message to the Librarian using a router. Because the message was forwarded, the sender is not changed, so it appears to the Librarian that the sender was the Customer actor. The Librarian then sends a new BookFound message directly to the Customer.

Now that you know how to create a remote reference from a Customer to the RareBooks actor and you know that the Customer actors needs their own actor system, all that remains is a driver application for the Customer actors.

6.3.4 Creating another driver

As shown in figure 6.7, the customer driver follows the same pattern as the RareBooks driver you created earlier in section 6.2. There is a `CustomerApp` class with a corresponding companion object containing the `main()` function. It creates the actor system, and the guardian user actor is created automatically by Akka.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

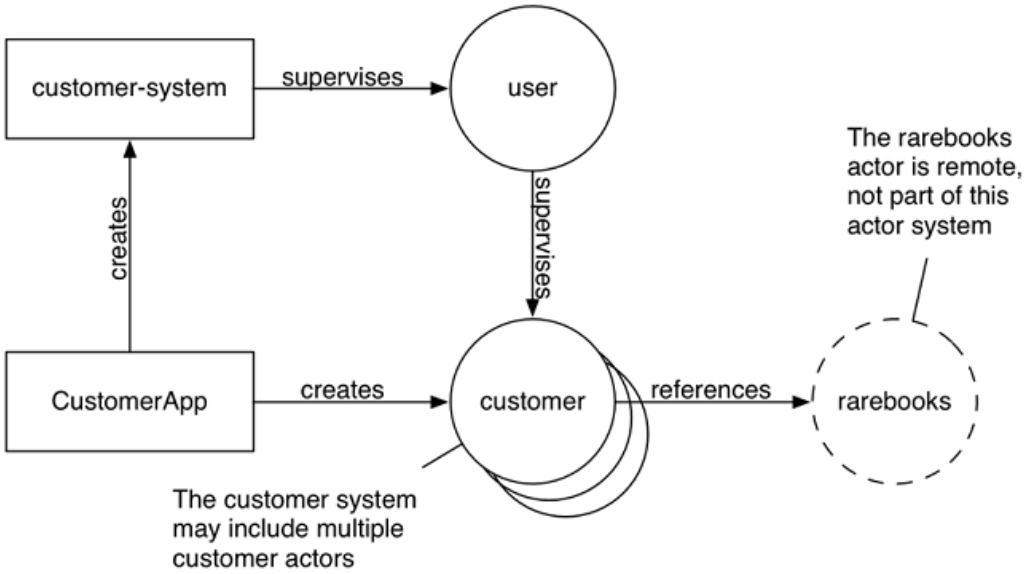

Figure 6.7 The CustomerApp driver and Customer actor

The difference between the `CustomerApp` and the `RareBooksApp` you created earlier in section 6.2 is that the `CustomerApp` resolves an ActorRef to RareBooks and creates multiple Customer actors instead of just the single RareBooks actor. That step is shown in the following listing, which combines the resolution steps and creates *count* customers. You can download the complete source from https://github.com/ironfish/reactive-application-development-scala.

Listing 6.9 Resolving the RareBooks actor and creating multiple Customer actors

```
protected def createCustomer                                    (1)
[CA](count: Int, odds: Int, tolerance: Int): Unit = {           (1)

system.actorSelection(RootActorPath(                            (2)
[CA]rareBooksAddress) / "user" /"rare-books").                  (2)
[CA]resolveOne(resolveTimeout).onComplete {                     (2)

case scala.util.Success(rareBooks) =>

    for (_ <- 1 to count)                                       (3)
    system.actorOf(Customer.props(rareBooks, odds, tolerance))

  case scala.util.Failure(ex) =>
   log.error(ex, ex.getMessage)
  }
}
```

(1) Called by the command loop
(2) Combine the steps to resolve the ActorRef

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

❸ Create *count* customer actors using the resolved ActorRef

Most of the configuration is the same too, as shown in the following listing:

Listing 6.10 Customer application.conf

```
akka {                                                    ❶
  loggers  = [akka.event.slf4j.Slf4jLogger]               ❶
  loglevel = DEBUG                                        ❶
                                                          ❶
  actor {                                                 ❶
    debug {                                               ❶
      lifecycle = on                                      ❶
      unhandled = on                                      ❶
    }                                                     ❶
                                                          ❶
    provider = remote                                     ❶
  }                                                       ❶
                                                          ❶
remote {                                                  ❶
    enabled-transports = ["akka.remote.netty.tcp"]        ❶
    log-remote-lifecycle-events = off                     ❶
                                                          ❶
    netty.tcp {
      hostname = localhost                                ❷
      port = 2552                                         ❷
    }
  }
}

rare-books {
  resolve-timeout = 5 seconds                             ❸
  hostname = localhost                                    ❹
  port = 2551                                             ❹
}
```

❶ Same as RareBooks application.conf
❷ Both JVMs in the example use localhost, so choose a different port number than RareBooks
❸ How long to wait for RareBooks remote reference to be resolved
❹ Hostname and port number of RareBooks

Now you have the two applications ready and configured. The result of the refactoring should contain the files shown in figure 6.8. Congratulations! The only thing left to do is try it.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

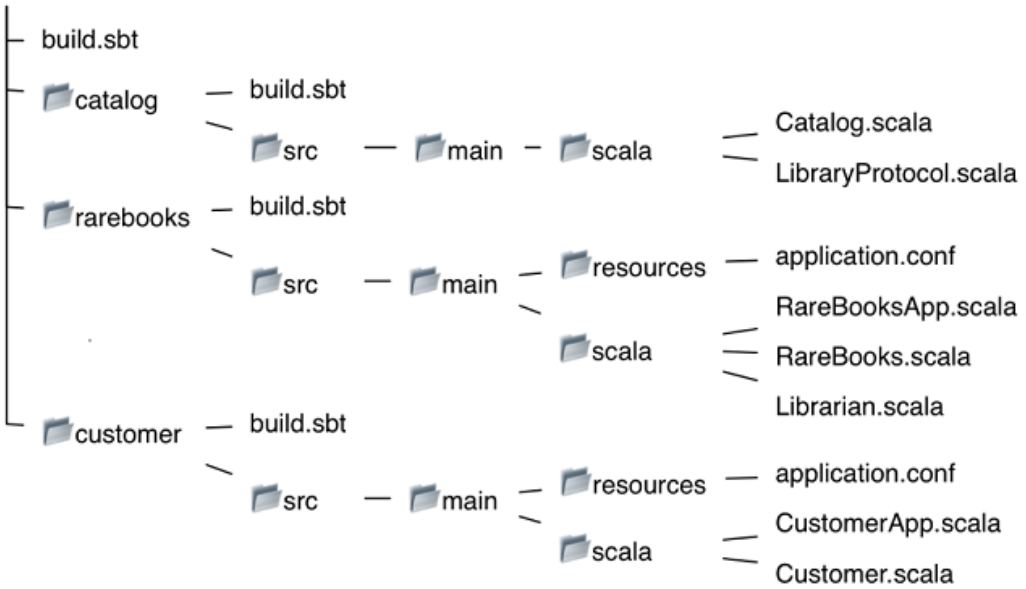

Figure 6.8 The completely restructured application

6.4 Running the distributed example

In chapter 4, you ran the application from a single terminal session. Because RareBooks now supports remoting, that is no longer possible. You need to run two terminal sessions, one for Customer and the other for RareBooks.

6.4.1 Starting the customer application

Begin with Customer. Start a new terminal session and enter the following:

Listing 6.17 Start sbt

```
$ cd reactive-application-development-scala/                    (1)
$ sbt                                                           (2)
[info] Loading project definition from
[CA] .../reactive-application-development-scala/project         (3)
[info] Set current project to reactive-application-development-scala
[CA] (in build file…/reactive-application-development-scala/)
```

(1) Navigate to examples directory
(2) Start sbt
(3) Complete path omitted

You are now running sbt in the root of the examples project. Switch sbt to the customer sub-project, as follows:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

Listing 6.18 Switch to customer subproject

```
> project chapter6_001_customer                                    (1)
[info] Set current project to customer (in build file
[CA].../reactive-application-development-scala/)                   (2)
>                                                                  (3)
```

(1) Switch to the customer subproject
(2) Complete path omitted
(3) Prompt for next sbt command

At this point, you have set the project to the customer and the next step is to start it running, as follows:

Listing 6.19 Start the customer application

```
> run                                                              (1)
[info] Running com.rarebooks.library.CustomerApp
10:33:35 WARN  [com.rarebooks.library.CustomerApp
[CA](akka://customer-system)] - CustomerApp running
Enter commands [`q` = quit, `2c` = 2 customers, etc.]:             (2)
```

(1) Start the customer application
(2) Waiting for you to tell the application how many customer actors to start

If you have made it this far, pat yourself on the back. You have just started the first remote application. It is ready to create some customer actors. Now before you create any customers, you need to start RareBooks. No sense ordering books if the store does not exist!

6.4.2 Starting the RareBooks application

As shown in the following, the steps are almost the same as for starting the customer application:

Listing 6.20 Switch to the RareBooks project and start the application

```
$ cd reactive-application-development-scala/                       (1)
$ sbt                                                              (1)
[info] Loading project definition from                             (1)
[CA] .../reactive-application-development-scala/project            (1)
[info] Set current project to reactive-application-development-scala (1)
[CA] (in build file…/reactive-application-development-scala/)      (1)
> project chapter6_001_rarebooks                                   (2)
[info] Set current project to rarebooks (in build file
[CA].../reactive-application-development-scala/)
> run                                                              (3)
[info] Running com.rarebooks.library.RareBooksApp
18:04:58 WARN  [com.rarebooks.library.RareBooksApp
[CA](akka://rare-books-system)] - RareBooksApp running
Waiting for customer requests.                                     (4)
```

(1) Same as starting the customer application
(2) Switch to the rarebooks subproject

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

❸ Start the rarebooks application
❹ Wait for the first customer

You now have both applications up and running! Let's start the ordering.

6.4.3 Creating some customers

Switch back to the terminal where you started customer and enter "2c" for two customers. You should see the following:

Listing 6.21 Customer output

```
2c                                                          ❶
18:13:38 WARN  [akka.serialization.Serialization            ❷
[CA](akka://customer-system)] - Using the default Java      ❷
[CA]serializer for class ...                                ❷
```

❶ Request two customer actors
❷ Ignore the serialization errors

If you switch to the RareBooks terminal session, you will see similar output.

6.4.4 Reviewing the results

Let the application run for a little while. Open a third terminal session to view the log output produced by the two applications sending book requests and responses back and forth. The result should look something like listing 6.22. Your log output will vary from the listing depending on how long you wait before starting the actor systems and creating customers, how many customers you create, and the settings in your application.conf files. Look at both applications and the log output. Try to find examples of librarians receiving complaints, issuing credits, and even shutting themselves down and being restarted by the supervisor actor after receiving too many complaints.

Listing 6.22 Logging from the complete system

```
18:02:04 WARN  [com.rarebooks.library.CustomerApp           ❶
[CA](akka://customer-system)] - CustomerApp running         ❶
Enter commands [`q` = quit, `2c` = 2 customers, etc.]:      ❶
18:04:58 WARN  [com.rarebooks.library.RareBooksApp          ❷
[CA](akka://rare-books-system)] - RareBooksApp running      ❷
Waiting for customer requests.                              ❷
18:04:58 DEBUG [akka://rare-books-system/user/              ❸
[CA]rare-books/librarian-1] - started                       ❸
[CA](com.rarebooks.library.Librarian@448eb83b)              ❸
...
18:04:58 DEBUG [akka://rare-books-system/user/              ❹
[CA]rare-books] - now supervising Actor                     ❹
[CA][akka://rare-books-system/user/rare-books/              ❹
[CA]librarian-0#-2013227586]                                ❹
...
18:13:38 DEBUG [akka://customer-system/user/$a] -           ❺
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

```
[CA]started (com.rarebooks.library.Customer@2b07b134)                    ❺
...
18:13:39 INFO  [akka.tcp://rare-books-system@                            ❻
[CA]localhost:2551/user/rare-books] -                                    ❻
[CA]Closing down for the day                                             ❻
...
18:13:39 INFO  [akka.tcp://rare-books-system@                            ❼
localhost:2551/user/rare-books] -                                        ❼
2 requests processed today. Total requests processed = 2                 ❼
...
18:42:26 INFO  [akka.tcp://rare-books-system@                            ❽
[CA]localhost:2551/user/rare-books/librarian-4] –                        ❽
[CA] BookNotFound(Book(s) not found based on                             ❽
[CA]Set(Unknown),1487140944853)                                          ❽
```

❶ Customer application startup
❷ RareBooks application startup
❸ Librarian actors starting
❹ RareBooks actor begins supervising the librarians
❺ Customer application starts some customers
❻ RareBooks closes for the day
❼ RareBooks generates daily report
❽ A search did not find a book

Pat yourself on the back a second time! You have just created a reactive application that supports remoting!

6.5 Reliability in distributed systems

Reactive systems are message-driven, so of course they send a lot of messages. At the small scale of a single computer running an example application as you did in the previous section, dropped messages should be rare. Consider the ramifications of a large-scale reactive system. Reactive systems are elastic and resilient. That means they may grow or shrink over time as actors are added and removed, and they are expected to handle failure even while passing millions or even billions of messages. Some messages will not arrive at their intended destination the first time they are sent.

6.5.1 Reliability as a design parameter

Ideally, every message would be delivered to its intended destination immediately and exactly once. Knowing that failures do occur, you can add logic to detect failures and retransmit messages that did not arrive on the first try. But there is a cost. Each new piece of logic adds to system overhead, so decreases the total capacity and creates another component that itself can fail. In reactive systems, failure conditions are made explicit. You as the application designer make decisions about the message reliability your application needs.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

AT-MOST-ONCE DELIVERY

It is easy to guarantee that a message will be delivered *at-most-once*. If you send it only once, either it will be delivered or it will not be delivered. Very often the reliability of delivery is high enough and the cost of missed messages is low enough that an at-most-once guarantee is perfectly acceptable.

AT-LEAST-ONCE DELIVERY

Guaranteeing that a message will be delivered *at-least-once* is more difficult. You can send a message and wait for an acknowledgement that it was delivered (figure 6.9). If you do not receive an acknowledgement within some time limit you can send the message again and again until you do receive an acknowledgement (figure 6.10). There are a few problems with that approach:

- There is no guarantee of how long it will take. If the intended recipient has failed or a network connection is not repaired before whatever time limit you have, the message will not be delivered in time.
- The sender must keep track of all the messages it sends until it receives an acknowledgement, in case it needs to try again.
- The problem could be that the messages are being delivered, but the acknowledgements are not being returned successfully (figure 6.11). In that case, many duplicates of the same message could be delivered as the sender keeps retrying without receiving an acknowledgement.

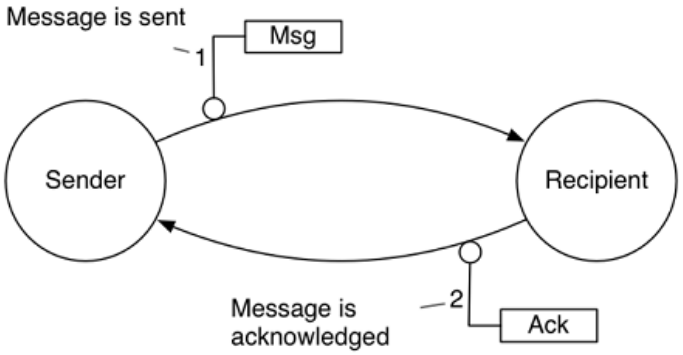

Figure 6.9 Guaranteed delivery requires some form of acknowledgement from the recipient

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

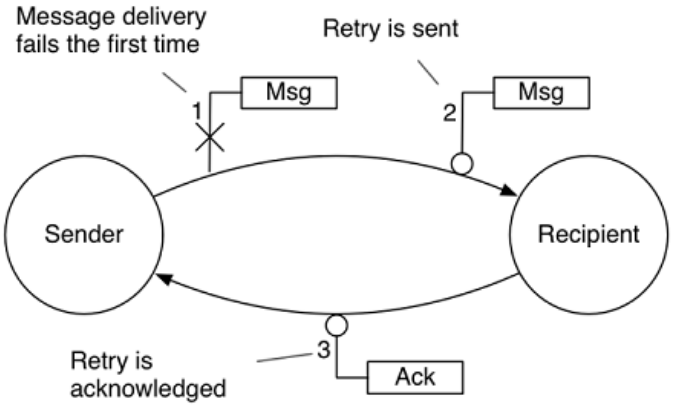

Figure 6.10 Retries can continue until either an acknowledgement is received by the sender or a timeout is exceeded

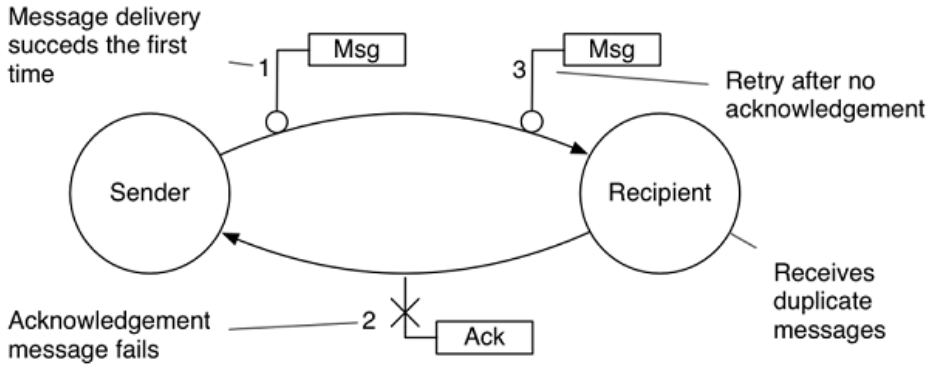

Figure 6.11 If the problem is acknowledgments are failing to reach the sender, the recipient could receive duplicate messages

Despite those problems, it is not uncommon for messaging systems to provide at-least-once delivery by setting reasonable time limits and bounds on how many unacknowledged messages the sender will retain before dropping some or refusing to accept more.

EXACTLY ONCE DELIVERY

Guaranteeing that a message will be delivered *exactly* once adds a new difficulty. The sender might not receive all the acknowledgements in time to avoid unnecessary retries, so the *receiver* must keep track of all the messages too. It needs them so it can recognize duplicates. In a busy service, that could be a lot of messages.

Worse, if you try to scale the service by adding more instances of the recipient, you face a new problem: If a message is received but the acknowledgement does not get back to the

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

sender in time, it will resend the message. The retry could be routed to a different recipient, as shown in figure 6.12. How does the second recipient know that it is a duplicate that has been processed already by the first recipient? It would need reliable communication with the first recipient, and the coordinator becomes another source of failure.

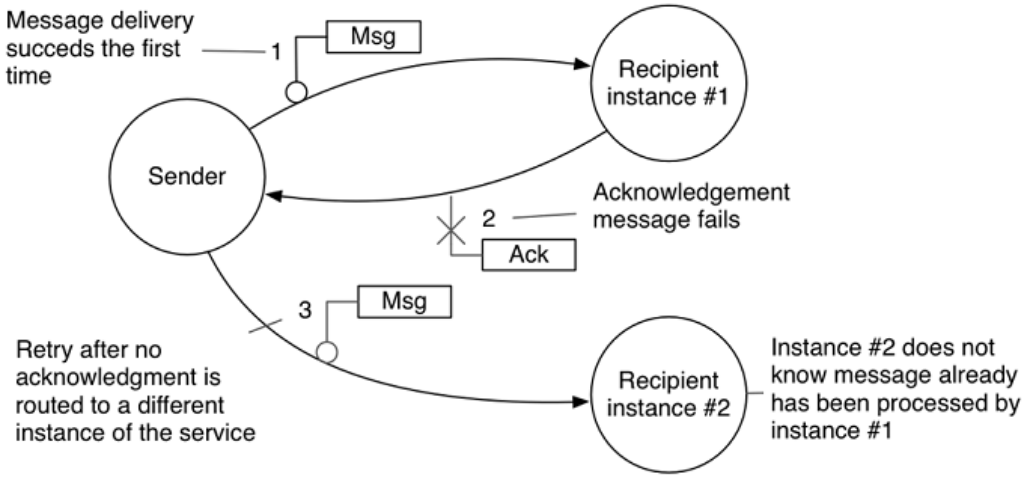

Figure 6.12 Guaranteeing that a message is received exactly once is even more difficult if there are multiple receivers needed to handle the processing load

6.5.2 Akka Guarantees

It should be apparent that a complete guarantee of delivery, whether at-most-once or exactly once, is not possible in a real system. You do not have forever and infinite retries available. You also should realize that each increment of reliability adds perhaps considerably to the system overhead. Once that overhead is added, you cannot recover the performance that was sacrificed. The solution in Akka is to provide the fastest option by default and allow you to layer additional reliability on top of actors as needed.

DEFAULT DELIVERY

As you learned in chapter 3, Akka messages are based on a mailbox concept. Each actor has its own mailbox that acts as a queue, and delivery to the mailbox is at-most-once. Akka adds an additional guarantee that messages delivered *directly* between two actors into the default mailbox type will not be out of order. They always will be added to the mailbox in the order they were sent.

The ordering property is useful for constructing application business logic. In the example application, at the end of each day RareBooks receives a Close message to close the store followed by a Report message to generate daily reports. It would be messy if the Report

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

message arrived before the Close message. The report might not include requests that arrived after the report was generated but before the store closed for the day.

The ordering property is also useful for adding resilience features to your application. If there is a gap in the sequence of messages, the receiver might take steps to recover. Having the original message arrive while the recipient is recovering for its absence again leads to messy logic. It is much easier to reason about the recovery process secure in the knowledge that the missing message will never arrive to confuse the recovery.

The ordering guarantee is only valid for direct messages from one actor to another. Inserting another actor such as a router between the actors can lead to unexpected results. Messages from a sender to the router will remain in order, as will messages from the router to an individual recipient (routee). The catch is that messages may arrive at different routees in a different order. This is illustrated in figure 6.13, where the first message may arrive at Recipient Aeven after the second and fourth messages are received (in that order) by Recipient B.

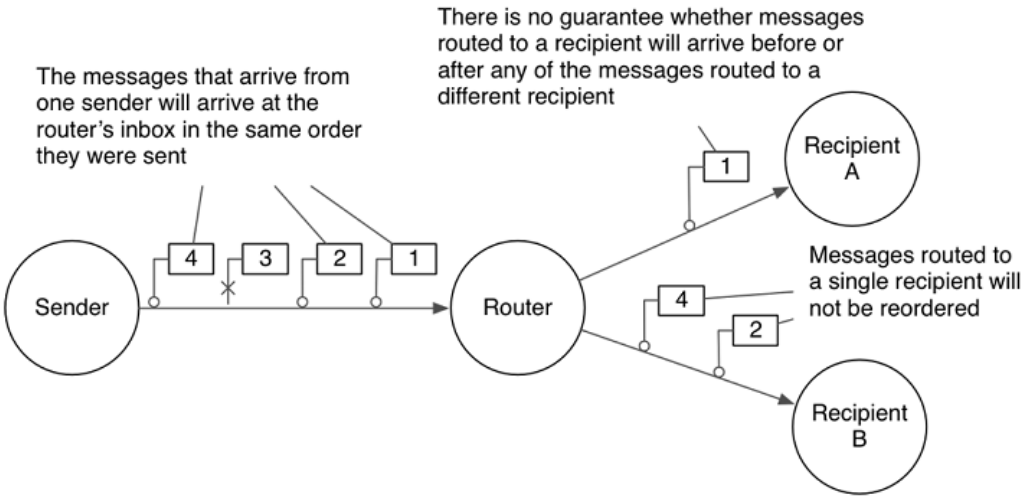

Figure 6.13 Akka defaults to at-most-once delivery with some guarantees on message ordering

OTHER MAILBOX TYPES

The guarantee on message ordering applies to the default mailbox. Akka has more than a dozen different mailbox implementations available that you can choose, and not all guarantee ordering in the same way. Those with the word "Priority" in their name reorder messages on purpose.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

AT-LEAST-ONCE DELIVERY

At-least-once delivery in Akka is perhaps better described as a toolkit for building at-least-once reliability into your application rather than as a complete solution itself. Underlying the at-least-once delivery mechanism is Akka Persistence, which is an Akka module that provides reliable state management by way of persistence across actor starts and restarts. Akka Persistence is covered in more detail in chapter 8.

To implement at-least-once delivery, you create the sender as a `PersistentActor` and mix in the `AtLeastOnceDelivery` trait. `PersistentActor` extends `Actor` with persistence features, and to use it you must include a build dependency on the Akka Persistence module.

To send a message with at-least-once delivery:

1. The sending actor produces an event that can be persisted, and provides a function to convert that event into a message for the destination actor.
2. The at-most-once delivery mechanism persists the event and sends the corresponding message.
3. When the receiving actor processes the event, it sends a confirmation message back to the sender.
4. The sender then marks the persisted event as confirmed.

The `AtLeastOnceDelivery` trait enables re-sending unconfirmed messages and supports a configurable timeout. While `AtLeastOnceDelivery` is a nice addition, it does require some work on your part. You are responsible for persisting both the message sent and confirmation received in the case of a JVM crash, and must provide a special Receive function to assist with recovery. Because it is at-least-once rather than exactly once, the recipient may receive duplicate messages. Finally, the order of messages may change.

Listing 6.23 Sample Send and Receive Actors Extending AtLeastOnceDelivery

```
import akka.actor.{ Actor, ActorSelection }
import akka.persistence.{ AtLeastOnceDelivery, PersistentActor }

sealed trait Cmd                                                    (1)
case class SayHello(deliveryId: Long, s: String) extends Cmd        (1)
case class ReceiveHello(deliveryId: Long) extends Cmd               (1)

sealed trait Evt                                                    (2)
case class HelloSaid(s: String) extends Evt                         (2)
case class HelloReceived(deliveryId: Long) extends Evt              (2)


class SendActor(destination: ActorSelection)                        (3)
    extends PersistentActor with AtLeastOnceDelivery {              (4)


  override def persistenceId: String = "persistence-id"             (5)


  override def receiveCommand: Receive = {
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

```
case s: String =>
      persist(HelloSaid(s))(updateState)                          ❻
    case ReceiveHello(deliveryId) =>
      persist(HelloReceived(deliveryId))(updateState)             ❼
  }


  override def receiveRecover: Receive = {                        ❽
    case evt: Evt => updateState(evt)                             ❽
  }                                                               ❽

  def updateState(evt: Evt): Unit = evt match {
    case HelloSaid(s) =>
      deliver(destination)                                        ❾
[CA](deliveryId => SayHello(deliveryId, s))                       ❿

    case HelloReceived(deliveryId) =>
      confirmDelivery(deliveryId)                                 ⓫
  }
}

class ReceiveActor extends Actor {                                ⓬
  def receive = {                                                 ⓬
    case SayHello(deliveryId, s) =>                               ⓬
      // ... do something with s                                  ⓬
sender() ! ReceiveHello(deliveryId)                               ⓬
  }                                                               ⓬
}                                                                 ⓬
```

❶ Define the messages that are exchanged between SendActor and ReceiveActor
❷ Define the objects that are persisted locally to track state
❸ The destination actor is an ActorSelection rather than an ActorRef so that it can be persisted
❹ The sending actor must be a PersistentActor and extend AtLeastOnceDelivery
❺ Name of the unique key for entries in the persistence layer
❻ To send a string, create the persistent event and update the state of the actor
❼ To process confirmation response, create the persistent event and update the state of the actor
❽ Used to replay events when the actor is recovering from a restart
❾ Tell the at-most-once delivery mechanism to deliver the message to the destination
❿ Used by the deliver function to transform the delivery ID into a message
⓫ Tell the at-most-once delivery mechanism that a confirmation message has been received
⓬ The ReceiveActor simply confirms every SayHello message with a ReceiveHello message that includes the deliveryId

Fully understanding `AtLeastOnceDelivery` requires understanding Akka Persistence and Event Sourcing, which are both covered in chapter 8. For now, the important thing to understand is that `AtLeastOnceDelivery` keeps track of which messages have not been confirmed, handles re-sending them, and uses Akka Persistence to manage storing the messages that have not been confirmed.

