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
brief contents

PART 1: FUNDAMENTALS

1 What is a reactive application?
2 Getting started with Akka
3 Understanding Akka

PART 2: BUILDING A REACTIVE APPLICATION

4 Mapping from Domain to Toolkit
5 Domain-driven design
6 Using Remote Actors
7 Reactive Streaming
8 CQRS and event sourcing
9 A reactive interface
10 Production Readiness
The words *react, reactive,* and *streams* are popular today. You may think that they’re the newest trends in programming, but though they *are* trendy, they’re hardly new. Reactive programming techniques, especially the actor model, are decades old. What has changed is that internet-scale applications are no longer limited to a few giant companies. Your application may have to grow from toy to powerhouse in far less time than you’d need to rewrite it.

Services such as Amazon Web Services (AWS) make adding servers easy, but that capability does you no good if your application isn’t designed to be scalable. The first part of this book delves into factors that can prevent your application from taking advantage of the additional power. Chapter 1 breaks down a traditional application and shows why more servers sometimes make performance even worse. It describes properties of an application that avoids these limits. Chapter 2 is a fast-paced introduction to the Akka toolkit. You start with a simple example that runs in a single process, and with a few small changes, you transform it into a flexible architecture that spans multiple servers. Chapter 3 examines the workings of the toolkit and addresses a problem that you probably didn’t encounter in chapter 2: handling failure gracefully. With this foundation, you’ll understand why reactive applications can withstand often-unpredictable challenges.
What is a reactive application?

This chapter covers

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

One of the most fascinating things in nature is the ability of a species to adapt to its changing environment. The canonical example is Great Britain’s peppered moth. When newly industrialized Great Britain became polluted in the 19th century, slow-growing, light-colored lichens that covered trees died, resulting in a blackening of the trees’ bark. The impact was quite profound: Light-colored peppered moths, which historically were well-camouflaged and in the majority now found themselves the obvious targets of many a hungry bird. Their rare dark-colored siblings, which had been conspicuous before, now blended into the 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.

What do moths have to do with programming? Moths in and of themselves aren’t particularly interesting in this regard, but how they adapt to their environment is. The peppered moth survived because a genetic mutation allowed it to react to its changing environment. Likewise, a reactive application 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; more on that later.

With the ever-growing complexities of modern computing, you must be able to build applications which display this trait. As user expectations of split-second performance, spikes in application load, demands to run on multicore 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’s designed from the ground up to meet them head on.

Although the peppered moth achieved 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 of reactive applications.

Sound programming principles such as message-driven, elastic, resilient, and responsive must be embedded in a reactive application’s DNA from the beginning. The following list defines these principles, which are contained in the Reactive Manifesto¹ (a blueprint for building reactive applications):

- **Message-driven**—Based on asynchronous communication in which the designs of sender and recipient aren’t affected by the means of message propagation, which means that 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 workloads. Reactive applications can actively scale up or scale down based on use or other metrics employed by system designers, saving money on unused computing power, but (most importantly) ensuring the servicing of a growing or spiking user base.

- **Resilient**—Reacting to failure. The system stays responsive in the face of failure. Failure is expected and embraced, and because 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**—Reacting to users. The system responds in a timely manner if at all possible. Responsiveness is the cornerstone of usability and utility; more than that, it also means that problems may be detected quickly and dealt with effectively.

Reactive applications aren’t boilerplate applications; they’re challenging to build. They’re designed to react to changes in their surrounding environment without new code, which is a hefty task. Additionally, they’re based on principles and techniques that aren’t new but are only now becoming mainstream.

Many current applications on the Java virtual machine (JVM) favor frameworks such as Spring and Hibernate, whereas reactive applications tend to favor toolkits such as Akka, which is both a toolkit and a runtime for building highly concurrent, distributed, resilient, message-driven applications.

Don’t let this new paradigm, with its use of robust toolkits such as Akka, give you pause. This book teaches you a very different way of building applications, embracing the traits listed

---

¹ We go into more detail on the Reactive Manifesto throughout the book. Jonas Bonér, Dave Farley, Roland Kuhn, and Martin Thompson contributed to this blueprint for building reactive applications, which you can find at [http://www.reactivemanifesto.org/](http://www.reactivemanifesto.org/).
earlier in this section. This book enables you to solve the complex problems associated with distributed systems, concurrent programming, fault tolerance, and more. This chapter introduces you to the key principles of reactive applications that we 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 past 50 years is the internet. The internet dates back to the 1960s, when the U.S. government commissioned research to build a robust, fault-tolerant computer network. This process began with a series of memos written by J.C.R. Licklider of the Massachusetts Institute of Technology in August 1962 and was known as the Galactic Network concept. Licklider envisioned a globally interconnected network of computers that allowed users to access data and programs from anywhere in the world. Licklider was director of the Information Processing Techniques Office within the Advanced Research Projects Agency, which we know today as the Defense Advanced Research Projects Agency (DARPA).