6.6 Summary

In this chapter, you learned that

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

- SBT can be used to define the structure of a reactive project with multiple driver applications.
- The Lightbend Config Library provides a human-readable format for storing and managing runtime configuration of your applications.
- Akka uses location transparency to maintain a uniform view actors, so an ActorRef may refer to a local or to a remote actor. Local and remote references share a URI syntax that does not use HTTP.
- Establishing a reference to a remote actor involves contacting the remote actor system to ensure the actor exists.
- All actors are peers to each other. If a client makes a request to an actor system and expects a reply, the client also needs to have an actor system.
- Akka provides limited delivery guarantees for messages. By default, it will guarantee *at-most-once* delivery and guarantee messages will not be received out of order. Using Akka Persistence, it can provide *at-least-once* delivery limited by timeouts and capacity.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

8

CQRS and Event Sourcing

As you've seen in the previous chapters, the Reactive paradigm is a powerful way to think about distributed computing. In this chapter, we will build upon that foundation by exploring two design patterns first introduced in chapter one: Command Query Responsibility Segregation (CQRS) and Event Sourcing (ES). We should note however that while these techniques work in concert and are a natural fit for Reactive programming, they are by no means the only way to design a Reactive system.

To achieve our goal of exploring these two design patterns, we will take a look at a very common type of application. The "database-driven application," which is the foundation of many, if not most systems built today. We will explore the challenges behind building these types of applications in a distributed environment and show how through the Reactive paradigm, using CQRS/ES we can overcome their difficulties. Before we get started though, we need to level set what we mean by "database-driven application". The term *database-driven application*, can mean many things to many people, but at its root we find a simple meaning. It is a software application that takes in and persists data and provides a means to retrieve that data. After a discussion of the driving factors towards CQRS/ES we will explain the concepts and alternatives to your typical monolithic, database-driven application. We will look beyond the theoretical notion of our reactive applications being message-driven and ground that concept into a reality using CQRS *commands* and event sourcing *events* as our actual messages.

8.1 Driving Factors Towards CQRS/ES

A major driving factor and something all of us have dealt with in our careers is RDBMS, *Relational Database Management Systems*, traditionally in the form of SQL, *Structured Query Language* databases, have directly impacted how we build applications. Frameworks such as

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

JAVA Enterprise Edition (EE), Spring and Visual Studio made SQL the foundational norm for system development as we entered the age of affordable and more agile computing. Prior to SQL database, mainframe systems were the central computing areas of business and end users were limited to usage of dumb terminals, reports, etc. SQL databases offered an accessible and easy to use storage model and we quickly became dependent on the capabilities offered and even built systems around them. The growing number of desktop PCs led to a growing number of systems built against these databases and freedom to create systems, free from rigid Management Information Systems fiefdoms was great for computing overall. Unfortunately traditional solutions have clear limitations, preventing the ability to distribute applications and usually resulting in monolithic designs such as *ACID transactions*. We'll discuss ACID transactions and their prevention of distribution as well as the pitfalls of relational databases and CRUD next.

8.1.1 Atomic, Consistent, Isolated, Durable Transactions (ACID)

Typically monolithic applications were built, allowing database transactions across domain boundaries. In this world it was perfectly possible to have a transaction that guaranteed that a customer was added to a customer table before and order was added to an order table, all at once because all of the data resided in the same place. These transactions are typically referred to as *ACID Transactions*[15], meaning transactions having the properties of *Atomicity, Consistency, Isolation and Durability*.

- Atomicity allows performing multiple actions as a single unit. This is the basis of transactions where an update to a customer address can take place in the same unit of work as creating an order for that customer.
- Consistency here really means strong consistency as all changes to data are seen by all processes, at the same time and in the same order. This also is a characteristic of transactions and is so expensive that processes dependent on this type of consistency, as most database-driven applications are, can not distribute.
- Isolation provided safety against other, concurrent transactions. Isolation dictates how data in the act of change behaves when read by other processes. Lower isolations levels allow greater concurrent access to data but a greater chance of concurrent side effects on the data (reads on stale data). Higher isolations levels yield more purity on reads but are more expensive and comes with the risk of processes blocking other processes or worse yet *deadlocks*, the case in which they stay blocked indefinitely.
- Durability means that transactions, once committed, survive power outages or hardware failures.

[15] https://en.wikipedia.org/wiki/ACID

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

In order to have a reactive application, the application must be able to be distributed. It is not possible to have ACID transactions across distributed systems because the data is physically located across geographic, network or hardware boundaries. There is no such thing as an ACID transactions across these types of boundaries so unfortunately we need to cut the cord of traditional RDBMS dependence. Distributed systems cannot be ACID compliant.

We say *Traditional RDBMS* because some relational databases such as PostgreSQL have evolved and overcome relational limitations with features like object relational storage and asynchronous replication, allowing distribution.

8.1.2 Traditional RDBMS - Lack of Sharding

These RDBMS based systems do not *shard*, sharding is a way of spreading your data using some shard key and may be used to intelligently co-locate data but also to simply distribute that data. Sharding provides horizontal scaling across machine hardware and geographical boundaries and therefore allows elasticity and distribution of data. An example of sharding might be a photo sharing application that uses a user's unique id as the shard key. All of the user's photos would be found in the same physical area of the database. Without the ability to shard, a traditional RDBMS is reduced to throwing more hardware and storage at the problem *for the same physical database*, which is of course called vertical scaling and you can only scale so far with interconnected hardware. Remember *horizontal scaling* is scaling across geographical or machine boundaries and sharding is a way of achieving distribution and paves the way for dividing up systems in terms of CQRS.

8.1.3 CRUD

As discussed in chapter 1, *CRUD*, create, read, update, delete is the in place modification of a piece of data, in a single location. With CRUD, all updates are destructive, losing all sense of the previous state of the data, including deletes, which is the most radical update of all. With CRUD, valuable data is lost constantly. There is no concept of state transition, the current state of any object is all you get. For example, a completed order is just that, all notions of the new order, fulfilled order, in process order, etc. are lost. There is no history of the behavior of any given order making it impossible to trace how it got from point A to point Z. CRUD is not easily distributable in that the domain is mutable. Any distributed entity could be changed anywhere at any time and it's difficult to know what the single source of truth is. With CRUD, there is only ever a single source of truth and all other references to the CRUD entity is done by copy or reference only.

Far and away, the most common use of CRUD is with a relational database. In creating our database structure, we follow widely accepted best practices so that there is no redundant data, and allow relationships through the use of primary and foreign keys, for example; A customer table associated with a concrete, associated table or orders and joined by a foreign key field in the orders table. This model cannot scale because of the interdependency of the data constructs, which are bound together. To put it simply if the only way you can distribute

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

is to distribute the entire thing, you haven't really distributed anything. The table structures follow the patterns of what we think our business objects will look like usually through the use of hierarchical relationships. We then attempt to layer on top an object-oriented domain structure to tie it all together. While this approach is the foundation of many an application, the implied costs can be painful. Sometimes CRUD can be perfectly fine for simple applications. For example, a simple application to maintain "contacts" that has no views but that of the contact as it is stored in the database and no relationships to any other, or hopefully very few domains. It should be understood that it is much easier to build this type of application and if the simplicity of the system warrants it, it's perfectly ok. In most of the real work applications we've seen CRUD is not enough or correct. We need another solution and a way to do away with all of this pain. For those systems we apply CQRS, usually in combination with event sourcing as we'll start discussing next.

8.2 CQRS Origins

While many are just now familiarizing themselves with CQRS, it's been around since early 2008, crafted by Greg Young an independent consultant and entrepreneur. CQRS in many ways is similar to a much older pattern that is part of imperative programming: Command Query Separation (CQS). CQS was first coined by Bertrand Meyer[16] in his book, Object-Oriented Software Construction, 1997. The premise of CQS is that an object's methods should be distinctly divided into two areas:

- One that only returns results and does not change observable state: Queries
- One that changes observable state, but does not return results: Commands.

CQRS, on the other hand, takes this notion a step further. Rather than placing the burden of division on the object, CQRS says commands and queries should be separate objects. As is Greg Young succinctly puts it:

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

The single object we are used to is now divided into two, one for queries and one for commands and they now have two distinct paths. A real life example would be the creation of an order, which would be done against an order command object or module. The viewing of open orders would be against an order query module.

[16] https://en.wikipedia.org/wiki/Object-Oriented_Software_Construction

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

8.2.1 Command, Queries, and Two Distinct Paths

As its name so eloquently portrays, CQRS is about Commands, Queries and their segregation. We will talk about each of these concepts in detail as we work our way through this chapter, but as an opener, let's focus on the latter: segregation. It bears mentioning that CQRS combines so nicely with event sourcing (as we'll get to later), that we usually refer to the systems we build using this model as CQRS/ES.

Unfortunately, the word segregation (the S in CQRS) often has a negative connotation, and when applied to human relations and rightfully so. However, when it comes to Reactive systems, it is focal for resilience. The dictionary defines it as *"the action or state of setting someone or something apart from other people or things."* It is this *"setting apart"* that empowers CQRS/ES as a Reactive pattern. Segregation, in essence, isolates each side of a CQRS/ES system, providing fault tolerance to the system as a whole. If one side goes down, it does not result in complete system failure as we have *"isolated"* one side from the other. This pattern is commonly referred to as the *Bulkheading.*

Bulkheading finds its origins in the shipping industry, as shown in Figure 8.1, where bulkheads are watertight compartments designed isolate punctures to the hull. In the figure below (drawn by a famous NYC artist) there are 4 bulkheads. If one or more of the bulkheads are compromised, the ship can still remain afloat.

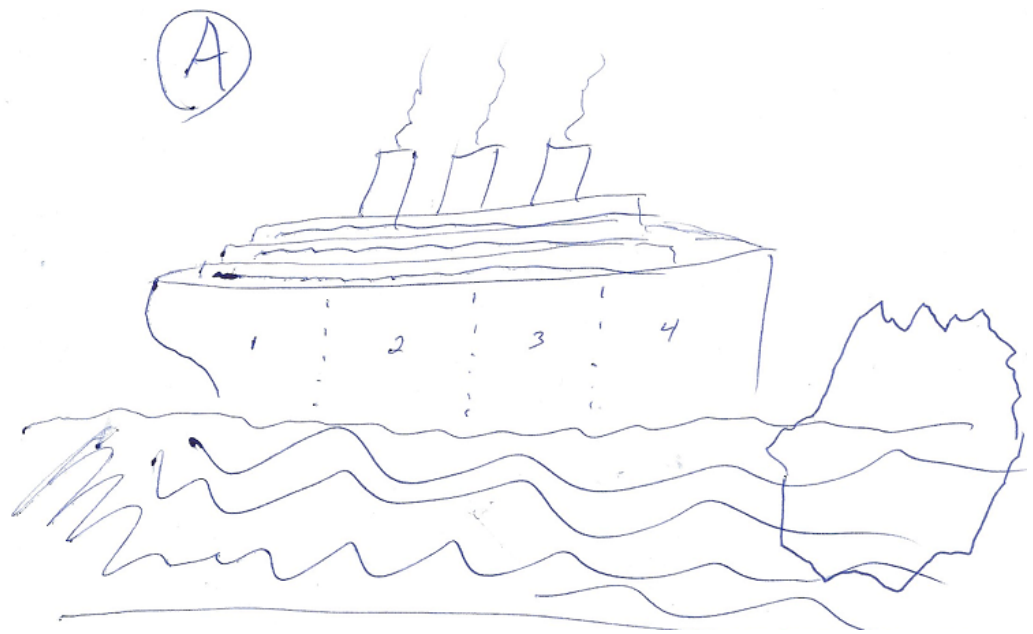

Figure 8.1 The SS !Titanic - Bulkheaded compartments ensure damage to one or more do not take down the ship.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

As the figure shows, if one of the bulkheads is compromised it prevents water from flowing into another, thus limiting the scope of failure. In order for the ship to sink, total failure, multiple bulkheads would have failed. The bulkheading pattern in CQRS may be demonstrated by a failure on the query (domain side) that precludes making any changes to an area of your domain. Since you have implemented the command side, all of the data necessary to read the last known state of the domain is still available and clients of that data are completely unaffected in terms of reads. We will illustrate bulkheading as it applies to CQRS a bit further down in this section.

Another aspect segregation provides, besides isolation, is the ability to optimize the different concerns or data writing vs reading. Rather than using a single pipeline to process both writes (commands) and reads (queries)...(think monolithic CRUD application), CQRS implements two distinct paths: Commands and Queries, thereby achieving a measure of bulkheading.

This division shadows the Single Responsibility Principle (SRP). SRP states: every context (service, class, function, etc.) should have one and only one reason to change: in essence, a single-responsibility. The single responsibility of the command side is to accept commands and mutate the domain within, the single responsibility of the query side is to provide views on various domains and processes in order to make client consumption of that query data as simple as possible. Through segregating and focusing on a single goal for each path, we can now refine our writes and reads independently. Also, we now have bulkheading in that the command side functions independently from the query side and any failures in either would not directly affect the responsiveness of the other.

A practical example is the case of many applications have significant imbalances between reads and writes. Systems such as high volume trading or energy might have a staggering amount of writes in terms of trades or energy readings but much less viewing of that data, or at least the data is aggregated before it is read. The Query side typically requires complex business logic in the form of *aggregate projections*, which are views on the domain(s) that may and usually don't look quite like the domain itself, for example, a view that includes customer information in detail, all their order history and a list of sales contacts. While the Command side simply wants to persist. A single model encapsulating both does neither well.

In Figure 8.2 below, we see four CQRS command modules representing the sales, customer, inventory and billing domains. Each of these domains are purpose-build to concentrate only on their narrow area of focus, with little concern of the other modules or for how the data will be queried. The domain concerns are neatly modeled in each of the command modules while the order to cash query module handles all of the heavy lifting required for presentation, which includes joining and pivoting the data to fit the client needs. The order to cash query module is a first class citizen in CQRS and exists solely to accumulate and return data across all of the command modules.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

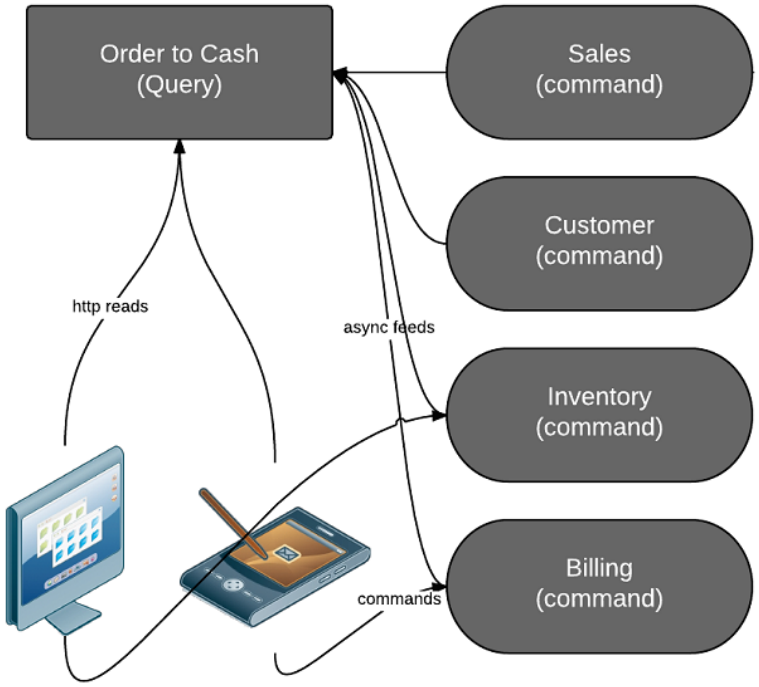

Figure 8.2 Two Distinct Paths - purpose-built domain and query modules focusing on a single duty

In Figure 8.2 you can see that reads and writes follow their own discrete paths, entirely independent from each other *in real time*, meaning that in real time, during reads and writes there is no connection between read and write data but it's important to keep in mind that there is an asynchronous relationship between the two and the constant syncing up over time using message driveness as we'll cover in event sourcing later in the chapter. The query data is always waiting to serve the clients in the last known state and offers the highest performance and simplicity in terms of reads. The query store is there for the sole purpose to serve the clients and may be built atop multiple domains or data feeds. Clients may used the read data to construct commands upon the domains (command sides). Those commands are always sent to a single command side and may result in a change to the query side but in an eventually consistent, asynchronous manner.

Now let's take a look at how bulkheading applies to a CQRS system.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

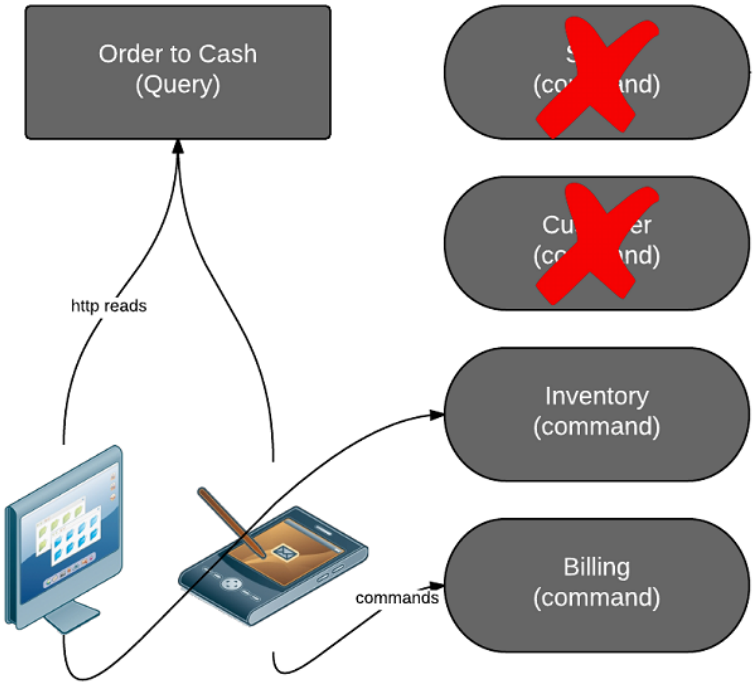

Figure 8.3 The natural bulkheading of CQRS, two command modules are unavailable but the rest of the system remains functional

In the above figure we can see that the sales and customer command systems have gone down or become unavailable. The natural bulkheading that CQRS provides allows users full access to the data necessary to populated displays and issue commands *but* obviously those commands will fail against the systems that are down. It is possible to to queue up commands somewhere so that they don't fail but this is of little value since the commands have no impact anyway until the failed systems are available once again. A better design would be something like redundant systems to mitigate these situations but the main point here is that there is always the possibility of failure somewhere and the best thing we can do is contain that failure in the best way possible, as we've shown here.

8.3 The "C" in CQRS

The *command* side, sometimes called the write side, is really about more than just commands. It represents the domain and the domain only. The command side is theoretically never meant

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

to be read directly.[17] The command side is tuned for high throughput writes and may have specific database and/or middleware selections for this very purpose, differing from the query sides. In this chapter we are going to show CQRS with event sourcing as that is the most valuable pattern in our experience but there is nothing to stop you from using CQRS alone when read and write separation is a requirement.

8.3.1 What is a Command?

Before we define what a command is, let's review our definition of what a *message* is from chapter one. A message is an immutable data structure used to communicate across service boundaries. Remember, one of the principal attributes of Reactive systems is that they are Message Driven. One of the types of messages that allows message-driveness are commands. Commands are sent from clients or services to various other CQRS services.

A *command* is a request to do something, and that something is usually a request to change state. A command is imperative, and while authoritative in nature, as a request, rejection is acceptable. Commands follow a Verb-Noun format, where the verb is the action requested and the noun is the recipient of that action. Typical commands look as follows:

```
CreateOrder(...)
AddOrderLine(...)
```

An interesting and powerful aspect to commands is that they may be rejected as we will see next.

8.3.2 Rejection Oh Noes!

The idea of rejection is an important concept in a CQRS system, but can be confusing when dealing with the notion of a command. Because commands are authoritative in nature, acceptance is often assumed. Moreover, why not? It is an imperative delivered by the authority! As mentioned above, in the context of CQRS, a command is more of a request than an order. As such, the recipient is at liberty to reject. Rejection is generally the result some form of business rule violation or attribute validation failure. An example of this, might be the rejection of a CreateOrder command. Why would this request be rejected? Perhaps the sender does not have the proper credentials to create orders, or maybe their credit line is overdrawn, etc.

Remembering back to chapter 4 and DDD, the command handler on the command side is also an anti-corruption layer on its associated domain so another interesting thing about a command is its structure in that it may contain multiple attributes requiring validation. This structure presents an intriguing question. What does one report back to the sender when

[17] Recent technologies such as Akka Persistence allow in-memory domain state to provide simple and performant reads of the domain but the lion's share of reads are left to the query side.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

rejection occurs? One error at a time or a list of errors? The answer to this question will be explored a little later in the chapter under Non-Breaking Error Validation. It bears mentioning that command validation comes with a cost in terms of performance and is sometimes not used, such as in cases of very high volume trading.

8.3.3 Atomicity

There seems to be an overwhelming buy-in of the microservice paradigm in large part because of the popularity of the reactive manifesto. The microservice paradigm dictates that we build many services, each doing a few focused things. This implies atomicity, not talking about ACID here. Atomicity meaning that each service solves a particular problem without knowledge or concern with the internal workings of others services around it. This philosophy falls in comfortably with the reactive manifesto, so much so that it could easily be the fifth pillar.

Atomicity allows solutions to smaller problems. The smaller the problem and the larger the isolation of that problem, the easier it is to solve. Look to avoid cross-cutting concerns in your service code. If an external concern or edge case tries to get worked into your service, find a way to elegantly handle such cases in a non-specific way. A small example before moving on; take for instance a service that applies discounts for books. There is a business case that applies a greater discount if the book was bought in a store rather than online. Rather than adding the ability to track whether or not any particular book was bought in a store, support it rather with an additional discount attribute, which is not at all knowledgeable of the forces that caused the discount but just the net result. This leads to a more open and core behavior to your service.

8.3.4 Jack of All Trades, Master of None

We are all familiar with the adage, *"jack of all trades, master of none"*. It is a figure of speech used in reference to a person who has a broad variety of skills, but is not particularly proficient in any of them. In many ways, this adage is apropos for CRUD based solutions. In chapter one, we explored a Monolithic shopping cart example comprised of several services. Digging deeper into that design by looking at an individual service, we will see why this design is a *"jack"* and not a *"master"*.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

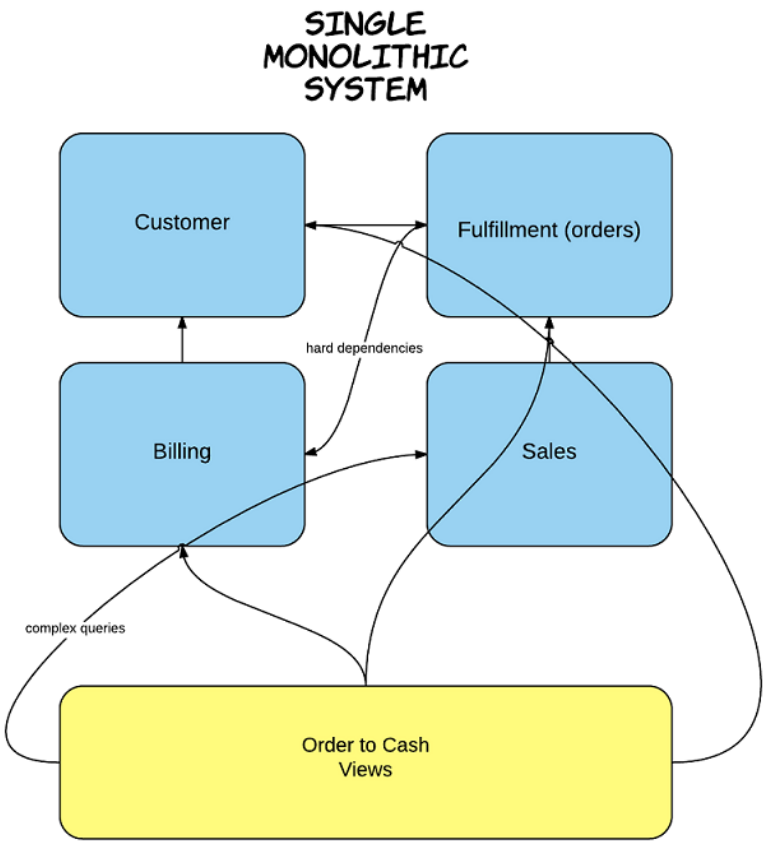

Figure 8.4 Detailed View of the Monolithic Order Service, domains with hard dependencies on other domains in the same application instance

In Figure 8.4, all things related to orders are in a single monolithic application for the sake of transactionality, build using the traditional RDBMS techniques, resulting in tying subsystems at the hip simply because they interact in some way. This will not scale and if one piece breaks, it's all broken.

Within the orders functionality in Figure 8.4 there is a single domain aggregate representing `Orders`. It is against this object that we action our three command behaviors, create, update and delete and our one query behavior read. The problem is that domain aggregates are designed to represent current-state and are not particularly conducive to data projections (queries). Usually, when projecting data, we need some form of aggregation as we want to see more than just the order on the screen. As a result, we have to rely on building Data Access Objects (DTO) projections off the aggregate that in turn require dynamic querying based on underlying SQL joins. This design is messy at best and becomes a continual point of

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

refactoring for query optimization. In the end, we are trying to use the aggregate for more than what its design supports.

8.3.5 Lack of Behavior

A major and very costly area of pain in typical monolithic applications (typically CRUD based) is the absence of data needed to derive intent. We discussed this in detail in chapter one, but it is important to review as this is an area where CQRS/ES shines. With CRUD, we have only four behaviors, create, read, update and delete. These are summary in nature, designed to modify current state models and lack the historical deltas required for capturing purpose. This loss of intention has a significant impact on the value of our business data as we are unable to determine user motive, which is paramount to understanding our user base. In using CRUD models we effectively throw away potentially massive amounts of data.

In order to better understand our users' needs, we must be able to create a profile of their usage habits so we can accurately anticipate future requirements. Creating this profile requires a detailed view of the actions that lead up to a particular decision made by a user. Data capture at this level allows us to not only predict future needs, but also answer questions not yet asked as we saw in chapter one. Additionally we can aggregate these results across our entire user base to discover all kinds of patterns related to the context of our system. In essence, this approach provides the bounty from which we can do Big Data analysis.

To build a system of this variety requires component focus and specialization. Our system components must be "*masters*" and not "*jacks*". CQRS/ES provides the philosophy and lays the groundwork for achieving this type of application focus. To demonstrate the focus and specialization of CQRS/ES we will devise a simple order tracking system. We will define the domain aggregates and their behavior in the form of commands and record those behaviors if valid in the form of historical events. We will see that not only do we have access to the current state of the aggregate, but we can derive that the state for any time in the past up to now. From this structure, we will have a natural audit log, be able to infer motive and have an architecture that is designed to distribute.

8.3.6 The Order Example Part 1 - The Order Commands

Let's take a look at a very familiar construct called an Order. We've all seen it before so theoretically it will make it easier to grasp what is different in CQRS. If you want a more interesting domain, take another look at the flight domain in Chapter 4 or keep reading the later chapters as we build a reactive application together.

An order domain would appear something like the following:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

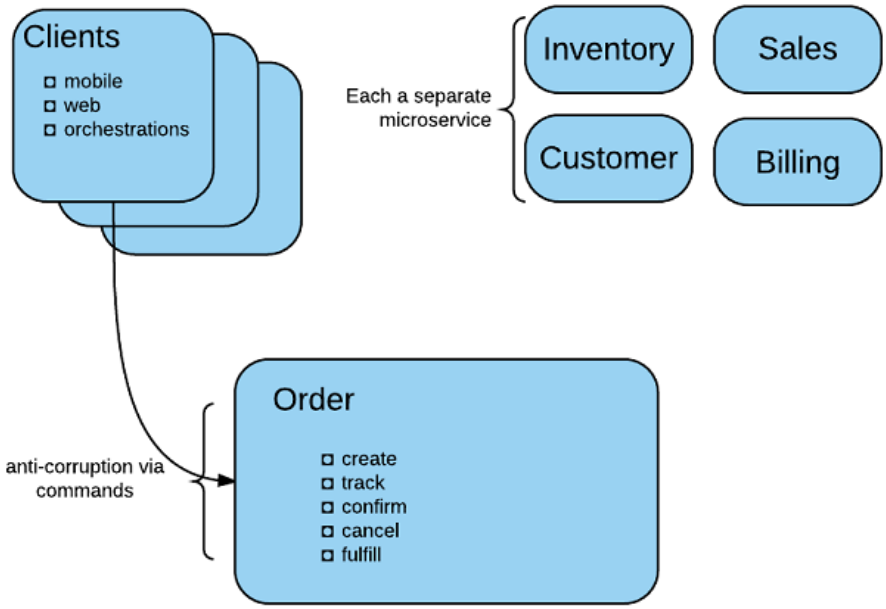

Figure 8.5 The new order domain, separated from other domains.

Above you now see the order domain as a completely separate and atomic service. The only way to interact is via commands sent to the service by http but any other transport will do the job as well. How does the order service interact with the other services in the upper right part of the diagram? The answer here is they also receive commands but these services also emit meaningful business events. These events may be utilized by orchestrations that may issue a command to order as a result of a successful billing event for example.

The order commands will functionally mutate the order but technically no object mutation occurs as immutable objects are first class citizens of a reactive application. The commands should be as granular as possible because the events derived from those commands are the building blocks of the event-driven nature of your applications. For example, the ChangeOrder command would result in an OrderChanged event. This event provides no business insight as to what actually happened to the order so at the very least, any other system that consumes order events must consume each and every OrderChanged event and interrogate it to determine if the change of the order is of interest. It is better and more expressive to break the events down to OrderShippingAddressChanged, OrderTotalChanged, etc.

The commands would look as follows:

- CreateOrder()
- ChangeBillingAddress()
- ChangeShippingAddress()

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

- PlaceOrder()

Remembering back to Chapter 4 and domain driven design, the Order is the aggregate root and contains OrderLine entities. All access to the order lines is through the root; Order. Now that we have the idea of commands in place, let's look at a clean way to ensure those commands are valid.

8.3.7 Non Breaking Command Validation

Non breaking validation is important to the responsiveness and usability of any microservice. Microservices that accept commands should validate those commands and not allow pollution of the domain within (remember anti-corruption layers? This is it). At the same time we should embrace and expect failure and the fact that the clients of of domain may not fully understand the ins and outs of the commands they are sending. What we can do is return either a positive acknowledgment that the command is accepted and we'll do our best to process it and generate resultant events OR a very convenient set of failed validations. The client can inspect these validations, make the repairs and send the commands again. This is especially valuable when teams are working in parallel and microservices are evolving, quick code fixes can be made to speed integration. It's easier to deal with an in-your-face command failure than trying to read and re-read another team's documentation as their domain evolves.

We don't break, ever if we can help it. That means guarding against nulls, empty strings, numericals vs text, as well as more complex structures that we can validation against a known set of valid values.

The example in Listing 8.1 uses the Scalactic library and implementation details may vary but the important thing is the contract. Using http/rest we can express the contract for a command response as either a 400-bad request, containing a json list of the failed validations or a 204-accepted meaning the command is valid and the system will make the best effort to process it. Let's take a look at how we might implement some validation on an imaginary Order domain object.

Listing 8.1 Order Validation

```
trait ValidationFailure {
  def message: String
}

case class InvalidId(message: String) extends ValidationFailure
case class InvalidCustomerId(message: String) extends ValidationFailure
case class InvalidOrderType(message: String) extends ValidationFailure
case class InvalidDate(message: String) extends ValidationFailure
case class InvalidOrderLine(message: String) extends ValidationFailure
case class InvalidOrderLines(message: String) extends ValidationFailure

❶
object OrderType {

  type OrderType = String
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

```
  val Phone = new OrderType("phone")
  val Web = new OrderType("web")
  val Promo = new OrderType("promo")
  val OrderTypes: List[OrderType] = List(Phone, Web, Promo)
}

❶

import org.scalactic._
import Accumulation._
import OrderType._

case class OrderLine(
  itemId: String,
  quantity: Int
)

case class Order private[Order] (
  id: String,
  customerId: String,
  date: Long,
  orderType: OrderType,
  orderLines: List[OrderLine]
)

❷
object Order {

❸
  def apply(customerId: String, date: Long, orderType: OrderType, orderLines:
       List[OrderLine]): Order Or Every[ValidationFailure] =
    withGood( ❹
      validateCustomerId(customerId),
      validateDate(date),
      validateOrderType(orderType),
      validateOrderLines(orderLines)
    ) { (cid, dt, ot, ols) => Order(UUID.randomUUID.toString, cid, dt, ot, ols) }


  ❺
  private def validateId(id: String): String or Every[ValidationFailure] =
    if (id !=null && !id.isEmpty)
      Good(id)
    else
      Bad(One(InvalidId(id)))

  private def validateDate(date: Long): Long Or Every[ValidationFailure] =
    if (date > 0)
      Good(date)
    else
      Bad(One(InvalidDate(date)))

  private def validateCustomerId(customerId: String): String or Every[ValidationFailure] =
    if (customerId !=null && !customerId.isEmpty)
      Good(customerId)
    else
      Bad(One(InvalidCustomerId(customerId)))
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

```
  private def validateDate(date: Long): Long Or Every[ValidationFailure] =
    if (date > 0)
      Good(date)
    else
      Bad(One(InvalidDate(date)))


  private def validateOrderType(orderType: OrderType): OrderType Or Every[ValidationFailure]
      =
    if (OrderTypes.contains(orderType))
      Good(orderType)
    else
      Bad(One(InvalidOrderType(orderType)))

  private def validateOrderLines(orderLines: List[OrderLine]): List[OrderLine] Or
      Every[ValidationFailure] =
    if (!orderLines.isEmpty)
      Good(orderLines)
    else
      Bad(One(InvalidOrderLines(orderLines.mkString)))
}
```

1. A structure we can validate order type against
2. This is the domain aggregate for Order
3. The default constructor is private so we can insure proper validation upon creation using apply in the companion object
4. All validations are performed and will either result in a call to the order's default constructor, thereby creating the order and returning it OR a collection of the failures
5. Each specific validation may be made as elaborate as necessary, we kept these very simple

Using the code illustrated above we can try constructing an order a couple of times, once with valid attributes and again with a couple invalid ones.

```
❻
scala>
import com.example._
println(Order("cust1", 1434931200000L, OrderType.Phone, List(OrderLine("item1", 1))));
Good(Order(14bad175-0fd9-4710-aa5b-
       c75006dc1246,cust1,1434931200000,phone,List(OrderLine(item1,1))))
```

6. As you can see, the order is created successfully

```
❼
scala>
import com.example._
println(Order("cust1", 1434931200000L, OrderType.Phone, List(OrderLine("item1", 1))));
Bad(Many(InvalidCustomerId(), InvalidOrderType(crazy order)))
```

7. The order is not created and the two validations are returned

In Listing 8.1 above we have shown embracing failure as a natural course of business via non-breaking validation. It shows that clients are imperfect and invalid data is expected, but not allowed to pollute the domain but instead returns the validations cleanly to the client so that they may change their call properly.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

8.3.8 Conflict Resolution

Now that we have a means to break down and distribute our applications in terms of queries and commands we need to pay attention to a small price to be paid, which is data consistency. We need to address the possibility that an action (command) may be taking place on a domain aggregate based upon a stale assumption of the state of the aggregate. With a distributed application, special attention must be paid to aggregate versions. A version is established each time any part of the aggregate state is changed, that change resulting in one or more events persisted to the data store, which may also be referred to as the event store on the command side. Each command is completely atomic from start to finish, meaning that if another command comes in it will be handled only after the previous one has mutated (changed) the aggregate state.

A potential problem arises when that next command has assumed a state to be of a previous version, a state previous to the completion of the command ahead of it. By allowing that next command to be handled and mutate the state of the aggregate based upon it's stale data would pollute the domain and this must be guarded against. For example, if an aircraft at 5000 feet of altitude is issued a command to reduce that altitude by 2000 feet by the tower and was then issued that same command by arrivals, who last knew the aircraft altitude as 5000, could easily cause a catastrophe. To guard against this we include an expected aggregate version in each command. Only commands that match the expected version to the current aggregate version are processed.

Now we'll show how to view all of this wonderful domain data, now carved up into independent and atomic services using the query side or "Q" in CQRS.

8.4 The "Q" in CQRS

Next we'll explore the query, or read side of CQRS. Here we'll show how to address the impedance mismatch between the read data and the domain data in which it derives. The asynchronous nature of updating the query data provides a clean separation from the sources of the data, with no interruption of any runtime behaviors on either the command or query sides but there is a small price. This design is subject to inconsistency in that there is always some delay between the current state of the domain(s) and the query stores that depend on them but the guarantee is that if all activity stopped in a series of connected CQRS systems, all data would eventually look the same, that is it will become *eventually consistent*. We sometimes call these stores *projections* of the domain. With CQRS alone, as opposed to CQRS with event sourcing, we don't prescribe exactly how the changes to the command side effect any given query sides.

8.4.1 Impedance Mismatch

When it comes to servicing clients of our systems, one of the first challenges we run into is object-relational impedance mismatch, as the queries we need to view the data is usually a

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

different shape than the data associated with the domain. The term *impedance mismatch* has It has its origins from an Electrical Engineering term known as impedance matching. In electronic circuit design, *impedance* is the opposition to the flow of energy from a given source. The idea is that as one component provides energy to another, the first components output has the same impedance as the second components input. As a result, the maximum transfer of power is achieved when the impedance of both circuits is the same. The simplest way to understand this in nature is using water as an example as in the next figure.

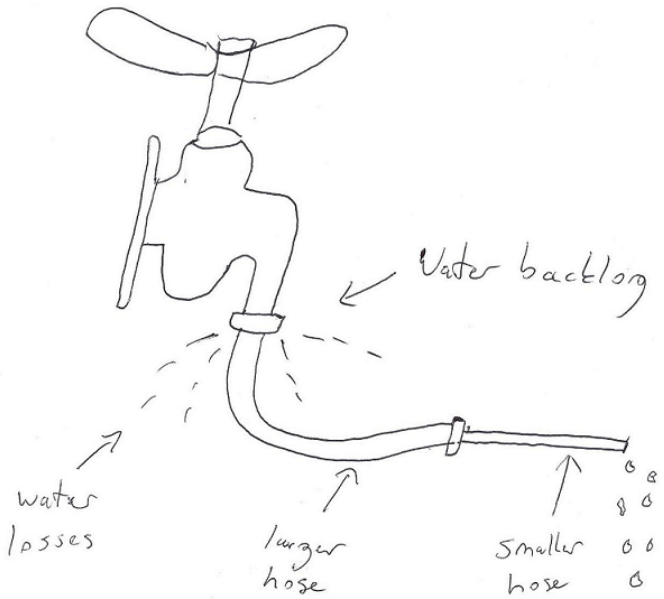

Figure 8.6 Impedance Mismatch in Water Flow - Large hose feeds into smaller one resulting in undesired water flow back in the opposite direction and water loss

The figure above should be easy for us non-engineers to follow. Hook a smaller hose up to a larger one and the water flowing from the source will meet the greater resistance (impedance in this case) of the smaller hose and water will be forced back and lost via leakage further upstream, here the leakage is in the larger hose connection to the faucet.

It may not seem obvious at first, but this impedance mismatch when it occurs between read and write concerns can be a significant challenge in our standard CRUD relational applications. Our object-oriented domain models that employ CRUD rely on techniques that encapsulate and hide underlying properties and objects. The problem results in CRUD semantics, requiring an ORM (object-relational-mapping), which must expose the underlying content to transform to a relational model. Thus violating the encapsulation law of OOP. While

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

not the "end of the world", just one more problem helping to cripple our ability to distribute. The following image shows impedance mismatch in software.

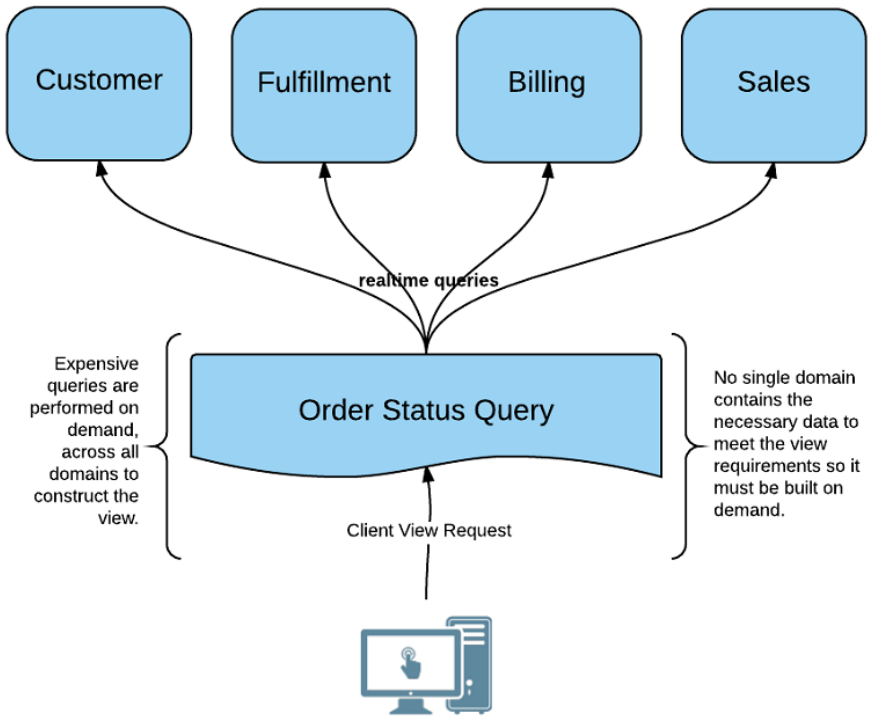

Figure 8.7 Impedance Mismatch in Software

Using Figure 8.7 as a reference, imagine you want to view the entire order status to get an overall picture of the order, who sold it, what was bought, what was shipped and what cash was collected, etc. Since our domains are nicely broken apart, even to the point of being entirely separate services, this order to cash view doesn't really exist, certainly it doesn't exist as a first class citizen within the diagram. The order status must be built on client demand by querying across all of the necessary domains, this is an expensive operation and hard to keep tuned, let alone scale. What is clearly missing is a query as a first class citizen. The above scenario demonstrates an impedance mismatch between the reality of the domains and the way the clients desire to view them. Just as a larger hose overwhelms a smaller one we see that trying to cobble together a client presentation in real time is too problematic and leads to not water loss, but loss in performance, risk and will be prone to errors.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

8.4.2 What is a Query?

A CQRS query is any read on a domain or a combination of domains. The associated domain data is considered a *projection* of the domain. A projection is a sort of picture of the domain state at a point in time and is eventually consistent with the events in which it is derived. The domain on the command side is rarely read directly but are used to build up current state that may quickly be read by clients. With CQRS queries, there are no calculations on reads, the data is always there waiting to be read and indexed according to the client's needs. Storage is cheap so different client data requirements simply mean multiple projections, avoiding anti-patterns such as the additions of secondary keys or worse, table scans to search within projections. Query projections can be as simple as the latest state of any domain or more complex and contain data across different domains. Below in Figure 8.8 we see our order to cash scenario we demonstrated in bulkheading, which serves also as a nice example of a query side.

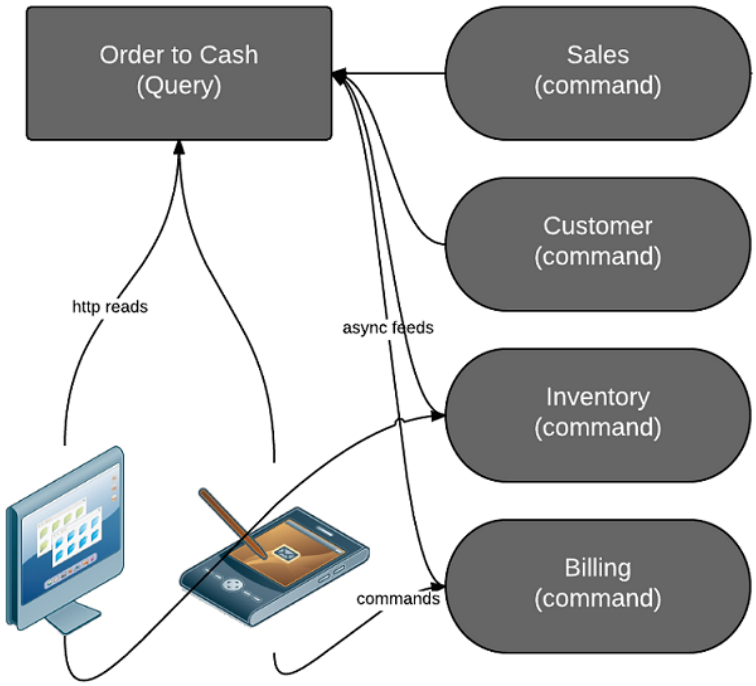

Figure 8.8 Order to cash query side, constantly fed command side events to build itself

In Figure 8.8 we see the independent command sides of sales, customer, inventory and billing all asychronously feeding the order to cash query module. By the time a client makes a call to view order to cash, all of the data is already built and waiting.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

Let's take a look at a simple projection of employee event data, assuming the command side is using event sourcing, which we will discuss in the next section. For this example, it's enough to understand that there is no single representation of an employee but a number of events that tell the story of that employee from start to finish. In this story, an employee has been hired, had his pay grade changed and was then terminated. We use something called a query projection, which is a static view on the employee to show that employees current state at any given time. The domain does not represent the current state of the employee as of now but is just a series of events occurring on that employee over time. Suppose that HR has required a view that shows all non-terminated (active) employees. We will use JSON to express the contents of the event data for ease of reading. The reason for this is that different events of a single domain are usually written to the same event log/table so each one has a different footprint and this does not lend itself to tabular presentation.

Table 8.1 Events that have occurred on the employee aggregate

1	EmployeeHired({"id":"user1","firstName":"Sean", "lastName":"Walsh","payGrade":1,"version":0})
2	EmployeeNameChanged({"id":"user1","lastName":"Steven","version":1})
3	EmployeePayGradeChanged({"id":"user1","payGrade":2,"version":2})
4	EmployeeTerminated({"id":"terminationDate":"20150624","version":3})

Our read projection of this employee is indexed on the employee's id and contains all employees, although we will illustrate just the one here.