In 1964, MIT professor Leonard Kleinrock published the first book on packet switching theory. Kleinrock persuaded Licklider’s successor, Lawrence G. Roberts, that the theory of communicating via packets rather than circuits was the next major step for networking computers. To explore this theory, Thomas Merrill and Lawrence Roberts connected two computers—a TX-2 computer in Massachusetts and a Q-32 computer in California—via a low-speed dial-up line. This significant event represented the first wide-area network and allowed time-sharing-based computers to interchange data and run programs on a remote machine. This effort led in 1968 to a DARPA-funded RFQ known as ARPANET. The RFQ focused on the development of a key component: an interface for packet switches called Interface Message Protocol. In December 1968, Frank Heart of Bolt Beranek and Newman won the RFQ, working with the University of California-Los Angeles to bring the first computer node online in September 1969.

1.1.1 Distributed computing

The result of this work in the 1960s and 1970s in the United States, as well as some additional work in Great Britain and France, paved the way for what we know as the internet. This work also resulted in a new computer model known as distributed systems that represented a shift in the computing paradigm. Before distributed systems, the foundational computer model was large, expensive mainframe systems affectionately referred to as Big Iron.

Mainframes historically represent a centralized computing model that focuses on efficiency, local scalability, and reliability. Although this model is effective, it’s also expensive beyond the reach of many companies, with the cost of memory, storage units, processing cores, and the like running into millions of dollars. Distributed systems are a less expensive way to achieve and even exceed the raw computing power that typical mainframe configurations represent.
That being said, however, distributed systems don’t preclude mainframes. A distributed system might consist of mainframes, minicomputers, and personal computers. The goal of a distributed system is to network a group of computers to work as a single system. We cover distributed systems throughout the book, with special focus in chapters 3 and 4.

1.1.2 Cloud computing

The advent of distributed systems and continual progress toward more powerful, less expensive computing hardware paved the way for cloud computing. Cloud computing represents another significant paradigm shift in the way that computer applications are written and managed.

Whereas distributed systems focus on the technical details of interconnected independent computer systems, cloud computing focuses on economics. It represents a departure from the norm of managing, operating, and developing IT systems that provides substantial economic savings as well as greater agility and flexibility, and this trend is here to stay.

In January 2008, Amazon announced that Amazon Web Services (AWS) consumed more bandwidth than its entire global network of retail services, as shown in figure 1.1.

![Figure 1.1 AWS surpasses Amazon's entire global retail services network in 18 months](http://aws.amazon.com/blogs/aws/lotsofbits/)

This new landscape of distributed cloud computing represents a dramatic change for the modern programmer, much as the Industrial Revolution of the nineteenth century did for the peppered moth. Recent hardware enhancements such as multicore central processing units (CPUs) and multisocket servers provide computing capabilities that didn’t exist even five years ago.

Figure 1.2 shows the current state of storage, CPU, and bandwidth compared with the number of network nodes.
As the figure illustrates, the decrease in the cost of storage, CPU cycles, and bandwidth coupled with an increase in network nodes means that cloud computing is shaping up to be a competitive environment. The reactive paradigm is designed for this environment, providing ready distribution across this vast ocean of processing power while maintaining resilience and responsiveness.

The best way to understand the advantages of a reactive architecture over other approaches is to view a comparison example. Our example uses a construct that everyone is familiar with: a web shopping cart. We provide a simple example of a customer browsing online inventory, choosing items, and checking out, in both monolithic (applications in which all layers are mutually dependent) and reactive architectures, and we explore how each type of architecture 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 you dig into the comparison, you need to know a few things about the example shopping cart. On the surface, it seems to be a simple use case, but a lot more is going on than meets the eye. On the internet, the customer is king. As a result, modern retailers have to fight for customers; they need to have an edge that draws the customer base back. To facilitate this, online sites craft a scenario in which 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 requires interaction with other systems, managing responses and handling failure while the shopper is none the wiser. In a traditional monolithic application, these interactions may be slow or fail entirely, because they’re only as strong as their weakest link. The reactive design paradigm deals with these challenges in an isolating, succinct manner, maximizing overall performance and dependability.