After the 3rd event, EmployeePayGradeChanged the projection appears as follows:

Table 8.2 The read projection of the employee *before* the termination has occurred

id	lastName	firstName	payGrade
user1	Steven	Sean	2

Ah but after all events through event number 4 the table would be empty since this user no longer exists in the eyes of any consumer of the projection upon event 4 becoming eventually consistent to this projection. Note that the event logs still exists even though the employee functionally does not due to the termination. This data is very valuable for use cases such as rehire but also for data retention for employment compliance purposes.

Table 8.3 The read projection of the employee *after* the termination has occurred

id	lastName	firstName	payGrade

...crickets

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

Because this query is constantly being updated in the background, it is always there, waiting to be read. This does away with so much of the pain suffered due to impedance mismatch, let's review that pain.

8.4.3 Dynamic Queries Not Needed - No More Pain

By using CQRS queries our data is aggregated asynchronously for us and is always there ready and waiting. The difference in user experience is sometimes staggering with the reduced latency of the CQRS approach, the screens render in the blink of an eye. What you see is what you get. There is no magic that happens and query time, the data is in a size and shape in the database and that size and shape is exactly what the client receives, although it will be marshalled to JSON, etc. This makes it very easy to support, track down anomalies and debug your applications without constantly having to dig through and debug code. Since we're beating CRUD to death let's give it one last kick in the teeth. In the world of CRUD. queries become expensive to run and maintain and sometimes require constant tuning. In the old, monolithic world it is common for clients to request aggregated domain data, that being domain data mashup across multiple domain contexts. Attempting to do this at runtime is ridiculous, we should know because we have done it in our past and it was the most significant source of problems. This pattern requires presentation to domain tier adapters that make multiple calls to the backend domain to create aggregated DTOs. Another pattern is to simply make SQL/NoSQL requests to the database directly from the presentation tier bypassing the domain entirely. In any case forgetting about the problems we mentioned above, if we design our systems like this they are closely coupled and cannot be distributed.

8.4.4 Relational vs NoSQL

Most of us have used relational (SQL) databases at some points in our careers and they have worked well. We can use SQL for event sourcing as well as the CQRS query side but the concrete column model (where every column and its type is known at design time and enumerated in a table) is inflexible compared to NoSQL. NoSQL allows the storage of documents containing any number of attributes that need not be known at design time making the construction of CQRS/ES systems much faster and more agile. With NoSQL one can simply serialize an object with it's varying fields to a table. The table may also contain multiple types of objects of differing size and shape, making it quite easy to store all the events of a domain type. With SQL you would have to know the columns ahead of time and constantly maintain migration scripts when your event or read stores change over time, lot's of overhead you can avoid by using any of the excellent open source NoSQL solutions available today.

You've learned about the distinct paths of CQRS and seen the distinct nature of the command and query sides but how do they relate? This is where event sourcing becomes a natural fit and sits atop CQRS nicely as we'll discuss in the next section.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

8.5 Event Sourcing

Event sourcing is making the source of all domain behavior being the accumulation of the events that have taken place on that domain. There is no "order" sitting in any database as part of the domain. The state of the order at any given time is the accumulated playback of the events. Before we go further, it is possible to use event sourcing independently of CQRS, although the two together are very complimentary as commands can result in events. We have seen large clients where the adoption of the message-drivenness is just too ponderous given the legacy systems and infrastructure and sometimes compromises must be made. In this chapter we will concentrate on the CQRS/ES combination.

8.5.1 What is an Event?

An event represents something meaningful in the domain has occurred and is usually in the past tense such as OrderCreated or OrderLineAdded, etc. Events are immutable in that they represent something that happened at a particular point in time and they are persistent, meaning stored in some event log such as a distributable database such as Apache Cassandra. At any point in time it is possible to witness the state of the domain by the ordered accumulation of the events through that point in time. This even works with deletes, which are events like anything else. An actual delete of something doesn't really occur as it would in CRUD, the event simply states that the domain object no longer exists at that point in time. Events are a cornerstone of being message driven and a great way of meaningful communication between microservices. Since it is possible to have a great number of events on some domains there is the concept of the *snapshot*. We'll also talk more about *replay* and *in-memory state* as well as *projections*.

As you can see in Figure 8.9, an order is merely a sum of its events over time.

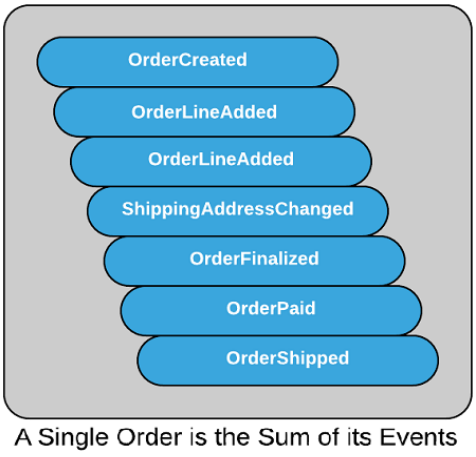

Figure 8.9 The events over time *are* the order.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

Event sourcing FTW!

In spring of 2014, many technical dignitaries were presenting at the PhillyETE conference. On that occasion, we went to dinner with a bunch of people from Typesafe, along with some others. Sean was sitting next to Dr. Roland Kuhn, head of the Akka team, when a guy walked in, still wearing his nametag. The name on the tag was Greg Young and we had the pleasure of meeting him and joining in some discussions."

Greg sat down, and I thanked him for his inspiration and the many hours of CQRS video materials he provided. We started talking about CQRS and Akka, and how we had implemented command sourcing for some of our microservices, for which he blasted us for the practice for a number of reasons, most notably use cases around one time only actions associated with commands such as a credit card transaction, that would never be repeated with the playback of events. At this point, Roland and I challenged Greg, and by the end of the conversation the decision was made by Roland and the Akka team to remove virtually all references to command sourcing from Akka Persistence documentation, and to embrace event sourcing only. Pretty cool!

SNAPSHOTS

Obviously if you have a domain with millions of events, materializing state on that domain would be non-performant and the answer there is snapshotting. Snapshotting involves a separate storage from the event log and is simply the periodic saving of the state of the entire domain. *Replay* is the full or partial rebuild of state based upon event history, on replay the snapshot is the first thing retrieved and then all subsequent events that occurred after that snapshot are replayed atop it. Let's us an energy industry example where various energy readings come into a domain representing a large industrial battery. Each second a reading comes in for charge and discharge kilowatts. The state of the battery includes its kilowatt hour capacity and these events add and subtract from that state. Every day a snapshot is saved to the database so that all events need not be replayed.

REPLAY AND IN-MEMORY STATE

Replay is ordered retrieval of the snapshots and events stored in their storage mechanisms. Implementing replay is as easy as doing a read of the latest snapshot from one table and then a query of all the events that have taken place since the snapshot and having the domain object apply them one by one. Akka Persistence has an elegant solution to this as replay and snapshotting are built in. One simply models the domain such as an order as a persistent actor. When the actor is instantiated, Akka Persistence automatically sends the snapshot and all the appropriate events as messages to the actor. From there it is a simple matter of building up private state within the actor from those messages. This state functions as a projection of the order and may be queried upon in a very performant and distributed manner. Replay is an important aspect to event sourcing in that there are no guarantees that all produced events will be received and successfully processed for every consumer. There is no true reliable messaging among so many moving parts and hence the need for replay to rebuild application state when in doubt. Another valuable application of replay is *seeding* of data. Microservices cannot normally be guaranteed to have the same up and down time as the other

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

services they interact with. Seeding provides the ability to "catch up" on all the events a service has missed upon coming back online but also when the new service is spun up for the very first time in an environment, replay to the rescue here.

8.5.2 It's all about Behavior

With event sourcing the events mimic real life and no behavior is lost. It is possible to answer questions at some future point in time that the business hasn't even asked. Events capture the true in-sequence behavior in the domain that map to real life. When we look at the concept of CRUD we see a completely non-business set of operations, create, read, update, delete. These are not meaningful behavior and thankfully we can now leave them behind in favor of explicitly expressing our domain via events.

8.5.3 Life Beyond Distributed Transactions

What are we going to do without our trusty ACID transactions, where an update to an "order" could also contain an update to "customer" as a single unit. These have been so convenient to use in order to ensure multiple operations, using ORMs and paying a very high penalty in terms of distribution. It turns out that these cross boundary transactions aren't necessary the majority of the time and service level agreements (SLAs) should be carefully thought through in the rare cases ordered transactions must occur as a single unit. In the majority of the cases we rely on eventual consistency across microservice boundaries in place of traditional transactions. Too often we have seen strong consistency the default in application design and when this is your go-to choice, you are choosing consistency over availability, a very expensive default. This is sort of like putting your hands up in the bank in case it gets robbed.

When we do have situations that require stronger consistency we can employ the saga pattern we discussed in chapter 4. This pattern is implemented using the process manager pattern in Akka. Using this pattern it is possible to perform a sequence of commands or other messages across different contexts in the form of an orchestration. The pattern is a state machine so in each state there is recovery logic in case of failure.

8.5.4 The Order Example Part 2

Next we will model our domain as a CQRS/ES command side, using an Akka persistent actor. You will see how to ensure consistency of the order domain by using versioning as well as examples of modeling the domain state, commands and events.

MODELING STATE

Since domain state is a function of events occurring over time it is usually necessary to model the most current state of the domain, exclusive of any consistency scheme, i.e. guaranteed latest state. You can approach this model using a distributed database such as Cassandra and using a *read projection,* which is a picture of the current state, stored in the database as we'll

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

discuss in section 6.5 but that state would always be eventually consistent and would not include the most current state guarantee.

There is a way to attain this and it is by using Akka Persistence to model your domain. Using a single persistent actor in an actor cluster to represent a single domain object, it is possible to have that actor contain the current state and be the single point of access to that domain object. When we use actors in this way we get caching for free as the actor can contain mutable internal variable(s) that it updates upon each new or replayed event. This mutable state is ok here because it is completely internal to the actor and therefore thread safe. This now will allow in-memory reads of the domain that may be used in orchestrations across domain objects etc. Reading the domain in this way is not pure CQRS but it is a helpful and cheap additional tool in the toolbox to use in a pinch.

In the code section below you will see how we can model state using an Akka persistent actor. The code is used to illustrate state management but there are other concepts as well that have either been covered in chapter 5 or will be delved into in later chapters as we build a real world application, consider this a taste of things to come.

Listing 8.2 Persistent Order Actor

```
package com.example

import java.util.UUID
import akka.persistence.PersistentActor
import org.scalactic._
import Accumulation._
import OrderType._

❶
object OrderActor {
 object OrderType {
   type OrderType = String
   val Phone = new OrderType("phone")
   val Web = new OrderType("web")
   val Promo = new OrderType("promo")
   val OrderTypes: List[OrderType] = List(Phone, Web, Promo)
  }
❷
 case object CommandAccepted

❸
case class ExpectedVersionMismatch(expected: Long, actual: Long)

❹
 case class CreateOrder(
   id: UUID,
   customerId: String,
   date: Long,
   orderType: OrderType,
   orderLines: List[OrderLine])

 // The add order line command.
 case class AddOrderLine(
   id: UUID,
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

```
   orderLine: OrderLine,
   expectedVersion: Long)

❺
 case class OrderCreated(
  id: UUID,
  customerId: String,
  date: Long,
  orderType: OrderType,
  orderLines: List[OrderLine],
  version: Long)

 case class OrderLineAdded(
   id: UUID,
   orderLine: OrderLine,
   version: Long)
}
❻
class OrderActor extends PersistentActor {

 import OrderActor._

 override def persistenceId: String = self.path.parent.name + "-" + self.path.name

❼
 private case class OrderState(
   id: UUID = null,
   customerId: String = null,
   date: Long = -1L,
   orderType: OrderType = null,
   orderLines: List[OrderLine] = Nil, version: Long = -1L)

 private var state = OrderState()

❽
 def create: Receive = {
    case CreateOrder(id, customerId, date, orderType, orderLines) =>
      val validations = withGood(
        validateCustomerId(customerId),
        validateDate(date),
        validateOrderType(orderType),
        validateOrderLines(orderLines)
      ) { (cid, d, ot, ol) => OrderCreated(UUID.randomUUID(), cid, d, ot, ol, 0L) }
      sender ! validations.fold(
        event => {
          sender ! CommandAccepted
          persist(event) { e =>
            state = OrderState(event.id, event.customerId, event.date, event.orderType,
       event.orderLines, 0L)
               ❾
            context.system.eventStream.publish(e)
            context.become(created)
          }
        },
        bad  =>
          sender ! bad
      )
  }
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

```
⑩
def created: Receive = {
   case AddOrderLine(id, orderLine, expectedVersion) =>
     if (expectedVersion != state.version)
       sender ! ExpectedVersionMismatch(expectedVersion, state.version)
     else {
       val validations = withGood(
         validateOrderLines(state.orderLines :+ orderLine)
       ) { (ol) => OrderLineAdded(id, orderLine, state.version + 1) }
       .fold(
         event => {
           persist(OrderLineAdded(id, orderLine, state.version + 1)) { e =>
             state = state.copy(orderLines = state.orderLines :+ e.orderLine, version =
     state.version + 1)
             context.system.eventStream.publish(e)
           }
         },
         bad => sender ! bad
       )
     }
 }

⑪
override def receiveCommand = create

⑫
override def receiveRecover: Receive = {
  case CreateOrder(id, customerId, date, orderType, orderLines) =>
    state = OrderState(id, customerId, date, orderType, orderLines, 0L)
    context.become(created)
  case AddOrderLine(id, orderLine, expectedVersion)             =>
    state = state.copy(orderLines = state.orderLines :+ orderLine, version = state.version +
      1)
}

def validateCustomerId(customerId: String): String Or Every[ValidationFailure] =
  if (Option(customerId).exists(_.trim.nonEmpty))
    Good(customerId)
  else
    Bad(One(InvalidCustomerId(customerId)))

private def validateDate(date: Long): Long Or Every[ValidationFailure] =
  if (date > 0)
    Good(date)
  else
    Bad(One(InvalidDate(date.toString)))

private def validateOrderType(orderType: OrderType): OrderType Or Every[ValidationFailure] =
  if (OrderTypes.contains(orderType))
    Good(orderType)
  else
    Bad(One(InvalidOrderType(orderType)))

private def validateOrderLines(orderLines: List[OrderLine]): List[OrderLine] Or
      Every[ValidationFailure] =
  if (!orderLines.isEmpty)
    Good(orderLines)
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

```
    else
      Bad(One(InvalidOrderLines(orderLines.mkString)))
}
```

1. Companion object for the order actor for neat encapsulation
2. A simple Acknowledgment object signifying that command is accepted
3. Any command must specify the last known version of the order
4. The create and add order line commands
5. The order created and order line added events
6. This actor would only ever have a single instance in the cluster
7. This is the internal state of the actor, changed each time a command is processed
8. This is the receive we'll use when the order has not yet been created, this would result in the initial state of the order
9. With successful validation we can now store the event and perform any side effecting logic such as emitting the event to interested parties via the event stream, transitioning to our new created command handler and updating the state
10. The created handler handles commands other than creation. We'll just illustrate the add order line here
11. Set initial command handler to our create partial function we defined above
12. Receive recover simply builds up state from the events that have taken place in the past. No need to validate here. This is executed when the actor is instantiated by the cluster

As you can see, using Akka Persistence there are elegant ways to handle all aspects of CQRS/ES domain design, including clean and non-breaking validation and uncompromising data consistency. Next we will look at consistency concerns that exist in the new world of CQRS/ES.

8.5.5 Consistency Revisited

With the disconnect that now exists between our read and write sides as they are now separate concerns, both are now message driven using events and consistency becomes an important aspect to consider. Consistency describes the guarantees of how distributed data is propagated across distributed systems and prescribes in what manner data will be seen across partition boundaries. The three types of consistency used in computing are *strong consistency, eventual consistency and causal consistency*, we'll talk about all of them next.

STRONG CONSISTENCY

Strong consistency guarantees that all data will be seen across all partitions, at the same time and in the same sequence. Strong consistency is the most expensive and will easily prevent you from distributing your applications and surprisingly enough has been the go-to method of consistency in system development for some time. When we use database transactions with ORMs such as Hibernate we get this level of consistency but in most cases there are no requirements that dictate the need for this. Any use of strong consistency should be thought through carefully because the only way to support this level of consistency is with close coupling of the related systems, which can quickly lead to a monolith. It is never recommended to use strong consistency across distributed boundaries, the cost is just too high.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

EVENTUAL CONSISTENCY

Eventual consistency is the cheapest and easiest to implement and you should strive to make this your consistency mode of choice. With this method your data will eventually become consistent across partitions but the timing and ordering are not guaranteed. Make this your first and hopefully only consistency model. The typical flow for eventual consistency is for the CQRS command side to emit an event using some bus, Akka cluster for example. On the read side of another microservice there is an event listener actor that subscribes to that event over the Akka event stream. Upon receipt of any such message that actor will determine how it affects the read projection(s) in which the actor oversees and mutate those projections accordingly to match the latest state of the domain(s).

CAUSAL CONSISTENCY

Causal consistency is the second most expensive consistency model and as such should also be avoided whenever possible, although it is more likely you will run into use cases that require it. Causal consistency guarantees that all partitions see the same data, in the same sequence *but not at the same point in time.* Think of causal consistency as an orchestration across your microservices/domains. There is an excellent pattern for this using Akka called the Process Manager as we will show next in the order example part 4.

8.5.6 Retry Patterns (Reference Reactive Design Patterns)

In a distributed, reactive application it is difficult and most likely impossible to have 100% reliable messaging but we can do our best to make messaging as reliable as possible by utilizing durable messaging using Kafka or RabbitMQ or utilizing the delivery semantics in Akka cluster as we'll discuss next.

Using akka messaging semantics it is possible to retry message delivery between actors until the message has been known to be delivered to the recipient. The follow section describes these semantics.

At-least-once - this is the least expensive for retry and requires the sending side to maintain storage of outstanding messages. With at least once the sender will always try to redeliver a message until a confirmation of the receipt is received from the recipient. With this method it is possible to deliver a message more than once since it is possible that the sender is receiving but having difficulty sending the confirmation response.

Exactly-once - This is the most expensive means of messaging and requires storage on both the sending and receiving sides. With exactly once a message will constantly be retried until it is received and the recipient is guaranteed to process it only once.

In summary, always use at-least-once if you can. Finally, we will discuss command sourcing vs event sourcing.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

8.5.7 Command Sourcing vs. Event Sourcing

Command sourcing is when the commands are logged as the source of record rather than the events and event sourcing is of course logging the events. There are problems with command sourcing in that commands do not always result in a change of domain state and are rejected. Why would one want to have a rejected command as a central part of the domain? It doesn't make much sense unless there is a clear requirement to do so for audit purposes and in such a case the events should be logged as the source of record as well. Another problem with command sourcing is that replay is an important part of CQRS/ES. Commands may trigger side effects such as a one time credit card transaction whereas the events occur after that logic and may be used to replay and rebuild domain state without any side effects. Replay of commands would be complex and problematic and should be avoided. In short, stick to event sourcing if you can.

8.6 Summary

- CQRS allows an easy way to be non-monolithic as reads are separated from writes, typically as completely separate applications.
- CQRS provides an easy way to bulkhead our applications.
- We saw that the relational databases we typically use do not scale due to transactionality.
- Event sourcing allows a nice basis to be message driven and ensures no historical behavior is lost.
- Akka provides an elegant CQRS and event sourcing solution right out of the box.
- Think carefully about your consistency models and always lean towards the less expensive option of eventual consistency.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

9

A Reactive Interface

This chapter covers:

- **Headless APIs**
- **Http, XML and REST**
- **JSON**
- **Expressive APIs**
- **Play**
- **Akka Http**

Now that we have seen how to build a focused microservice containing CQRS command or query functionality how do we get at it? How does a client make use of these shiny, new reactive applications? We now need an interface for our service to allow its rubber to meet the road. The service interface (or API) is the way for clients as well as other services to interact with that particular service. In this chapter we will show how this is done using the most common reactive tools and standards.

9.1 What is a Reactive Interface

9.1.1 The API Layer

The reactive interface is treated as the outer layer of the service and the overwhelming go-to interface of choice these days is REST over HTTP. Rest or *Representational State Transfer* in a lightweight interface, typically utilizing JSON, *Javascript Object Notation* as the payload of choice but XML, *extensible markup language* is sometimes used as well but is considered more verbose. This chapter focuses on servicing restful clients such as UIs and does not cover

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

streaming as an interface but it bears mentioning that streaming is becoming more common as a way to connect services and systems and the subject can easily fill an entire book.

Reactive applications are reactive on all levels, including the API layer. The interface should be non-blocking and responsive. As it sits upon a reactive, CQRS service it should operate whizzy fast since incoming requests from clients are either commands that are accepted in real time by the service and responded to immediately, as well as reads of query data that requires no manipulation and is also very fast, almost immediate.

This chapter will show you how to add a restful interface to your services and show some basics such as authentication, logging and bootstrapping. We'll start by looking at headless APIs, a term describing the need for each and every service to contain its own interface. Figure 9.1 shows a couple services containing headless APIs.

Though we concentrate on restful APIs in this chapter it should be noted that sometimes you can get away with serialized object communication between services, especially when those services on running on the same Akka cluster. If you can do this go for it and eliminate the overhead of json marshalling in favor of friendly case classes.

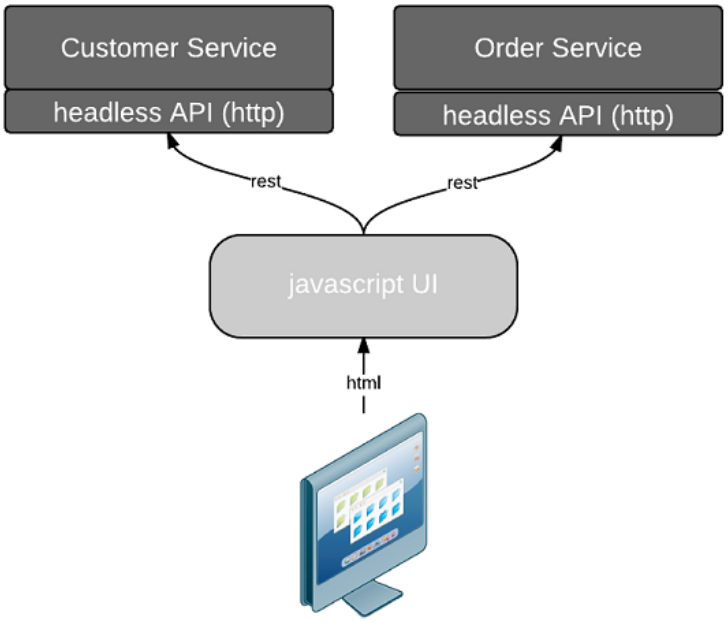

Figure 9.1 Customer and Order services with headless APIs.

9.1.2 The Headless API

It's very important that each service can be deployed and scaled independently, each having it's own interface to the outside world. This interface provides access to the specific

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

functionality of the service but does not prescribe or is in any way opinionated as to the way the service data should be displayed or consumed. This separation of the consumption of the service from how the service data will be utilized is made possible with a headless API.

In the past we usually built applications that included the graphical user interface (GUI or just UI). The UI design was closely coupled to the business logic implementations and in many cases contained embedded business logic. A headless API is one that presents a specific area of functionality as an interface, such as REST and leaves the business of UI presentation to the user interface designers and codes, using javascript libraries such as Angular.js or Backbone.js. An example of a way of the past might be a monolithic application that served up all pages using Java Server Pages (JSP), Java Server Faces (JSF) or similar from the application itself. The headless API decouples the UI from the application, which frees up front end developers to provide a truly pleasurable and interactive experience for users without regard to solving business problems. Likewise the API layer, the application, has no concern or interest in how its raw data is presented. The UI developers simply address the available service APIs to build the best and most responsive displays. The backend simply provides a lightweight restful interface as shown in figure 9.2.

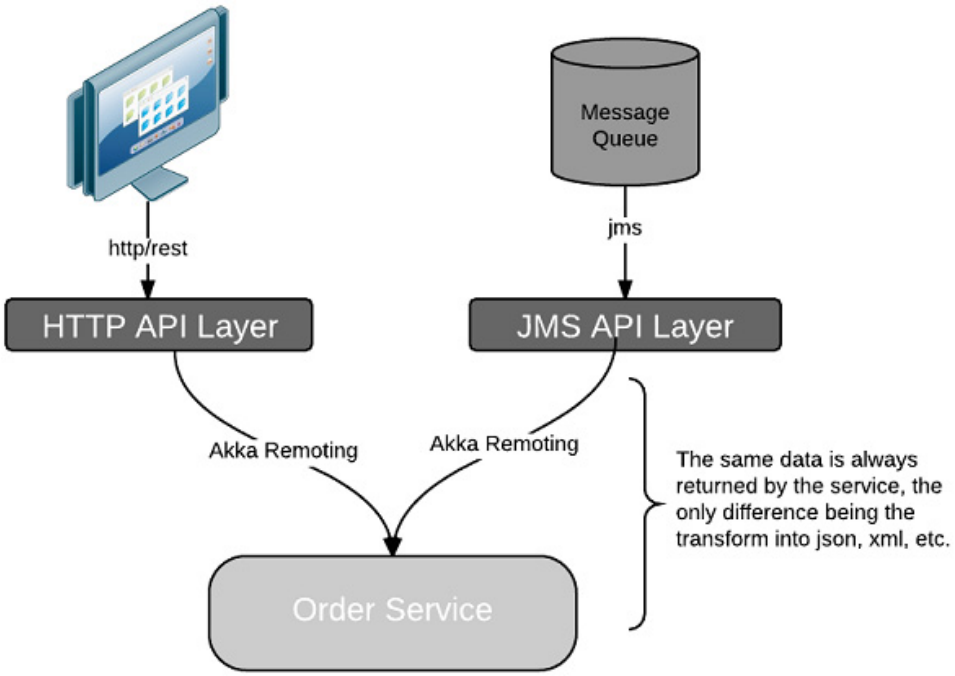

Figure 9.2 Incoming requests are converted to and from http and jms

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

As you can see, the microservices have a distinct restful interface available for any consumer, such as the javascript client depicted. Next we'll talk in more detail about REST and JSON and what how the interface typically appears.

You should look upon the restful or any other API as a layer that sits atop your service logic as a very dumb adapter to that service. It should be very simple to pick one service interface, say using message queuing instead of http with no difference in how that service performs its duty. Therefore it is of the utmost importance not to include any business logic in the API layers. Simply transform incoming requests to requests that the service layer understands, such as a message to a service actor using a case class as the payload. The response from the service is likewise transformed back to the API protocol (http/rest/json) before the ultimate response back to the client. The following diagram shows this in practice.

9.2 Expressive Restful Interfaces

REST and JSON have become overwhelmingly the protocol of choice for service interfaces using HTTP. REST gained popularity over the competing SOAP (Simple Object Access Protocol) and XML (Extensible Markup Language). SOAP is very verbose and complex and XML itself is quite verbose. REST and JSON offers a simpler and easier to use solution. SOAP (Simple Object Access Protocol) was anything but simple and required a strict interface described with XML, contained in schema documents in order to access a service. REST is much simpler, whereas if there is an available restful service and you know how to call it, you can have at it. The payload within the JSON is freeform and may contain whatever attributes the server desires, which results in a great deal of flexibility. The following sections will illustrate the differences between the older use of XML vs JSON, more commonly used today. We'll also see how best to have the most descriptive and intuitive interfaces using REST and JSON.

9.2.1 JSON vs XML

An example of a json payload for getting an order would like something like this:

```
{“orderId”: “12345”, “status”: “Shipped”, “items”: [{“itemId”: 321, “name”: “sink”},
       {“itemId”: 987, “name”: “faucet”}, {“itemId”: 756, “name”: “drain”}]}
```

The JSON above is very readable once you understand a few concepts.

- Any particular object such as *order* or *item* is wrapped in braces.
- Collections of objects are wrapped in brackets.
- All attribute names are wrapped in quotes.
- Attribute values are wrapped in quotes for strings but not for numeric values.

Now let's compare the JSON above with its XML counterpart.

```
<order><id>12345</id><status>Shipped</status><items><orderItem><itemId>321</itemId><name>sink
       </name></orderItem><orderItem><itemId>987</itemId><name>faucet</name></orderItem><orde
       rItem><itemId>756</itemId><name>drain</name></orderItem></items></order>
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

You can see that the XML is quite a bit more boilerplate and harder to read, where if one has a decent grasp of JSON then the snippet above can be easily understood at a glance. Imagine how unwieldy a complex data structure in XML can become.

HTTP headers are used in both requests and responses, describing the payload type as well as other metadata. When using JSON you must set the "Content-Type" header to a value of "application/json".

9.2.2 Expressive Restful Interface URLs

A good API in any language, clearly expresses the intent and capabilities for any given call against the API. HTTP has been around for a while and prescribes clear verbs and response codes for any given call, we can use these to standardize our restful APIs as much as possible. We'll first talk about the verbs you will use most frequently.

- *POST* - A post signifies that something new is being created so with something like adding an new customer order you would use a post.
- *PUT* - A put means you wish to do an in place modification of something that already exists, such as changing an order ship date.
- *GET* - A get is a simple read of one or more things such as orders. A get can contain query parameters (`/orders?id=order1`) or URL parts (`/orders/order1`). The latter URL style would be preferred.
- *DELETE* - When you wish to delete something such as a single order.

It should be noted that when we issue commands to command side services, we only use POST. In fact CQRS applications only use POST or GETS.

The response codes are important and should be used consistently for common purposes. These include:

- *201/Created* - The result of a successful post or put.
- *202/Accepted* - This signifies that the request will be processed but does not guarantee the outcome. This is the code used for the command side to accept commands and make a best effort to process those commands.
- *200/Ok* - Usually returned for successful gets.
- *400/BadRequest* - The URL is incorrect or a POST or PUT contained malformed or incorrect data. This is the code we use for returning failed command validations along with the JSON formatted reason(s).
- *401/Unauthorized* - You do not have access to the URL.
- *404/NotFound* - The GET results in no returned data found on the server.
- *500/InternalServerError* - This is the least friendly sort of response and should be avoided if at all possible but as is the nature of reactive and all applications, things sometimes fail!

There are many HTTP response codes and probably many more internet references to them, the ones above are those you will most likely encounter.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

9.2.3 Location

In practice, for crud type web applications, the URL for posting, putting and deleting (as well as some others) will be the same with the only difference being the verb used on that URL. The best practice is to return the "Location" HTTP header upon a successful post. The value for that header is the URL representing the entity created in order for the client to be able to access it at a later point in time. For an order it would look like:

```
"Location": "/orders/order1"
```

Obviously the client would have knowledge of the server and would append the location to the server HTTP address. Returning just the relative URL as we have done above is the best practice as the instance of the service need not necessarily have knowledge of the actual host name or IP address leading to it, which could change at any time.

Now that we have a familiarity with the best and most expressive use of REST let's look at some reactive restful library choices that are available.

9.3 Choosing Your Reactive API Library

There are a great many Scala as well as Java libraries to serve up your restful HTTP services. We will talk about the the most likely candidates, Play and Akka-Http and now Lagom. Lagom is quite new but appears to be the go-to framework for restful applications working together in a clustered environment and firmly embraces CQRS and event sourcing in an opinionated fashion.

9.3.1 Play

The Play open source framework; https://www.playframework.com is mature and powered many an enterprise at high levels of scale. We have had great success using Play in high scale and high throughput environments. At our large diet and wellness client, using a load balanced set of four instances of Play services, we were able to migrate massive user data into the new generation of services using Play, backed by a Cassandra cluster, and saw around 12,000 POST requests per second, not bad!

What is nice about Play is it offers a slightly opinionated way or building a complete, production ready service. Play takes away the hassle of inventing your own wheels for application layout, logging, configuration, localization and health monitoring. Play is usually used to build stateless applications where these may be any number of Play instances with a large, clustered database behind. In this scenario the application state is contained in the database and must be read *before* being acted upon. This has implications for causal consistency concerns (when atomic, ordered operations must be performed on the domain), so think this through very carefully before going stateless. It is possible to use play coupled to Akka cluster to have atomic, in-memory state in the middle tier but requires some work and you are probably better off using Lagom for such efforts. Lagom's cluster integration is by default and stateful. Figure 9.3 shows the typical Play application layout.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

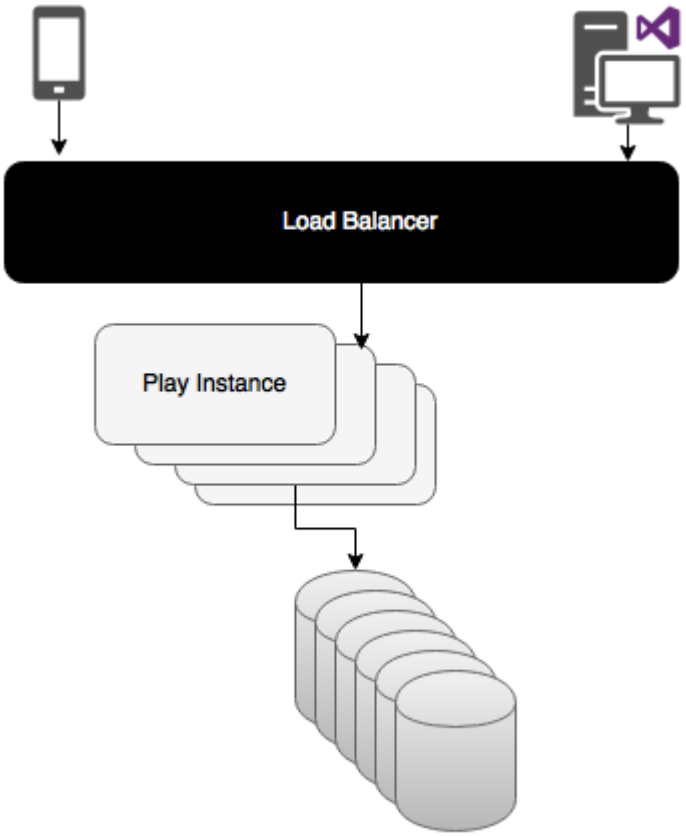

Figure 9.3 Play runtime footprint

You will see from above the architecture is fairly simple with a load balancer handing off requests across all of your Play instances. Each of these instances are backed by a large database cluster such as Apache Cassandra.

9.3.2 Opinionation

This deserves its own mention. As a software craftsman at startups where I was building new things I would have shied away from opinionated frameworks as a rule to give me the utmost flexibility in choosing every aspect of my stack. We were building a complex IoT (Internet of Things) energy application and I do believe my views were warranted for that case but also in cases of teams building out libraries and software products of their own where flexibility is key. Having said that, the larger the team and the more enterprise the environment, the most likely it is that opinionation is a benefit.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

The opinionated nature of Play and for that matter Lagom is usually a good thing for most teams. Using a less opinionated library such as Akka-Http there is the very real possibility of a team building the same kinds of services in very different ways and believe me we have seen it. Play keeps it simple, perhaps with a slight cost in flexibility but we've found no issues using the very simple routes and controllers.

9.3.3 Application Structure

Play dictates a default folder layout that may be overridden but it is not recommended. It's really nice for a developer with Play experience join a team and instantly recognise the project structure, reducing onboarding headaches. The application structure appears as follows.

`app`	→ **Application sources**
`└ controllers`	→ **Application controllers**
`└ models`	→ **Application business layer**
`build.sbt`	→ **Application build script**
`conf`	→ **Configurations files and other non-compiled resources (on classpath)**
`└ application.conf`	→ **Main configuration file**
`└ routes`	→ **Routes definition**
`project`	→ **sbt configuration files**
`└ build.properties`	→ **Marker for sbt project**
`└ plugins.sbt`	→ **sbt plugins including the declaration for Play itself**
`logs`	→ **Logs folder**
`└ application.log`	→ **Default log file**
`test`	→ **source folder for unit or functional tests**

Figure 9.3 Play application structure

There are other folder structures in Play besides the ones above but those are mostly dealing with UI presentation and as we have discussed, we're not interested in serving up a UI as part of our service, just a headless API. This is the structure you will most likely use.

9.3.4 A Simple Route

The first thing you need to do to create any route is define your route in the routes file in the conf folder. Here we are going to build a simple route to handle a Http client's request to get all customer orders. The routes file will appear as the following code listing:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

Listing 9.3 Get Orders Route File

```
# Routes
# This file defines all application routes (Higher priority routes first)
# ~~~~

(1)
GET     /orders controllers.OrderController.getOrders
```

(1) You specify the URL that will handle the request to get all orders, /orders. You then specify the controller and controller function to handle the request as we'll be seeing next.

We can also define a URL parameter using a colon followed by a descriptive name, these parameters are strings by default but to consume them as another type, such as an int, you would just specify the type to the right, such as getOrders(customerId: Int). Play does the mapping automatically.

You see above we have an OrderController but what is this madness? We're doing top down design here so let's create that next layer: the controller. The controllers live in, you guessed it, the controllers package. The order controller would appear as follows:

Listing 9.3 Order Controller

```
package controllers

import java.util.UUID
import javax.inject._

import akka.pattern.ask
import play.api.mvc._
import models._
import akka.actor.ActorRef
import akka.util.Timeout
import play.api.libs.json.Json
import services.OrderService.GetOrders

import scala.concurrent.ExecutionContext
import scala.concurrent.duration._

class OrderController @Inject() (orderService: ActorRef)(implicit ec: ExecutionContext)
       extends Controller {

  (1)
  implicit val timeout: Timeout = Timeout(5.seconds)

  (2)
  implicit def orderFormat = Json.format[Order]

  (3)
  def getOrders = Action.async { _ =>    (4)
    (orderService ? GetOrders)
      .mapTo[Seq[Order]]                 (5)
      .map(res => Ok(Json.toJson(res)))  (6)
  }
}
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

1. The timeout is used against the ask (?) of the actor for the customer orders. This timeout may also be configured in application configuration.
2. When using play json you must minimally create this implicit writes so play knows how to output to json. This is the most common and simple way but complex json output is also supported.
3. This is the function that maps to the routes file for get-orders. The customer UUID is automatically dealt with and the argument unmarshalled to the UUID by play.
4. The implicit request object is passed in here but we aren't using it so ignore it for the get.
5. The ask of the actor is not typesafe so we must specify what response type is expected so it may be tranformed.
6. The compiler knows that there is a json marshaller in scope for that return type. The sequence of orders is returned and wrapped in the HTTP ok (202).

9.3.5 Non-Blocking Service Interfaces

The API should be non-blocking from start to finish, meaning the only open socket/handle is held by the client. All service, database and heaven forbid, real time service calls are done with non-blocking futures, whereas the results are processed asynchronously and when complete. What are futures? Futures are scheduled computations that are to be carried out at some point from the present onward depending on resource availability. The computations that deal with the results of those futures are essentially "parked" until the future has done its work. All threads associated with creating the future and dealing with the result are freed up until the future gets access to the CPU and associated resources in order to complete its computation. Upon completion the callback logic is allocated thread(s) to reactively complete their tasks utilizing the future results. There are a finite number of threads available to your processes and keeping those threads available allows your computing resources to scale to the maximum before having to add more hardware. It is possible to be completely non-blocking from the client using web socket technology to achieve callbacks similar in the way computational results returned as future callbacks in the backend API but in practice we've seen very little of this and typically the client blocks until the response is received. The client (mobile device, pc, etc) blocking is of much less concern than blocking of any kind is avoided at the server as the blocking will only occur on that device. Remember from chapter one that the Universal Scalability Law dictates that blocking of any kind will affect scale and will make your application less reactive. Remember that if you are blocking you will only be able to add more hardware to a point, and when you hit that point, any further hardware will show *diminishing* returns and will *reduce* scale.

We will now look at non-blocking in practice using order creation as an example. Here, the client sends us a new order in JSON format. The service will validate the CreateOrder command, and issue the command to the order service where we stop and simply println an OrderCreated event. If we were going further in this chapter and covering more than just the reactive interface you would see an order persistent actor instantiated as well as the persistence of the OrderCreated event to the database. The following is our simple order domain as represented in a scala package object:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

Listing 9.3 Order Domain Objects

```
case class OrderLine(
  itemId: UUID,
  quantity: Int)

case class Order(
  id: UUID,
  customerId: UUID,
  orderLines: Seq[OrderLine])
```

Next the order service:

Listing 9.3 Order Service

```
object OrderService {
  case object GetOrders

  case class CreateOrder(ustomerId: UUID, orderLines: Seq[OrderLine])
  case class OrderCreated(order: Order, timestamp: Long)
}

class OrderService extends Actor {

  val fakeOrders = Seq(
    Order(UUID.randomUUID(), UUID.randomUUID(), Seq(OrderLine(UUID.randomUUID(), 9))),
    Order(UUID.randomUUID(), UUID.randomUUID(), Seq(OrderLine(UUID.randomUUID(), 3)))
  )
    (1)

  override def receive = {
    case GetOrders          => sender ! fakeOrders
    (2)
    case CreateOrder(order) =>
      println(s"generating event: ${OrderCreated(newOrder(cid, lines), new Date().getTime)}")
  }
    (3)

  def newOrder(customerId: UUID, orderLines: Seq[OrderLine]): Order =
    Order(UUID.randomUUID(), customerId, orderLines)
    (4)
}
```

(1) The data returned is just dummy test data.
(2) The fake orders are returned to the message sender on GetOrders.
(3) Upon receiving the CreateOrder command we simulate the OrderCreated event in a println.
(4) Helper function to create an order, generating a unique id.

Listing 9.3 Play Routes File

```
GET     /orders controllers.OrderController.getOrders
POST    /orders controllers.OrderController.createOrder
(1)
```

(1) Here we simply specify the routes we handle and map them to our controller functions.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

Finally we will show the fully implemented order controller with Json marshalling.

Listing 9.3 Order Controller

```
class OrderController @Inject() (orderService: ActorRef)(implicit ec: ExecutionContext)
       extends Controller {

  implicit val timeout: Timeout = Timeout(5.seconds)
   (1)

  implicit def orderLineFormat = Json.format[OrderLine]
  implicit def orderFormat = Json.format[Order]
  implicit def createOrderFormat = Json.format[CreateOrder]
   (2)

  def getOrders = Action.async { _ =>
    (orderService ? GetOrders)
      .mapTo[Seq[Order]]
      .map(res => Ok(Json.toJson(res)))
  }
   (3)

  def createOrder: Action[JsValue] = Action.async(parse.json) { request =>
    request.body.asOpt[CreateOrder].foreach { o =>
      orderService ! o
    }

    Future.successful(Accepted)
  }
   (4)
}
```

(1) Play responds automatically with configured error response upon timeouts in communicating with the order service, ie InternalError.
(2) These formats enable the automatic marshalling and unmarshalling to json.
(3) Returns all orders asynchronously and wraps in Ok response, the 200 HTTP code.
(4) Create order attempts to extract the CreateOrder command and optimistically responds with Accepted. There are lots more things you can do here such as validation and possibly responding with BadRequest but we're keeping it simple.

You can see that building fully featured, reactive and restful applications is quite simple using Play. Next we will look at the Akka-Http and it's more functional style.

9.4 Akka-Http

At this point we'll take a look at the other restful toolkit that is a bit different and more functional than Play. Akka-http although also a Lightbend open source product but comes from a very different origin. Originally created by the spray.io team and labeled Spray, it became very popular and was purchased by Lightbend and rewritten by the Akka team to better fit the overall Akka framework. Where Play is very expressive with the strict definitions of routes, which in turn point to functions to handle all URLs, Akka-http treats a service interface as more of a chain of functions. Not only is a particular route a function chain but completely

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

separate routes may be defined atomically, according to their area of interest and then themselves chained together at the top level of the application. With Akka-http, gone is the opinionated ways of the Play framework and now you are open to create your rest-based microservice in any way you see fit. It stands mentioning that the use of either one of these restful toolkits depends on the capabilities of the team as well as personal preference. Like Play, Akka-http includes a runtime and can act as a container or you may use the container of your choice and just utilized it as a library.

9.4.1 A Simple CQRS-esque Akka-Http Service

At this point we'll dive into a complete web service using Akka-Http and CQRS/ES semantics. We will stop short of implementing any persistence or true service behavior as it is outside of the scope of this chapter but will utilized a CQRS command for demonstration purposes. The following is a very simple order domain. Note the use of java UUIDs for unique domain identifiers, you'll see a lot of them! The order and its order lines may appear unfriendly with these cryptic identifiers for customer, item, and order id but remember; these are true domain objects and exist solely to satisfy the the command and event behavior. There are potently views on the CQRS query sides that can present orders with other friendly attributes such as customer name and address, etc.

Now let's take a look at the order service. The service is an actor that receives CQRS commands as it's message interface. In the real world this would be a service utilizing persistent actors to represent the orders and the events would be persisted to an event store (database). In the following code we will show the order service implemented as a simple Akka actor with a companion object, containing its *CreateOrder* command as well as the *OrderCreated* event. Once again our order service, which will have the same behavior as the earlier Play example.

Listing 9.4 Order Service

```
object OrderService {
  case object GetOrders

  case class CreateOrder(ustomerId: UUID, orderLines: Seq[OrderLine])
  case class OrderCreated(order: Order, timestamp: Long)
}

class OrderService extends Actor {

  val fakeOrders = Seq(
    Order(UUID.randomUUID(), UUID.randomUUID(), Seq(OrderLine(UUID.randomUUID(), 9))),
    Order(UUID.randomUUID(), UUID.randomUUID(), Seq(OrderLine(UUID.randomUUID(), 3)))
  )

  override def receive = {
    case GetOrders          => sender ! fakeOrders
    case CreateOrder(order) =>
      println(s"generating event: ${OrderCreated(newOrder(cid, lines), new Date().getTime)}")
  }
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

```
  def newOrder(customerId: UUID, orderLines: Seq[OrderLine]): Order =
    Order(UUID.randomUUID(), customerId, orderLines)

}
```

At this point we will show the HTTP interface for our service implemented in Akka-Http. The trait contains only the logic it needs to satisfy the interface and interact with the service layer. The trait specifies the resources it needs implemented by the wrapping implementation, in this case the runtime object we'll show next. You'll notice that this route only handles URLs prefixed with /order. The get and post are implemented inside of that block. The post assumes the JSON entity will be a new order and will deserialize it as such. Also in the post, the preferred CQRS command semantics are to accept the command and return the status code reflecting receipt of the command only. The CreateOrder command is then invoked on the order service. There are no guarantees that the order will be accepted. In practice we would validate the order first and possibly returned failed validations with a bad request status but we're keeping it simple in this example.

Listing 9.4 Orders Route

```
trait OrdersRoute extends SprayJsonSupport with AskSupport {

  def orderService: ActorRef
  implicit def ec: ExecutionContext
  implicit def timeout: Timeout
  ❶

  implicit val DateFormat = new RootJsonFormat[java.util.UUID] {
    lazy val format = new java.text.SimpleDateFormat()
    def read(jsValue: JsValue): java.util.UUID =
       UUID.fromString(jsValue.compactPrint.replace("\"", ""))
    def write(uuid: java.util.UUID) = JsString(uuid.toString)
  }
  ❷

  implicit val orderLineFormat = jsonFormat2(OrderLine)
  implicit val orderFormat = jsonFormat3(Order)
  implicit val newOrderFormat = jsonFormat2(NewOrder)
  ❸

  val ordersRoute =
    path("orders") {
      get {
        complete((orderService ? GetOrders).mapTo[Seq[Order]])
        ❹
      } ~
      post {
        entity(as[NewOrder]) { newOrder =>
          orderService ! CreateOrder(newOrder.customerId, newOrder.orderLines)
          complete((StatusCodes.Accepted, "order accepted"))
        }
      }
    }
}
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

1. Here we state our resource needs of the order service, execution context and timeout. These will need to be implemented in the bootstrapping in the next section.
2. Here we define the JSON format for the UUIDs, which is a bit tricky.
3. The rest of the order if quite simple in terms of the JSON protocol.
4. The path directive here will result in any URL to /orders will be handled in the inner get or post routes.
5. The get of all orders is implemented as an ask to the service actor. The ask assumes the service will respond to sender with a sequence of orders. Akka-http will utilize the spray-json marshalling of the order and will present it as a json collection.

In this final section we'll show how to wrap it all up in a runtime and actually start up and run the application. Here we provide concrete implementations of all the abstract requirements in the trait, the order service, timeout and execution context. This runtime object will run in an SBT terminal until any key is pressed. This technique is borrowed from the Akka documentation and is as good an elegant way to stop the process as any! We also point our "route" to the orders route we set up in the trait we extended. This is a nice way of doing it because if you want to extend multiple route traits you can concatenate them here as a single route to be run. This results in a clean, modular design of your individual routes.

Listing 9.4 Order Web Service

```
object OrderWebService extends OrdersRoute {

  implicit val system = ActorSystem("order-system")
  override implicit val ec: ExecutionContext = system.dispatcher
  override implicit val timeout = Timeout(5.seconds)
  override val orderService = system.actorOf(Props(new OrderService))

  def main(args: Array[String]) {

    implicit val materializer = ActorMaterializer()
       (1)

    val route = ordersRoute

    val bindingFuture = Http().bindAndHandle(route, "localhost", 8080)
       (2)

    println(s"Serving from http://localhost:8080/\nPress any key to kill...")
    StdIn.readLine()
    bindingFuture
      .flatMap(_.unbind()) // trigger unbinding from the port
      .onComplete(_ => system.terminate()) // and shutdown when done
  }
}
```

1. We set up a reactive stream flow on our actor for back pressure against faster or slower consumers.
2. The host and port is bound to the Http process and run.

The sample code for this chapter includes everything above and does indeed run. We use postman to test this out and here are the payloads to play with. Both use the URL of *http://localhost:8080/orders.*

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

- The get route has no body, be sure to set the *Content-Type* Http header to *application/json*.
- The post route can utilized the following json payload. Notice that the client only provides the customer and order lines, the orderId is generated by the service and results in a full order:

```
{
    "customerId": "e9adbed8-dae1-4d5b-92f0-2f056d3e4195",
    "orderLines": [
      {
        "itemId": "050dea2d-448b-4ae6-a128-a1d381e396d2",
        "quantity": 9
      }
    ]
}
```

As you can see Akka-Http is very functional and as such can be as concise or spaghetti-like as you choose to make it. We do recommend that to make it as concise as possible, associate any single route to a specific CQRS command or query side. Last and certainly not least we will look at Lagom for your restful service needs.

9.5 Lagom

Lagom is a complete open source solution to not only build web interfaces but fully reactive, scalable applications. Lagom was built as a response to the widespread migration of monolithic applications to microservices and to support large scale. The word Lagom is Swedish and means "just right", which is a nod to how one should size a microservice, they are not small or large but sized appropriately for their function. Lagom is also an answer to difficulty of properly designing fully reactive application stacks and as such is highly opinionated. In Lagom's view and indeed our own view, CQRS and Event Sourcing are the best way to build these systems at scale. Lagom takes this even further by providing Cassandra database and Kafka messaging support for ease of development by default and embraces these as great choices for storage and distributed pubsub of events. Lagom is built atop years of reliability baked into Play and Akka but provides abstractions to make the difficult, less difficult. Just like Play, in Lagom you can have code hot swapping right in your IDE so there are no costly compile/deploys between code changes. Even with the goal of making reactive applications as easy as possible, they are still not easy or simple. The fact of the matter is that the way we need to build applications today, supporting an ever impending sea of data due to the emergence of IoT is vastly different from the way we built things just 10 years ago. It is more complex because the amount of users and data have and are growing at a monumental pace. For the sake of this chapter we focus only on Lagom's restful capabilities but take a look at the complete capabilities at http://www.lagomframework.com.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

9.5.1 The Lagom Order Example

Lagom brings yet another way to define your restful interfaces where the controllers describe the routes they handle. This is a nice as everything is all in one place and there is no having to keep two things in sync as in Play with its routes file. We'll keep this example short just to illustrate the differences.

Listing 9.4 Lagom Order Web Service

```
class OrderService extends Service {
  val fakeOrders = Seq(
    Order(UUID.randomUUID(), UUID.randomUUID(), Seq(OrderLine(UUID.randomUUID(), 9))),
    Order(UUID.randomUUID(), UUID.randomUUID(), Seq(OrderLine(UUID.randomUUID(), 3)))
  )
  (1)
  implicit val orderLineFormat: Format[OrderLine] = Json.format
  implicit val orderFormat: Format[Order] = Json.format
  (2)
  def getOrders: ServiceCall[NotUsed, Seq[Order]] =
    ServiceCall { _ =>
      Future.successful(fakeOrders)
    }
  (3)
  override def descriptor = {
    named("orders").withCalls(
      restCall(Method.GET, "/orders/get-orders", getOrders)
    ).withAutoAcl(true)
  }
  (4)
}
```

(1) Again, we'll just return a fake collection of orders right from this service.
(2) These implicit formats enable Lagom to marshall the order to JSON.
(3) The function call to back the route simply returns the fake orders.
(4) The descriptor says what URLs you handle and how they map to your functions.

You can see above that even though Lagom is a complete framework capable of complex, distributed capabilities, the web service aspect is quite simple and compact.

9.6 Play vs Akka-Http vs Lagom

As you can see in the above code sections, Akka-Http being all code and no magic allows a great overall picture of an entire Http service. Indeed the accompanying code sample itself is contained in a single file. It's very easy to see the entire thing in a bird's eye view and there is less mystery than in Play, which does some of its magic at compilation and runtime. However at the time of this writing Akka-Http has the label of "experimental" as it has for some time, where Play has been put through its paces in many high throughput, complex environments for some years now. That being said, Akka-Http is based on Spray in which there are numerous production deployments. In fact when we first adopted CQRS/ES we used Spray for our many high throughput, mission critical services in the energy industry and it performed

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

flawlessly. It comes down to the expertise and risk tolerance of your team and/or the system you are building.

Play is a framework, the advantage of this is that it is easier to standardize your applications in terms of patterns and external libraries. This is important when:

- There is a large team and not enough oversight or code review. There I said it.
- Turnover is an issue as is hiring top talent.

In an environment like the above it may be hard for more aggressive developers to swallow the entire drink and there will be a desire to swap out pieces of the framework in favor of what may be new, cool or even better. Pressure may be applied against this on those developers resulting in stagnation of the code dependencies, which may lead to point b above.

9.7 Summary

- Restful interfaces are a clean and common way to have a headless API for your services.
- HTTP paired with JSON provides an expressive means of accessing your services, as well as describing their capabilities.
- We looked at the top choices for serving up reactive, restful applications, Play, Akka-Http and Lagom.
- We showed how the restful interface interacts with the CQRS command side in terms of responses and validation presentation.
- Although outside the scope of this book due to timing, Lightbend's Lagom deserves a hard look as a fully production ready restful, CQRS-ES framework.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

10

Production Readiness

This chapter covers

- Testing an application for deployment
- Securing against different threats
- Monitoring and logging strategies
- Deploying and scaling with cloud services

If you read this far into the book, congratulations! You know how to design an application that is responsive, elastic, resilient and message driven. The next step is to ensure those reactive design properties turn into real behavior when the application enters the turbulent world of a production system. In the past, the gap between development and operations has been vast, with sometimes disastrous results. The modern answer is DevOps, which aims to close that gap with a combination of cultural change, integrated tooling, and automation.

Covering the whole scope of DevOps is far beyond the scope of this book. Most of what applies to DevOps in general applies to reactive applications as well. In this chapter, we will look at some areas that reactive applications are different in operation than other architectures, and suggest a few things you as a developer can do to make them easier to manage in production.

As is so often the case in modern development, we will start with testing.

10.1 Testing a Reactive application

Actors are message-driven, so the one thing you know about every actor is that it receives messages. It might even react to a message, at some time in the future. Maybe. That is not much of a basis for writing tests, is it?

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

Recognizing common patterns of actor behavior will make it easier to design tests. Once you have identified the behavior that needs to be tested, you can apply corresponding patterns for testing. The Akka TestKit can help with common patterns.

10.1.1 Identifying test patterns

Actors that react to receiving a message by causing some side effect such as writing to a database may be tested synchronously by passing it a message and observing the result on the external object. One approach would be to instantiate the actor and call the `receive` method directly, but that omits initialization steps that can be difficult to get right and can lead to an erroneous test. Instead, as shown in figure 10.1, a `TestActorRef` can be created for the actor being tested with the details handled. The test case sends a message to the actor being tested, much as in the final application. After the message is sent, the test case uses assertions on the external object to verify that the test produced the expected result.

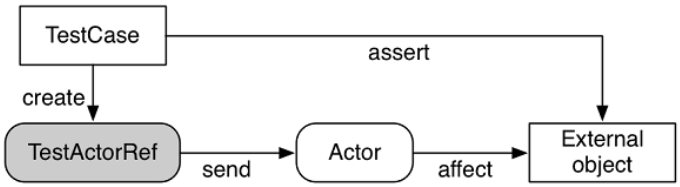

Figure 10.1 Actors with external side effects can be tested with traditional assertions on the affected object

A similar situation arises when the actor reacts to a message by changing some internal state, shown in figure 10.2. As with the side-effecting case just described, the solution is to create the actor using a `TestActorRef`. Again, the test case sends a synchronous message to the actor. The difference is that the test case needs access to the actor too, rather than just an `ActorRef`. The underlying actor is available as a property of the `TestActorRef`. The test case uses assertions on the underlying actor to confirm that the internal state has changed as expected.

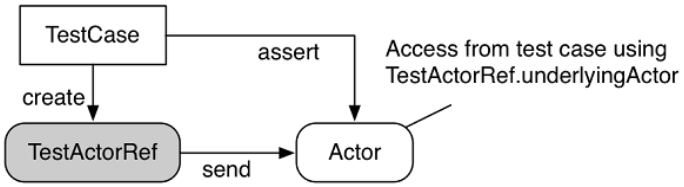

Figure 10.2 If the expected reaction is to change internal state, the assertion can obtain a reference to the underlying Actor

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

An actor that returns a `Future` using the *ask* pattern also can be tested using a `TestActorRef`, as shown in figure 10.3 Because this is a synchronous test, the ask is executed on the same thread as the test case and the `Future` is completed immediately. The test case can use `Future.get` and check assertions on the result.

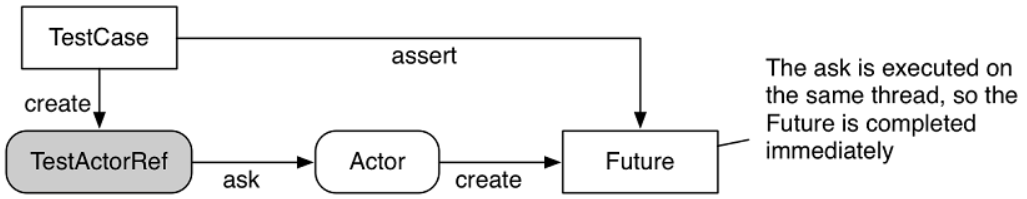

Figure 10.3 A TestActorRef can test the ask pattern by inspecting the completed Future it returns

The previous cases all use a `TestActorRef` test the actor synchronously. It would be better to perform tests asynchronously, by sending a message to the actor and having it produce another message. Asynchronous tests can use the Akka `TestKit`.

Asynchronous request-reply is a common pattern in actor systems. An actor receives a message and the sender expects a reply. Often the sender is another actor, but for testing the sender can be the test case itself. Do this by extending `TestKit` with the `ImplicitSender` trait, as shown in figure 10.3. The `TestKit` expects an actor system to be passed to the constructor, and uses that to execute the actor asynchronously. The implicit sender trait ensures that replies from the actor are sent back to the test case.

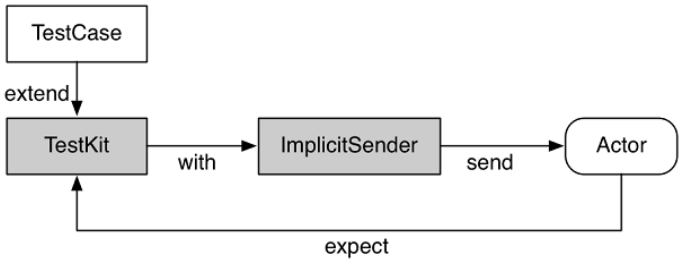

Figure 10.4 Use the TestKit to specify an expected reply

Because the actor is executing asynchronously, time becomes a factor in the tests. The test kit can enforce expectations for how quickly or how slowly the actor responds, and even allows for a scaling factor to account for the speed of the test server.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

Cleaning up after a test

When a test specification extends TestKit, it passes an ActorSystem to the constructor. Be sure to shut down that ActorSystem at the end of the test, like this:

```
import akka.actor.ActorSystem
import akka.testkit.TestKit
import org.scalatest.{BeforeAndAfterAll, Matchers, WordSpecLike}

class TestMyActor extends TestKit(ActorSystem("TestSystem"))
[CA]with WordSpecLike with Matchers with BeforeAndAfterAll {

  override def afterAll {
    TestKit.shutdownActorSystem(system)
  }
}
```

If the actor system is not shut down, the test actors will continue running and eventually consume all the system resources.

If the receiving actor reacts by sending messages to other actors rather than only replying to the sender, consider using two instances of `TestProbe` to stand in for those two actors, as shown in figure 10.5. This class extends `TestKit` and provides the ability to send, receive and reply to messages and create assertions about the messages it receives.

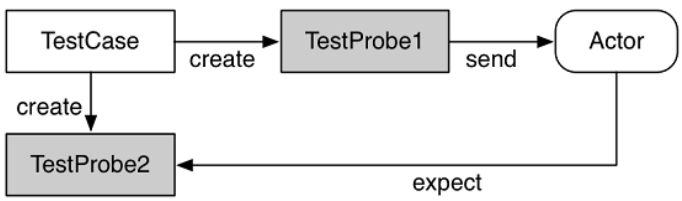

Figure 10.5 Use TestProbes in place of multiple ActorRefs

Finally, integration testing may require having multiple complete actor systems exchanging messages. Each `TestProbe` accepts an `ActorSystem` as a constructor parameter, so one approach is to execute the two actor systems in the same JVM. If this is not sufficient and you need multiple JVMs, consult the latest documentation for `akka-multi-node-testkit`.

10.2 Securing the application

Any time one system connects to another you must consider undesired communication as well as desired. There is much more to securing an application than enabling HTTPS and calling it done, and it starts with knowing how to identify the threats.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

In the past, security often was treated as an afterthought. More recently, well-publicized breaches have led to increased focus on securing applications. The principles that apply to securing reactive applications are no different than for traditional applications. The key is to take a disciplined approach to managing the threats and their corresponding mitigations. In this section, you will learn about the STRIDE approach and how boundary services and HTTPS can be used to counter some of the threats you identify.

10.2.1 Taking threats in STRIDE

When security professionals evaluate an application, they often use a threat classification scheme called STRIDE. This system was developed by Microsoft and, not surprisingly, the name is an acronym for different categories of threat:

- *Spoofing Identity*—Performing an action using another user's identity
- *Tampering with Data*—Changing data as it being processed or after it is stored
- *Repudiation*—Denying that a user has performed some action
- *Information Disclosure*—Accessing information without proper authorization
- *Denial of Service*—Preventing a system from servicing valid requests
- *Elevation of Privilege*—Gaining access to more powerful operations than intended

It is important to be systematic. Examine each interface in the system, and for each interface ask questions about all six components of STRIDE. Keep track of your work. Examples of each type of threat are shown in figure 10.6.

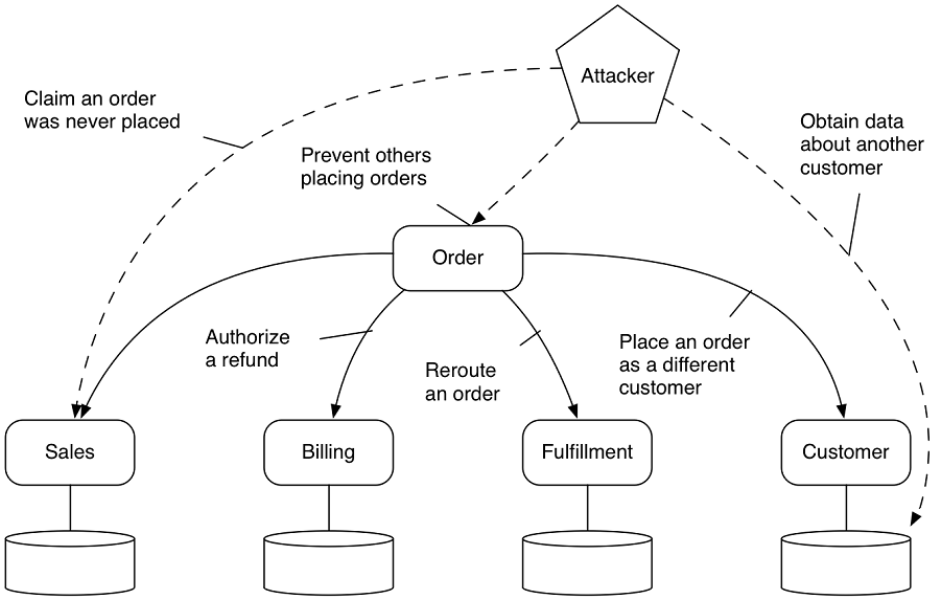

Figure 10.6 Use the STRIDE model to identify and categorize threats to the application

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

TIP The free Microsoft Threat Modeling Tool is a great way to develop and manage a threat model. It provides a drawing tool to model the data flow of the application, generate a STRIDE-based threat model the data flow, analyze the threats, and track their corresponding mitigations. The tool may be downloaded from https://www.microsoft.com/en-us/download/details.aspx?id=49168. Although it can be used to model threats to any application on any operating system, the 2016 version of the tool itself is available only for Windows.

Here are a few sample questions to get started:

- *How do internal actors know that they are receiving messages from a trusted sender?* If an attacker finds a way to send messages directly to a service that is intended for internal use, checks that are supposed to have been performed by the sender can be skipped. At minimum, this opens the application to data tampering, repudiation and information disclosure threats.
- *How do actors know that they are sending a message to a trusted receiver?* If an actor always sends a reply to the original sender, information disclosure is a risk because the sender address may be forged.
- *Are AJAX and WebSocket connections secured as well as the initial HTTPS request for the page?* One way to mount an identity spoofing attack is to authenticate as one user, then change the username passed in subsequent requests. Potential solutions include not passing the username in the request, which could require maintaining session state on the server, or including a cryptographic signature with the data in the request to detect tampering with that data.
- *What limits resource consumption?* Suppose the application starts a new actor for every new user. An attacker could exhaust all the memory by pretending to be many users at once, denying service to legitimate users. The elastic properties of a reactive system allow it to scale horizontally. Automatic scaling can fend off an attack, but also could result in an expensive bill at the end of the month.
- *Can internal services be bypassed entirely?* This is especially important if the application uses a NoSQL persistence service that has an HTTP interface. Direct access to a data store can admit nearly any category of threat.

It is easier to attack a system directly rather than indirectly, so network access is a common theme across many of the threats. Administrative and customer service interfaces make especially tempting targets. In the next section, you will see a few ways to limit exposure to the outside world. Making your job easier and the attackers job harder is a good thing, right?

10.2.2 Barbarians at the gate

An important difference between reactive and traditional applications is that reactive applications tend to have many microservices, and they may be distributed across multiple servers. That translates into having a large attack surface. One way to mitigate that threat is to create gateway services that handle requests from outside the system, then ensure that

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

other pathways into the system are blocked at the network level. Figure 10.7 shows this approach applied to the system in the previous section.

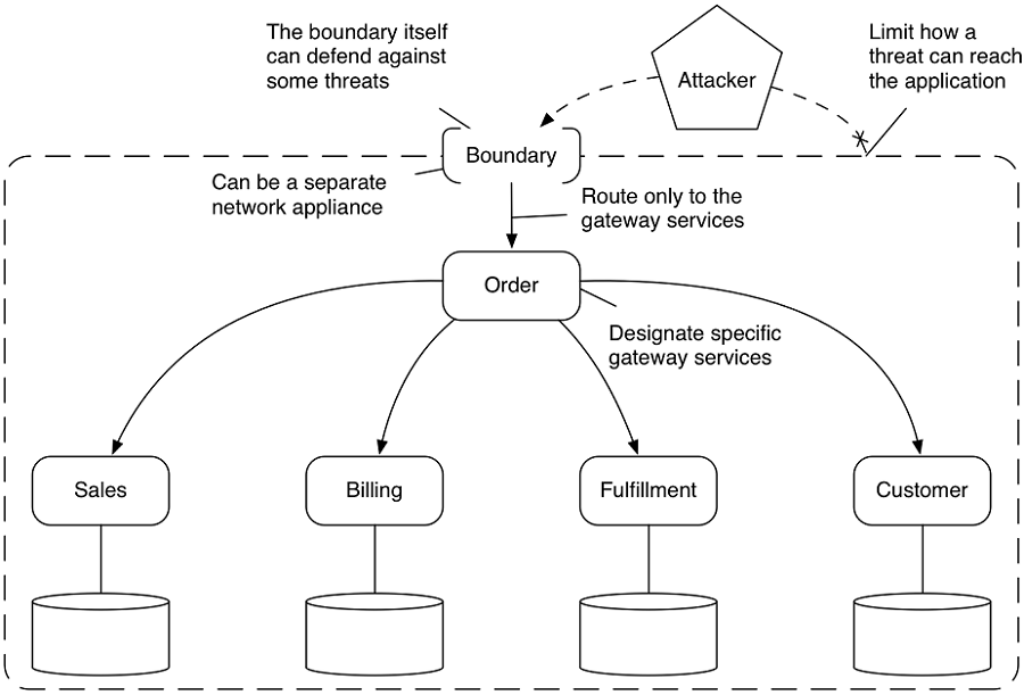

Figure 10.7 Allowing client access only to a few gateway services reduces the attack surface

There are many ways to create a boundary that prevents traffic reaching unauthorized endpoints:

- Services on a private subnet cannot be reached directly via the internet
- A virtual private network (VPN) can encrypt and validate all communications between system components
- A firewall can monitor and block messages into the system

Each alternative has different advantages and disadvantages, and can be used alone or in conjunction with others. What they have in common is they put the protected components in a common *threat zone*, which is a group of components that share set of threats and defenses, and that have a consistent level of trust in each other.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

10.2.3 Adding HTTPS

HTTPS establishes Transport Level Security (TLS) on HTTP messages. That means messages are encrypted between the two endpoints, so a client can communicate safely with a server. But which server? That question also can be answered by HTTPS, using a cryptographic certificate that is sent back to the client for verification. Modern browsers handle that verification automatically. Based on that information, a browser can be sure that it is connected to the intended server.

EVALUATING THE BENEFIT OF HTTPS

HTTPS also has the capability to validate in the reverse direction, so a client can present a certificate back to the server too. That form of identification is rarely used on the Internet today. Instead, applications rely on other means such as usernames and passwords. In terms of the threat model:

- HTTPS in front of boundary services defends against threats that happen in transit from the client (browser) and the application. It does little to defend against compromised clients, though it has some benefit in conjunction with setting the "secure" flag in cookies. Consider it primarily as a component in defending against identity spoofing, data tampering and information disclosure threats.
- HTTPS between services is useful to assure clients that they are communicating with valid services *if* they check the certificate. If it is important to validate the client too, client certificates are required. HTTPS between services within the same threat zone is usually redundant. When used across threat zone boundaries, the considerations are like those for boundary services.

Using HTTPS limits options for components such as load balancers because the HTTP headers are not visible to the intermediate nodes. Most often, HTTPS makes sense on the boundary between a browser and application, but not for communication between servers within the application.

TERMINATING HTTPS

The point where an HTTPS connection is received and decrypted is called the *termination* of the connection. In the past, this function has been so computationally expensive that it often was left to dedicated hardware to perform SSL acceleration. This is seldom necessary with modern processors, but it still makes sense to isolate the termination from other functions. Often the termination function is combined with load balancing. Some of the different approaches include:

- Akka-HTTP can be configured for HTTPS
- Play Framework provides a flexible web-facing front end
- Nginx can be used as both a load balancer and for termination

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

The SSL configuration module used by Akka HTTP was originally part of the WS module from the Play Framework, so the first two options have a lot in common.

Now that the application is well-tested and secure, it is well on the way to production readiness. Of course, nothing ever seems to go that smoothly. Sometimes things go wrong, and you need to check application logs to discover what happened.

10.3 Logging actors

Logging is obviously message-driven. You might expect that nothing needs to be done to fit logging directly into a reactive system, but that is not true. The reason is that logging implementations often do the one thing that an actor should not do if it possibly can be avoided, which is perform synchronous I/O. In the most common logging packages, the decision whether a message is logged synchronously or asynchronously does not come from the logger. Instead, it is decided by the component, often called an *Appender,* that writes the message. Appenders are usually configured at runtime. That means application code has no choice in the matter. A small change to log configuration can have a huge impact on performance.

The reason this happens is that logging libraries are designed to be simple and work for any application. They do not assume that there is a sophisticated message passing system already built in to the application. In a reactive system, sophisticated message passing is exactly what you have available. The solution is to use it, and the next design choice is how to integrate it into the actors.

Akka has that covered for you!

10.3.1 Stackable logs

Logging is a *cross-cutting concern*, which means it is a capability needed by many different components that otherwise have little or nothing to do with each other. One way to approach these concerns is with a stackable trait. A stackable trait is a design pattern used to compose the functions of two different classes. You stack the logging function on top of another actor. Akka provides a stackable log in the form of `ActorLogging`, as follows:

Listing 10.2 Stacking the `ActorLogging` trait on an `Actor`

```
import akka.actor.{Actor, ActorLogging}

class SimpleLogging extends Actor with ActorLogging {  ❶

  override def receive = {
    case msg => log.info("Received {}", msg)           ❷
  }
}
```

❶ **Stack `ActorLogging` to get a `log` that is integrated to Akka messaging**
❷ **The methods on `log` are similar to other logging packages**

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

Stacking `ActorLogging` produces a `LoggingAdapter` that implements common functions such as error, warn, info and debug. The implementations do not evaluate the parameters or perform string interpolation unless the log is enabled, so it is not necessary to wrap these calls in the isDebugEnabled functions found in some packages.

> **WARNING** The log is intended for use within the actor, where thread safety is not a problem. Be careful not to pass a reference outside the scope of the actor's thread, such as by referencing it in a `Future`.

The `LoggingAdapter` sends all log messages to an Akka event bus. The event bus is an internal publish and subscribe system that is already used to pass many of the built-in notifications that you have seen previously, such as actor lifecycle events, dead letter notifications, and unhandled message errors.

Akka logging is intended to pass events to some other logging package that you select and configure.

10.3.2 Configuring the log system

Logging is configured through properties in application.conf. The default simply logs to stdout. For production, much more control is needed. The preferred configuration is a combination of *Simple Logging Façade for Java (SLF4J)*, http://www.slf4j.org, combined with *logback*, http://logback.qos.ch.

As shown in figure 10.8, filtering is applied to log events before they are published to the event bus, and the remainder of the logging is handled by the log event subscriber. You want the filtering to happen before a log event reaches the Akka event bus, otherwise the production system could be clogged with debug messages that have not yet been filtered. The approach Akka uses is to provide a `LoggingFilter` interface and an implementation, `Slf4jLoggingFilter`, that uses the SLF4J backend configuration rather than inventing one of its own.

Once the log event is published to the event bus, it can be read by a logger. The provided `Slf4jLogger` can be configured with logback as normal. If the application is being deployed to a containerized environment such as Docker, it is preferable to use the logback `ConsoleAppender` rather than the default stdout because it provides more control over filter configuration.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

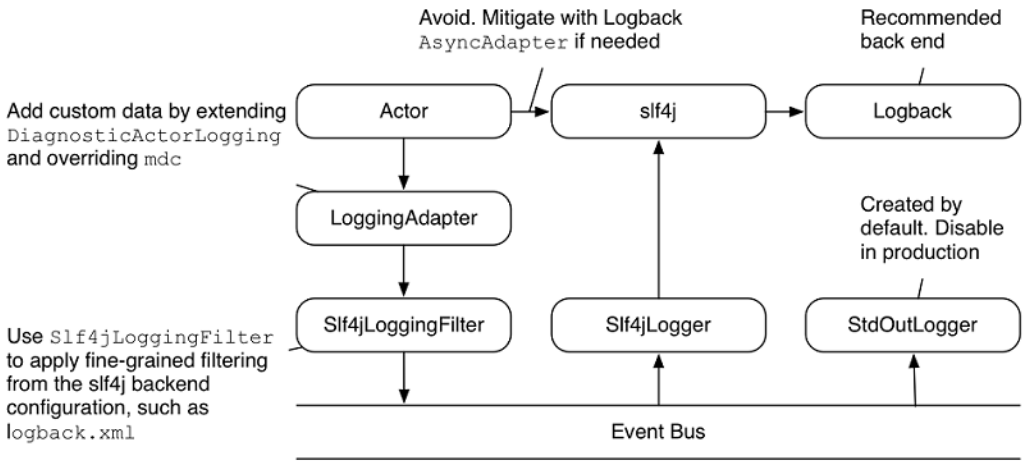

Figure 10.8 Akka integrates with logging through SLF4J

Integration includes setting configuration parameters for the log system. A few of the values that can be set are shown in the following listing, and more are available in the Akka documentation.

Listing 10.3 Log configuration in `application.conf`

```
akka {
  loggers = ["akka.event.slf4j.Slf4jLogger"]                    (1)
  loglevel = "DEBUG"                                            (2)
  logging-filter = "akka.event.slf4j.Slf4jLoggingFilter"        (3)
  log-config-on-start = on                                      (4)
  log-dead-letters = 10                                         (4)
  log-dead-letters-during-shutdown = on                         (4)
  actor {
    debug {
      lifecycle = on                                            (5)
      autoreceive = on                                          (5)
      receive = on                                              (5)
      unhandled = on                                            (5)
    }
  }
}
```

(1) Recommended in place of the default `StdOutLogger`
(2) Consider filtering to only INFO or even higher in production
(3) Activate additional filter based on the back end logging package
(4) Refer to Akka documentation for auxiliary logging parameters
(5) Log common actor events and messages

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

TIP If the situation requires calling the logging package directly, such as a third-party JAR that has logging built in, be sure to configure an intermediate stage such as the logback `AsyncAppender` to prevent blocking I/O.

Sometimes it is helpful to log every user message received by an actor. That can be implemented easily by wrapping the receive function in the `LoggingReceive` trait, as shown in the following listing.

Listing 10.4 Logging every message received by an `Actor`

```
import akka.actor.Actor
import akka.event.LoggingReceive

class WrappedLogging extends Actor {

  override def receive = LoggingReceive {     (1)
    case _ => {}                              (2)
  }
}
```

(1) Add `LoggingReceive` to log every message received by an actor. This also requires `akka.actor.receive = on` to be configured in application.conf
(2) Application `Receive` logic is within the `LoggingReceive` block.

When something goes wrong in a monolithic application, often you can figure out what happened by analyzing the log messages associated with a single request. The same is true for a reactive application. The difference is that in a reactive application, the log messages associated with a single request may span many microservices. There are a few log options related to remote messaging, but stitching logs together across many JVMs to form a trace would be difficult to do from scratch. Instead, consider using a dedicated system to collect and manage distributed trace events.

10.4 Tracing messages

Akka does not have tracing covered for you, but there are several solutions available so it is not necessary to build something from scratch. Both the monitoring tools described in the next section "Monitoring a Reactive application" include tracing modules that are specific to Akka. But because they are designed to be general-purpose, a dedicated tracing solution can cover both actor and non-actor system components.

Conceptually, tracing is simple. The systems that process messages generate trace events whenever a message is sent or received. The events are then sent to a common collector, where they are correlated and queries can be run, as shown in figure 10.9. The result of a naïve implementation would be an immediate tripling of the number of messages being sent. There is the actual message, plus one trace message from the sender when a message is sent, plus another from the receiver when a message is received. Two thirds of the traffic in the system would be sent to the single collector. This would not scale well. It would scale even

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

less well if you wanted to trace messages between actors within the same actor system in addition to messages that flow between systems. A tracing library can take care of managing the messages to the collector for you. Understanding a few basic strategies and selecting a tracing library such as one of the implementations described in the next section is enough to get you started.

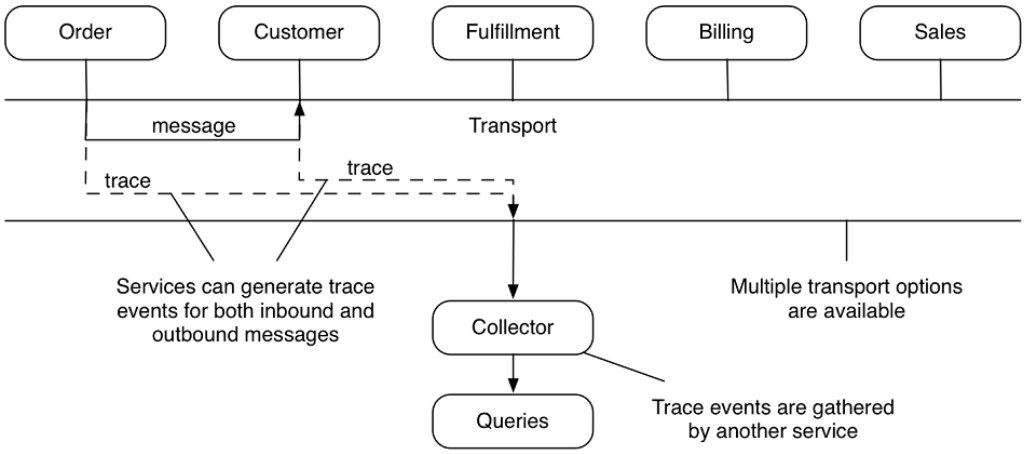

Figure 10.9 Tracing requires instrumentation on the services, a transport layer, and a collector to accumulate the trace events

Message volume can be reduced by accumulating trace events and sending many events to the collector in a single trace message. This is best accomplished with dynamic tuning to strike a balance between the number of events sent at once and the latency between when an event occurs and when it is sent to the collector.

Another strategy to reduce volume is to sample events rather than tracing every message. The decision whether to trace a chain of messages should be made at the initial event and carried through subsequent messages, otherwise there would be too many fragmentary traces. The decision may be made randomly, or based on a parameter passed with the initial event, or both.

In addition to deciding which events to trace and how to collect them, you must decide what data to collect for each event so that they can be correlated easily. One approach to is to use application data from the log messages, such as customer identifiers, product SKUs, invoice numbers, account numbers, and so on. That may suffice in a small test environment, but it falls apart quickly in production. The volume is too high, the queries are too complex, and it relies on developers having added the right information to the logs at the right place and at the right message level. A debug message is not helpful if production operates with debugging turned off!

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

Rather than relying on application-specific information, it is preferable to have a consistent strategy for labeling events. One widely adopted strategy is the Dapper pattern.

10.4.1 Collecting trace data with the Dapper pattern

In 2010, Google described their solution in a paper titled *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. When an initial message is received from the outside world, it triggers a tree of additional messages. The initial message is assigned a *trace id*. In figure 10.10, that unique identifier is assigned by the order service. Each subsequent message is then assigned a unique *span id*, which is passed along with the trace id. In the example, the first message from the order service is assigned span id s1, which is passed along with the trace id to the billing service. When the billing service in turn sends a message to the invoicing service, that message is also assigned a unique span id. The new span id is passed along with the original trace id and the span id of the parent message. The same pattern is repeated for the message from the order service to the fulfillment service, and continues to the warehouse, packing and shipping services. Using this data, it is possible to construct the entire graph of messages triggered in response to a single initial message, and it is possible using the trace id to correlate any message directly back the original request without having to traverse each step along the way.

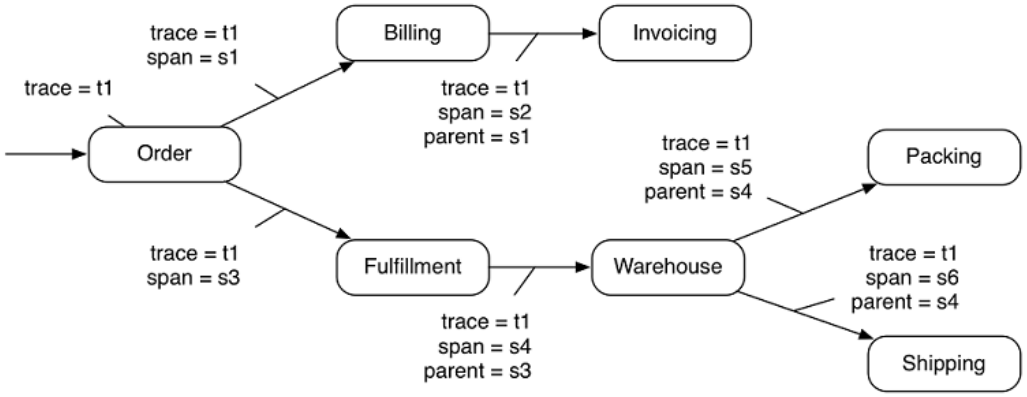

Figure 10.10 The original input is assigned a trace identifier that is passed along with every resulting message. Each message is given a unique span identifier, and the prior (parent) span is passed too if there is one

A minor weakness of the Dapper pattern is that it does not recognize when multiple messages come together to trigger a single outbound message. Figure 10.11 shows a simple case where either of two messages into the "select carrier" actor could be considered the parent of an outbound message that contains the carrier selection. Normally it will be simplest to select the final message that arrived at the sending actor.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

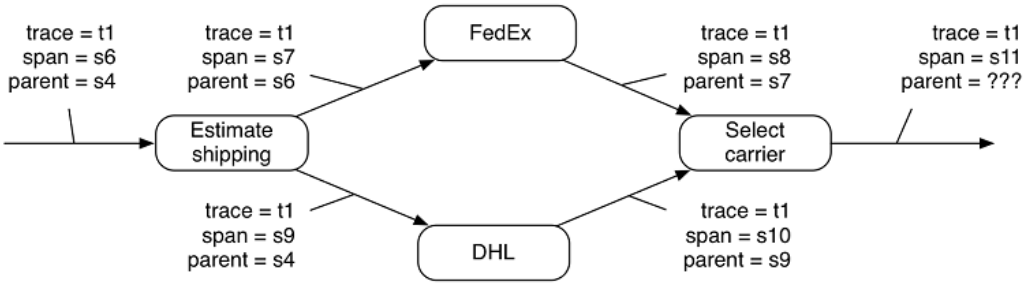

Figure 10.11 The Dapper pattern assumes that each message has a single parent span. When multiple messages are collected, the selection of which span to call "parent" may be arbitrary

There are several implementations based on the ideas in the Dapper paper:

- *Dapper*—The original implementation from Google is not open source as-of this writing, but the research paper that inspired many of the other efforts is available online. http://research.google.com/pubs/pub36356.html
- *Zipkin*—An open source implementation used by Twitter that draws heavily on the Dapper research paper, Zipkin offers some integration libraries supported by the OpenZipkin Github group, and still more supported by the community. Zipkin is also interesting because it supports multiple transports including HTTP, Kafka and Scribe. http://zipkin.io
- *Spring Cloud Sleuth*—An open source implementation specialized for Spring Cloud and featuring direct integration to Zipkin. Consider this particularly in a mixed environment that includes both Spring and Akka components. http://cloud.spring.io/spring-cloud-sleuth
- *Apache HTrace*—An open source implementation that as-of this writing has Apache incubator status. It also has direct integration to Zipkin, and features Java, C and C++ libraries. http://htrace.incubator.apache.org

Having multiple APIs available for tracing leads to the problem of having to choose one for your application. The task will be complicated further by features of the tracing system you choose, such as sampling, annotations and different transports. There are some libraries available, but as-of this writing none has official support. Look at the existing instrumentations section of the documentation for the tracing system to find the latest developments.

10.4.2 Reducing dependencies with OpenTracing

OpenTracing (http://opentracing.io) addresses implementation dependence by providing standard instrumentation that can be configured with pluggable providers. OpenTracing was initiated by Ben Sigelman, who is also one of the authors of the Google Dapper research paper.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

The OpenTracing is based on a semantic model that is mapped idiomatically onto multiple programming languages. You use the language-specific OpenTracing API throughout your application, and configure which tracing implementation will receive the data in just one place. Not every tracing implementation supports the complete set of languages, so the choice of provider should consider all the components of your application. Table 10.1 shows some of the combinations that are available.

Table 10.1 For most providers the Java API is supported, but not a specialized Scala API.

Provider	Scala	Java	JavaScript	Go	Other
Zipkin		✔		✔	
Jaeger		✔	Node.js	✔	Python
AppDash				✔	
LightStep		✔	✔	✔	Python, Objective-C, PHP, Ruby, C++
Hawkular		✔			
Instana	✔	✔	Node.js		PHP, Ruby
sky-walking		✔			

In OpenTracing, a span:

- Either begins a new global trace or joins an existing trace
- Has a name
- Has a start and finish timestamp
- May have relationships to other spans, such as "child of" or "follows from"
- May have *tags*, which have a name and a primitive value such as a string, Boolean or numeric value
- May have *log* items, which have a name and an arbitrary value

OpenTracing has the ability for a span to identify multiple parent spans, which was a weakness of Dapper discussed in the previous section. But if the pluggable provider you use does not have that capability then only one parent will be used.

Because Akka does not have built-in support for any of these, you must do some integration.

10.4.3 Integrating OpenTracing with Akka

You have already learned that tracing has the concept of a *span*, which is a chunk of processing that has its own trace measurements. In traditional systems, a span may be placed around a servlet receiving and responding to an HTTP request. An application might create

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

additional child spans around a service call or database operation that happens during the course of processing the HTTP request.

In Akka, the most obvious place for a span is around an actor's `receive` function. The approach is shown in figure 10.12. As each message is received, we will start a new span that continues for the duration of the receive processing. When our actor sends a message to another actor, it will attach context information about that span to the message. When the receiving actor starts its span, it will use the context information to identify the sender as its *parent* span.

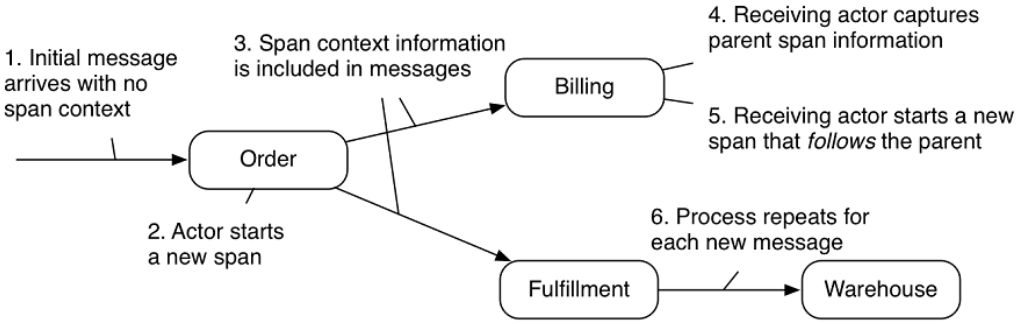

Figure 10.12 Each actor starts a new span when it receives a message. If trace information about the sending span is found in the message, the new span records a reference to the parent.

Before starting the integration, you will need to add the OpenTracing API as a dependency in your build.sbt file. The examples in this section were created using version 0.21.0, so add the following to libraryDependencies:

```
io.opentracing" % "opentracing-api" % "0.21.0"
```

Because the OpenTracing API is generalized for multiple languages and many design patterns, and is written in Java rather than Scala, the first step is to create a few adapters to fit better with Akka. After that, you will instrument an example actor and configure an application with a tracer implementation.

BUILDING AN INTERFACE TO OPENTRACING

Each message needs to carry the additional trace, span and parent identifiers. In addition, metadata about each message sent or received by each actor needs to be collected and sent to the trace server. In a reactive system, that means the messages contain trace information, and the receive functions are instrumented to receive and manage the tracing spans. OpenTracing requires tracers to support two different formats to carry the trace information: a binary format and a text map format. For convenience and readability, we will choose a text map. The following listing shows a simple trait that can be mixed in to message definitions:

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development

Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

Listing 10.5 Traceable messages contain generic carrier information

```
trait Traceable {
  val trace: Map[String, String]                          (1)
}
```

(1) Trace information will be carried in messages as an immutable Map

The `Traceable` trait follows reactive conventions, so it is immutable. You will need a function to convert trace information from the current `Span` into an immutable `Map`. But where is the current span? Every actor that is being traced will have a var to hold the current span and a function to tell the tracer to convert the span context information into a map that can be added to a `Traceable` message. The following listing shows how to provide all of those things in a trait that can be added to any actor that is being traced.

Listing 10.6 Create a trait to hold the variable span and convert it to traceable representation

```
import io.opentracing.{Span, Tracer}
import io.opentracing.propagation.TextMap
import io.opentracing.propagation.Format.Builtin.TEXT_MAP
trait Spanned {

  val tracer: Tracer                                      (1)

  var span: Span = _                                      (2)

  def trace(): Map[String, String] = {
    var kvs: List[(String, String)] = List.empty          (3)
    tracer.inject(span.context(), TEXT_MAP, new TextMap() {  (4)
      override def put(key: String, value: String): Unit =
        kvs = (key, value) :: kvs

      override def iterator() =                           (5)
        throw new UnsupportedOperationException()         (5)
    })
    Map(kvs: _*)                                          (6)
  }
}
```

(1) The tracer will come from the actor's Props
(2) The span state is a var, not a val
(3) The span state will be converted into a list of key-value pairs
(4) The inject function is supplied from the underlying tracer
(5) Injection does not need to extract data from the message
(6) Use the list of key-value pairs to create an immutable map that can be included in a Traceable message

Now that you have a way to hold a span and a way to add the span context information to an outbound message, the final piece is something to receive those messages and extract the span context.

As with logging, tracing is a cross cutting concern and stackable traits simplify implementation. `Receive` is just a type alias for `PartialFunction[Any, Unit]`, so a stackable

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

version needs to override the isDefinedAt and apply functions. The isDefinedAt function just passes through to the Receive this is stacked on top of. The apply function:

1. Extracts the span context from the incoming message
2. Starts a new span with a reference to the incoming span
3. Invokes the apply function on the underlying Receive
4. Finish the span

Stackable tracing can be implemented as follows:

Listing 10.7 Implement the message interception as a stackable trait that extracts the span contex

```
import akka.actor.Actor.Receive

import io.opentracing.{SpanContext, Tracer}                (1)
import io.opentracing.propagation.Format.Builtin.TEXT_MAP  (1)
import io.opentracing.References.FOLLOWS_FROM              (1)
import scala.util.{Failure, Success, Try}

class TracingReceive
[CA](r: Receive, state: Spanned)                           (2)
[CA]extends Receive {                                      (3)

  override def isDefinedAt(x: Any): Boolean =              (4)
[CA]r.isDefinedAt(x)                                       (4)

  override def apply(v1: Any): Unit = {
    val operation: String = v1.getClass.getName            (5)

    val builder: Tracer.SpanBuilder =                      (6)
[CA]state.tracer.buildSpan(operation)                      (6)

    state.span = extract(v1) match {                       (7)
      case Success(ctx) =>
        builder.addReference(FOLLOWS_FROM, ctx).start()    (8)
      case Failure(ex) =>
        builder.start()                                    (9)
    }
    r(v1)                                                  (10)
    state.span.finish()                                    (11)
  }

  def extract(v1: Any): Try[SpanContext] = ???             (12)
}
```

(1) Import OpenTracing API components
(2) Require the wrapped Receive function and the span state from the Actor
(3) Make the trait stackable
(4) Proxy isDefinedAt to the wrapped Receive function
(5) You can use the incoming message type as the operation name
(6) Begin building the new span
(7) Extract the trace information from the incoming message
(8) Create a reference to the sender and start the span
(9) If no trace information was received, start a new global span with no reference

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

⑩ Invoke the wrapped Receive function
⑪ Finish the span
⑫ Extract will be discussed next

The extract function is implemented as a Try because there are several ways it can fail and we do not want the entire Receive to fail just because tracing information could not be extracted. If it fails, we will start a fresh global trace and continue. The extract can fail if the incoming message is not Traceable, if the trace information is missing, or if the configured tracer is unable to read it. Other than the error handling, the extract function is mostly concerned with converting mapping between Java and Scala collection data types, as shown in the following:

Listing 10.8 The extract function is just a decorator that converts data types and handles errors

```
import io.opentracing.propagation.TextMap
import java.util.{Iterator => JIterator, Map => JMap}          (1)
import scala.collection.JavaConverters._                       (1)

def extract(v1: Any): Try[SpanContext] = v1 match {
  case m: Traceable =>                                         (2)
    if (m.trace.isEmpty) Failure(new NoSuchElementException("Empty trace"))
    else Try(state.tracer.extract(TEXT_MAP, new TextMap() {
      override def put(key: String, value: String): Unit =
[CA]throw new UnsupportedOperationException()                  (3)

      override def iterator(): JIterator[JMap.Entry[String, String]] =
        m.trace.asJava.entrySet().iterator()
    })) match {
      case Success(null) =>                                    (4)
[CA]Failure(new NullPointerException                           (4)
[CA]("Tracer.extract returned null"))                          (4)
      case x => x                                              (5)
    }

  case _ =>
    Failure(new UnsupportedOperationException                  (6)
[CA]("Untraceable message received"))                          (6)
}
```

① The OpenTracing API uses Java collections rather than their Scala counterparts
② Check whether the message can carry trace information
③ Extract function does not need to put data into the message
④ OpenTracing extract may return `null` rather than throwing an exception if no tracing data is found
⑤ Otherwise return the extracted trace information
⑥ Signal the messages cannot carry trace information

Finally, an apply function in the companion object makes stacking this trait more fluent, as you will see in the next section:

Listing 10.9 The companion object enables more fluent declarations

```
object TracingReceive {
  def apply(state: Spanned)(r: Receive): Receive =  (1)
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

```
    new TracingReceive(r, state)
}
```

❶ Declare the Receive function as a second parameter list to make the trait easily stackable

Now that you have a clean integration between Akka and OpenTraciing, you can reap the benefits by instrumenting a few actors.

INSTRUMENTING AN ACTOR

Our example revisits the guidebook/tourist actor system that you saw in chapter 2. The source is available at https://github.com/bhanafee/HelloRemoting/tree/tracing. The first step is to modify messages to the to carry trace information using the previously defined `Traceable` trait, as follows:

Listing 10.10 Add traceability to the guidebook's Inquiry message

```
object Guidebook {

  case class Inquiry(code: String)
[CA](val trace: Map[String, String])                    ❶
[CA]extends Traceable                                   ❷
}
```

❶ Add the trace information as a second parameter list.
❷ Label the message as traceable

> **TIP** If a case class has multiple parameter lists, only the parameters in the first list are used to generate the `apply`, `unapply`, equality and other case class goodies. You can use this to your advantage when adding trace information to messages. By declaring the trace information in the second parameter list, the rest of the actor mostly can ignore it.

Now that the guidebook message is traceable, the next step is to stack `TracingReceive` on the existing receive implementation. To prepare for that, modify the `Guidebook` constructor to require a tracer. Next, add the `Spanned` trait to the actor so it can keep track of the current span, and stack the trait. Finally, add a call to the trace() function as part of constructing `Guidance` messages. Bringing these steps together, the Guidance actor will have the changes shown in the following listing:

Listing 10.11 Tracing requires some changes to the actor

```
import java.util.{Currency, Locale}

import akka.actor.Actor
import Guidebook.Inquiry
import Tourist.Guidance
import io.opentracing.Tracer

class Guidebook(val tracer: Tracer)                     ❸
[CA]extends Actor
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

```
[CA]with Spanned {                                                    ❹

  def describe(locale: Locale) =
    s"""In ${locale.getDisplayCountry}, ${locale.getDisplayLanguage} is spoken and the
       currency is the ${Currency.getInstance(locale).getDisplayName}"""

  override def receive = TracingReceive(this){                        ❺
    case Inquiry(code) =>
      println(s"Actor ${self.path.name} responding to inquiry about $code")
      Locale.getAvailableLocales.filter(_.getCountry == code).
        foreach { locale =>
          sender ! Guidance(code, describe(locale))(trace())          ❻
        }
  }
}
```

❶ Add the trace as an additional parameter list
❷ Declare that the message carries trace information
❸ The tracer is passed to the class constructor
❹ Add span state to the actor
❺ Stack TracingReceive on the receive function
❻ Add the trace information to the message

As you can see, the integration is not too intrusive on the actor, especially when you consider how useful it could be to have a complete map of every message that passes through your application. Whether it makes sense to track *every* message is up to you. Some tracing systems allow you to be more selective. It depends on the tracer that is configured into your application. Configuring a tracer is covered next.

CONFIGURING A TRACER IMPLEMENTATION

One of the nice aspects of OpenTracing is that the tracer configuration can be handled in one place, as part of application initialization. In an Akka-based system, that means adding to the tracer to the driver. Start by including the tracer library to build.sbt as a dependency. For this example, we will use the mock tracer that matches the tracing API version, so add the following to libraryDependencies:

```
io.opentracing" % "opentracing-mock" % "0.21.0"
```

The mock tracer itself is each to configure, as shown in the following:

Listing 10.12 The driver instantiates the tracing implementation and includes a reference to it in the actor's Props

```
import akka.actor.{ActorRef, ActorSystem, Props}
import io.opentracing.Tracer                                          ❶
import io.opentracing.mock.MockTracer                                 ❶

object GuidebookMain extends App {
  val tracer: Tracer =                                                ❷
    new MockTracer(MockTracer.Propagator.TEXT_MAP)                    ❷

  val system: ActorSystem = ActorSystem("BookSystem")
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

```
  val guideProps: Props = Props(classOf[Guidebook], tracer) ❸

  val guidebook: ActorRef =
[CA]system.actorOf(guideProps, "guidebook")
}
```

❶ Import the OpenTracing API and a mock implementation
❷ Instantiate the tracer
❸ Add the tracer to the Props

You can complete the implementation by making similar changes to the Guidebook.Inquiry message and the Tourist actor. As in chapter 2, the systems can be started in separate JVMs using

```
sbt -D"PORT=2553" "run-main GuidebookMain"
```

and

```
sbt -D"PORT=2552" "run-main TouristMain"
```

You should see messages exchanged exactly as before, but now they are being traced too! Because we used just a mock tracer for the example, the traces will appear as additional console messages. Once you have selected a tracing implementation, follow its instructions to replace the mock tracer with the real tracer. The changes should affect the `GuidebookMain` and `TouristMain` drivers, but not the actors themselves or the messages. Those are universal and should work unchanged with any tracer that supports the OpenTracing Java API. Congratulations! You have just instrumented a reactive system to work with multiple tracing libraries.

Operating a complete tracing solution may be more than your application needs. Sometimes you just want to discover which actors exchange messages with which other actors, and a full tracing solution is too heavy. For a lightweight solution to finding pathways among actors, consider the spider pattern.

10.4.4 Finding pathways with the spider pattern

Rather than providing a trace of every message, the spider pattern simply identifies pathways through an actor system. As the system expands to many actors, that is more useful than you might expect. Because the pattern is lightweight, it is well-suited for tracking interactions that happen among actors within a single actor system. Those pathways can get quite complex!

In the spider pattern, each actor keeps track of other actors that have

- Sent it messages
- Been sent messages
- Been created by it

Individual messages are not traced, so it is not necessary for them to carry unique identifiers. The message definitions within the application can remain unchanged. Instead, a new message is added to the application. As shown in figure 10.13, the new message is a probe that tells each actor to collect all the connection information is has and pass it along to a

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

collection actor, then forward the same probe to each of those actors so they do the same thing. The initial probe is assigned a unique identifier that the actors use to ensure they do not respond to the same probe more than once. The information is passed using the existing Akka transport.

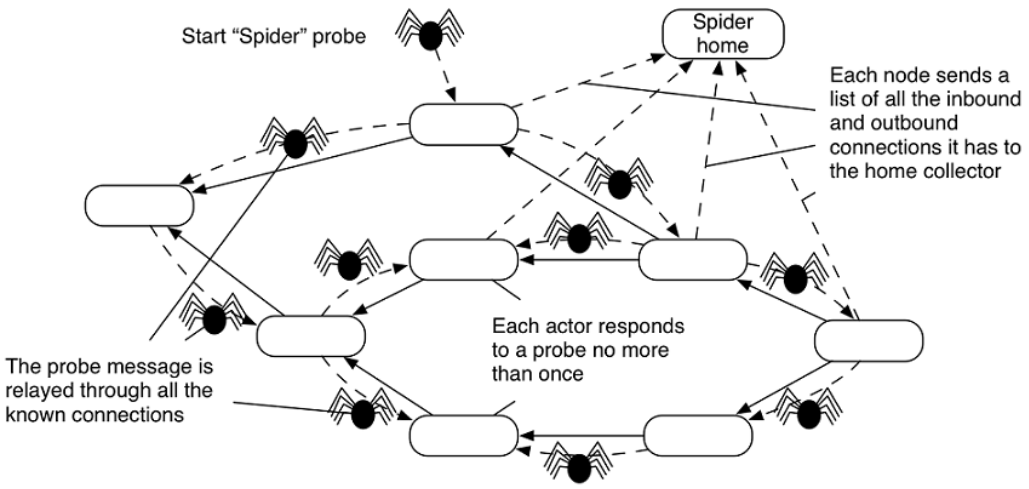

Figure 10.13 The spider pattern is useful for finding connections among actors

Implementing the spider pattern requires the communicating actors to record the *ActorRefs* for the other actors. A drawback of this approach is that the `Actor` trait does not have an easy extension point to intercept messages, so the instrumentation may intrude into the actor code.

The spider pattern is described in more detail by Raymond Roestenburg in a blog post http://letitcrash.com/post/30585282971/discovering-message-flows-in-actor-systems-with. The full post also describes ways this pattern can be extended to include diagnostic data, create a wiretap that copies all received messages, or perhaps kill slow actors.

Using the spider pattern to kill slow actors would require knowing how fast the actor normally should perform, so that unhealthy actors are identified while leaving healthy actors to complete. Understanding the overall health of the application requires monitoring, which is the topic of the next section.

10.5 Monitoring a Reactive application

If you add copious log messages to all the actors, turn on all the default logging, and configure the system to trace every message, you should be able to find bottlenecks where the application is slower than it needs to be. With all that instrumentation, the answer might well be that the logging is itself the bottleneck. At the other extreme, disabling the logging and tracing could make troubleshooting difficult or impossible.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

There is not a single correct answer to getting the right amount of monitoring, but there are some core metrics that have proven themselves useful: monitoring core metrics with Lightbend Monitoring and creating custom metrics with Kamon.

10.5.1 Monitoring core metrics with Lightbend Monitoring

In traditional systems, the emphasis is on monitoring indirect measures of system health: request threads, memory usage, I/O operations, and similar metrics. In contrast, the internals of an actor system focus on managing queues of work for actors to perform. The result is that monitoring focuses on counting things: number of actors, mailbox size, messages per minute, and the like. Figure 10.14 shows how this is reflected in the default actor metrics displayed by the Lightbend Monitoring dashboard.

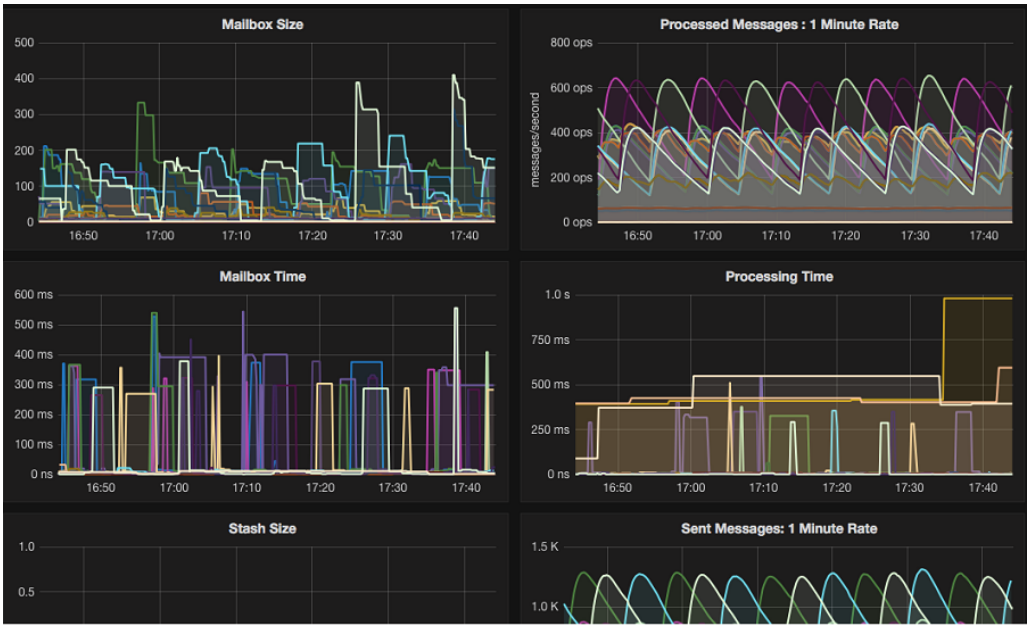

Figure 10.14 The Lightbend Monitoring dashboard emphasizes counts and rates

Reactive applications, particularly those based on strong domain-driven design, can use metrics that map more closely to the domain model. As you drill down from the dashboard into more detailed displays, Lightbend Monitoring will show metrics that are increasingly specialized to the actor system, such as how many actors there are of a type, or the mailbox size for a group of actors that share a router.

Because actors and messages map to the domain model, you can ask real-world "does this make sense?" questions about the metrics. For example, if there are 50% more "order" actors

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

than there are active customer orders, that could be something to investigate, or if 1% of users appear to be clicking 5 times or more per second that may indicate bot activity. The important thing is to remain curious about what the system is doing.

10.5.2 Creating a custom metric with Kamon

Kamon (www.kamon.io) is an open source tool for monitoring JVM applications. It integrates well with several popular toolkits for Reactive systems: Akka, Spray, Play Framework. Where it shines is its rich catalog of backend integrations. For simple development, you can use a backend that dumps metrics directly into application logs. In more advanced configuration, you can select from a variety of choices. Because JMX is widely supported by commercial monitoring tools, that backend may be used as a bridge into enterprise systems as well.

Another benefit of Kamon is support for custom monitoring metrics. You can create instrumentation for exactly the measurement that is important. A downside is that it requires some development, and the documentation is sparse in places. It requires some familiarity with AspectJ, and the internals of Kamon use their own Akka and Spray configurations.

10.6 Handling failure

Distributed systems typically do not fail all at once. More often, individual services fail. One of the difficult design problems in a traditional architecture is deciding which component is responsible for recovery of which other components. The result sometimes is an endless stream of error messages in the log, but no automated recovery. In an actor system, the answer is clear. Every actor is a member of a supervision hierarchy, and is responsible for all its child actors.

There is really only one recovery strategy, and that is to restart. Some basic design considerations are how much of the actor system to restart, how to prevent a large backlog of messages accumulating during a restart, and how to ensure data loss is zero or at least minimal. Roland Kuhn has developed a more advanced treatment of patterns for handling failure, and worked with me (Brian) and Jamie Allen to catalog them into *Reactive Design Patterns* (Manning Publications, https://livebook.manning.com/#!/book/kuhn/Chapter-1/).

10.6.1 Deciding what to restart

When a single child actor fails, the supervisor is notified through lifecycle events and decides what to do. The most important decision is whether the failure handling should be applied just to the failing actor or to all the other supervised children as well. Within that strategy, the configuration can provide responses specific to each type of `Throwable` that the supervisor receives.

This often is not enough to make the best decision. For example, if an actor experiences a networking error it could be a transient problem affecting just a single request, or it could be something more pervasive. If it the problem does not resolve itself right away, continually restarting a single actor or group of actors can exacerbate the problem. If there are multiple

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

actor systems using the same strategy, they may all restart at the same time. When the problem is cleared, it could lead to an immediate deluge of requests to the same service. One way to address this is with a backoff strategy. With this strategy, the supervisor adds a delay before restarting. The amount of delay increases randomly with each restart.

10.6.2 Routing considerations

An `ActorRef` survives a restart, so other actors continue sending messages, perhaps before the restarting actor is ready to process them. If an actor is going to take a long time to restart, you may wish to suspend sending it new requests for some period, as shown in figure 10.15.

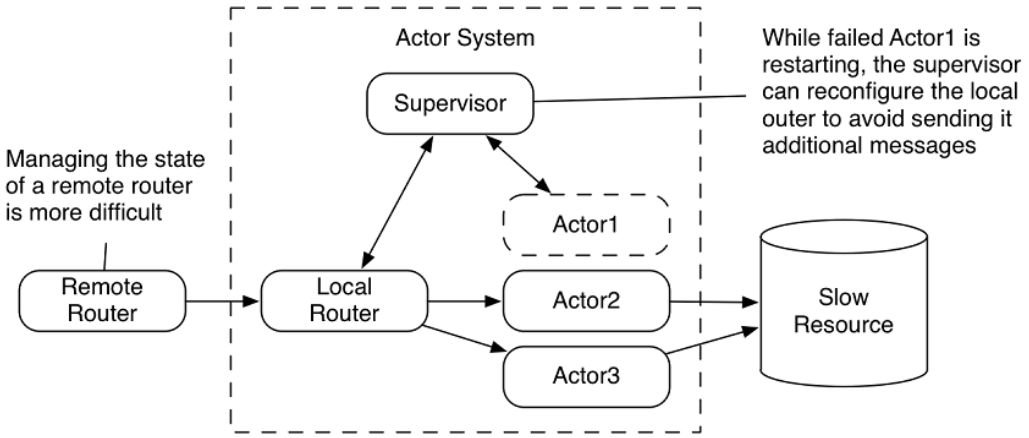

Figure 10.15 It may be necessary to suspend sending messages to a redundant actor if it will take a long time to recover

Keeping multiple remote routers apprised of the state of potentially multiple actors restarting can become very complex, especially considering that the network may have been the original source of the problem leading to the restart. Almost always it will be better to allow messages to continue to be queued during the restart, or to use a local router with a pool strategy that takes workload into account, such as a `SmallestMailboxPool`.

10.6.3 Recovering, to a point

The amount of time that is allocated to getting the system running again is known as the *recovery time objective* and is one of two critical measurements for failure recovery. The other is the *recovery point objective*. The recovery point measures how much information, if any, was lost forever because of the failure. See figure 10.16.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

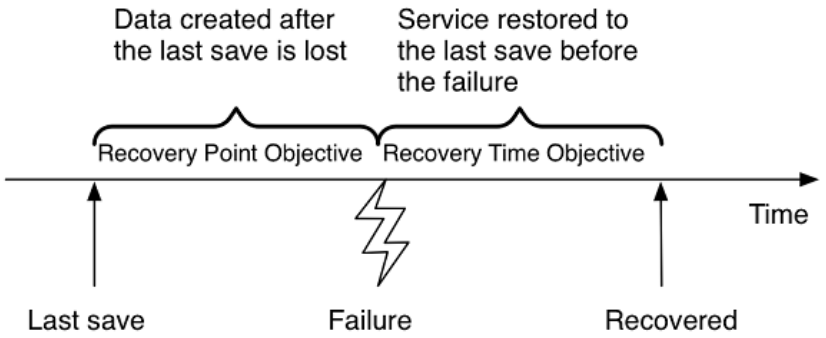

Figure 10.16 The recovery point objective measures how much data can be lost, and the recovery time objective measures how quickly the system is expected to recover to that point

In a CQRS design, the separation of recovery time and recovery point is made clear by the design itself. The command side is vulnerable to losing data from commands that had not yet been executed with the actor failed. The query side cannot lose any data, but building up a new cache can delay complete recovery.

In a traditional RDBMS design, the recovery point is defined by the last successful commit to the database. If the commit boundaries do not correspond to handling a complete message, the database could be left in an inconsistent state that requires manual intervention to correct.

Recovering data that has been saved to disk is outside the scope of this book. For the most part, that discipline focuses on ensuring that writes are redundant, so that failure of a single component does not lose data.

10.7 Deploying to a cloud

It is virtually certain that your application will need to be packaged for deployment to some sort of cloud environment, such as Amazon Web Services (AWS) or Google App Engine. At the very least, it would be convenient to generate a package appropriate for the host operating system, such as deb or rpm packages for Debian/Red Hat Linux systems, msi for Windows, or dmg for OS/X. All that and more can be accomplished using the sbt native packager.

Packaging the application alone is no guarantee that it will run successfully in every environment. It is very easy to create hidden dependencies on the runtime environment that constrain portability. One well-known approach to removing dependencies is the Twelve-Factor Application.

10.7.1 Isolating the factors

The *Twelve-Factor App* concept that was formalized by Adam Wiggins at www.12factor.net describes a methodology for building applications that are easily manageable in a Software-as-

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

a-Service environment. This approach was originally created based on experience with Heroku, but is equally applicable to other environments.

One of the factors is managing the codebase so that each component is a separately deployable application. This can be further refined in a reactive application.

MANAGING CODE DEPENDENCIES

As a reactive design evolves, actors may be moved from one actor system to another. Sometimes it makes sense to have both local and remote instances of the same actor available. In a well-designed application, it makes no difference to the actors how they are distributed. The deployment configuration can evolve as needed.

It would be inconvenient if the code base had to be refactored every time a deployment decision changed. At minimum, each actor system should be managed as a separate codebase. Use that codebase to contain the `main` function, extract environment variables, create the actor system, set up routers, start the initial actors, and perform any other housekeeping that would change if the deployment were to change. Figure 10.17 shows the general idea.

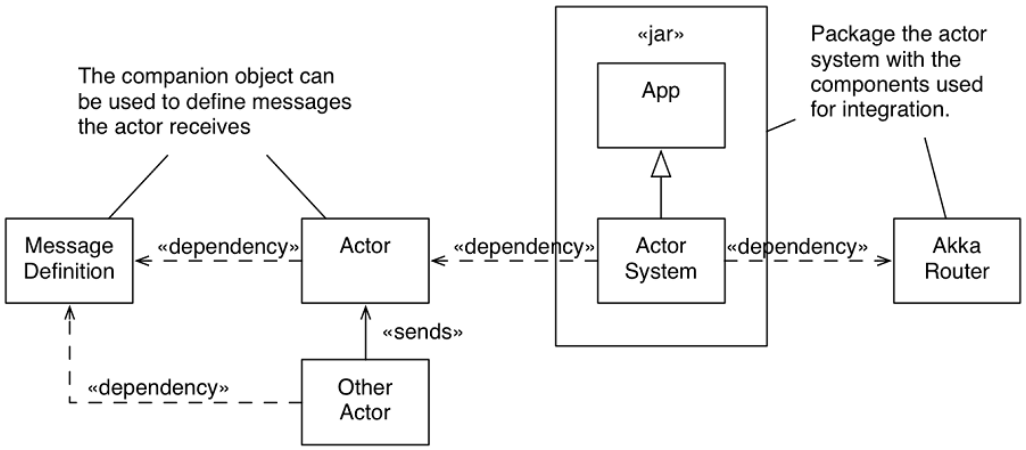

Figure 10.17 Organize the application build components so that collaborating actors can be organized into the same or different actor systems

The actors themselves can be packaged however it makes sense for the domain model. In relatively small applications it is convenient to define messages in the companion object to the actor that receives them. As the system grows, that can create unwanted dependencies between the interface contracts and their corresponding implementation, at which point it makes sense to refactor them into separate traits.

Eventually the application complexity may grow to the point that message serialization must be managed explicitly. Akka includes an extension point,

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

`akka.serialization.Serializer`, that can be used to configure a serializer or inject a custom implementation. It even allows different serializers to be used for different classes through the `akka.actor.serialization-bindings` configuration.

CONFIGURING WITH ENVIRONMENT VARIABLES

Akka's powerful configuration system allows many aspects of an application to be configured dynamically, without having to recompile the code. However, some aspects of configuration need to change in each deployment environment. For example, normally development and production require different URLs for data storage. It would be unwieldy and error-prone to keep separate application.conf files for each environment. Instead, those values should be provided by environment variables.

The syntax for referencing an environment variable within application.conf is straightforward, as shown in the following:

Listing 10.13 application.conf with environment variable substitution

```
akka {
    enabled-transports = ["akka.remote.netty.tcp"]
    netty.tcp {
      hostname = ${?HOST}          (1)
      port = 2552                  (2)
    }
  }
}
```

(1) The value of the `$HOST` environment variable will be substituted
(2) In a containerized environment such as Docker, use default ports and let the container handle port mapping

Once you have the environmental dependencies extracted from your application, it is ready for packaging. One of the more popular packaging options is Docker.

10.7.2 Dockerizing actors

The *SBT Native Packager* plugin makes it easy to build a Docker image for an application. This plugin is enabled by adding it to the project/plugins.sbt file, shown here.

Listing 10.14 project/plugins.sbt for the SBT Native Packager

```
addSbtPlugin("com.typesafe.sbt" % "sbt-native-packager" % "1.1.5")
```

Next, update build.sbt to enable the plugin, as follows.

Listing 10.15 build.sbt updates for the SBT Native Packager

```
name := "MyAppName"                (1)

version := "1.0"                   (2)

// …additional directives
```

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

```
enablePlugins(JavaAppPackaging)      ❸
```

❶ The name will be used as the default container name for the docker image
❷ The version will be used as default container tag for the docker image
❸ Enabling JavaAppPackaging automatically enables DockerPlugin and adds the archetype necessary for Akka to be packaged as a standalone application

Setting the docker tag

By default, the docker tag will be taken from the version property in build.sbt. If you prefer, you can override this value in build.sbt. For example, to have images always tagged as the latest in docker, add the following line:

```
version in Docker := "latest"
```

At this point the command

```
sbt docker:stage
```

will generate a Dockerfile and stage all the project dependencies in the target/docker/stage directory. If a Docker server is running locally, use the command

```
sbt docker:publishLocal
```

to take the additional step of building the image. Verify that it worked using the docker command, as follows:

```
$ docker images
REPOSITORY     TAG        IMAGE ID         CREATED           SIZE
myappname      1.0        fe20268f78d9     6 seconds ago     662.9 MB
java           latest     861e95c114d6     2 weeks ago       643.2 MB
```

Docker images can be used by many cloud providers. An easy way to get started is to install docker-machine and check the latest list of available drivers.

If your application will not be run in a Docker container, there are other packaging options available.

10.7.3 Other packaging options

The *SBT Native Packager* is built around format plugins and archetype plugins. Format plugins describe how the application files are packaged for different target systems. In addition to the docker format described in the previous section, it includes formats for Windows, Linux (Debian and RPM) and Oracle's javapackager tool. Archetype plugins are concerned with application structure and scripting. The archetypes cover a wide variety of situations, and are highly customizable to fit other situations.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development
Licensed to RODRIGO LARRIMBE CUITIÑO <ops@etermax.com>

10.8 Summary

In this chapter, you learned that

- The Akka test kit simplifies testing Actor behavior. Some commonly used patterns can be combined to create necessary test conditions.
- Application security begins with a threat model that is used to analyze different ways the application may be attacked. The STRIDE system provides a systematic way to manage the threat model.
- Encrypted transports can limit the attack surface of an application.
- Logging is a side effect just like any other I/O operation. Log messages should be treated the same as any other message in a reactive application. Akka includes built-in support for message-based logging using SLF4J and logback.
- Tracing involves instrumentation, a transport layer, a collector and query engine. Zipkin provides implementations all these, and there are other open source libraries that integrate with Zipkin.
- Kamon and Lightbend Monitoring provide monitoring metrics that are customized for Akka-based systems.
- Ability to recover from failure is measured by a recovery time objective and a recovery point objective. The recovery time measures how long it takes to become stable again. The recovery point measures how much data is lost.
- Dependencies on the environment should be factored out of the application and supplied by the runtime environment.
- The SBT native packager can assemble an application for deployment to a cloud container service in Docker format, and supports several other native package formats.

©Manning Publications Co. We welcome reader comments about anything in the manuscript - other than typos and other simple mistakes. These will be cleaned up during production of the book by copyeditors and proofreaders.
https://forums.manning.com/forums/reactive-application-development