1.2.1 Monolithic architecture: difficult to distribute

Since the dawn of web development, most web applications have been based on a monolithic architecture. A monolithic architecture is one in which functionally discernible aspects of the system aren’t architecturally separate. Figure 1.3 shows the example shopping cart as it might look in a monolithic architecture.
Figure 1.3 Shopping cart modeled in typical monolithic architecture style

As the figure shows, components such as data access, error handling, and user interface are 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 that it centers on a centralized relational database management system. As a result, you encounter the first of many challenges with this model.

The majority of relational databases today use synchronous blocking drivers for transaction management. This kills scalability because all the components must run in the same application space, such as a single JVM process. They typically share a connection pool and often become a
single point of failure. Optimization is commonly made over time by means of queueing technologies and enterprise service buses (ESBs) to orchestrate processes and allow system-to-system communication. Admittedly, this technique is an improvement, because blocking at service level is better than a blocked database transaction, but it’s not enough, because any blocking is bad blocking and always results in diminishing returns (if not reduced returns), as more hardware is thrown at the problem.

**TIGHTLY COUPLED MIDDLEWARE**

Next is 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 compromise scalability; even worse, they make it difficult to version an application programming interface (API). You can’t update only part of an API because of the interdependencies. More often than not, you have to reason about the entire system as a whole because of the rippling effect of tight coupling. The blocking nature of these communications can become a substantial bottleneck. You can imagine calls being made to the other service components to retrieve daily deals, reviews by other customers, images, and so on. To create the final composite view, you must wait until all associated data is retrieved, which can result in a delay.

To solve some of these problems, more often than not you enter the dark world of concurrent programming. You attempt to spawn threads, write synchronized blocks, lock on a mutable state, use atomic variables, and use the other tools provided in a threaded environment. Although threaded programming provides a useful abstraction for concurrent or parallel execution, the price becomes unsustainable. You begin to experience significant problems in understanding your code, let alone predicting or determining what it’s supposed to do, and as a result, your code becomes widely nondeterministic.

Edward A. Lee sums up this situation 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 domain model, usually built on top of an object relational mapping (ORM) abstraction layer, that implements a create, read, update, and delete (CRUD) process to manage the domain’s current state. As you see shortly, this CRUD pattern can significantly diminish the value of your data. Traditionally, these services are designed with strict dependencies on one another and rely on blocking I/O for communication.

³ The Problem with Threads, Edward A. Lee, Berkeley 2006
LOAD BALANCER

Finally, to support fault tolerance and load spikes, you must implement a load balancer, as shown at the bottom of figure 1.3. Although load balancers 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 only the failing part. For the architecture to be truly fault-tolerant, load balancers 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; it can’t be bolted on as an afterthought. Another technique that’s commonly used is server clustering. The challenges with this approach are that it’s extremely costly and that it can open the door to an ominous cascading failure scenario that brings down the entire cluster.

All this IO blocking at all levels has proved to be a bad thing that must be avoided at all costs, as dictated by Gunther’s and Amdahl’s laws, which we explain in the next section.

UNIVERSAL SCALABILITY LAW

Blocking of any kind anywhere in the system has been proved to measurably affect scale due to the following:

- Contention—the wait for queues or shared resources
- Coherency—the delay for data to become consistent

Originally, this effect was shown by computer architect Gene Amdahl, who theorized that blocking causes diminishing returns with regard to scaling. His theory became known as Amdahl’s Law. Neil J. Gunther, a computer systems researcher, further proved that blocking reduces concurrency as a system is scaled. This theory holds true today and is called Gunther’s Law, but is widely known as The Universal Scalability Law. The reason is that the cost of coherency as the system grows becomes a drag on the overall system and causes a loss over overall scale, as figure 1.4 shows.

![Gunther's Law vs. Amdahl's Law](http://cmg.org/publications/measureit/2007)

Figure 1.4 Gunther and Amdahl’s laws (http://cmg.org/publications/measureit/2007)
The figure overlays Gunther’s and Amdahl’s laws, clearly showing that whereas Amdahl theorized diminishing returns, Gunther proved that concurrency—and, therefore, scale—drop off after a point. This law means that no matter how much hardware you throw at a problem, you make a blocking system worse.

Consistency is another topic often taken for granted by designers of traditional monolithic systems, as tightly coupled services are connected to a centralized database. These systems default to strong consistency, because access to data (in terms of reads and writes) is guaranteed to be consistently ordered, which means that every read must follow the last write, and vice versa. This consistency model is great for always having single-point access to the latest data, but it has a high cost in terms of distribution, as identified by the CAP theorem.

**CONSISTENCY AND CAP THEOREM**

In theoretical computer science, the CAP Theorem (also known as Brewer’s Theorem) states that it’s impossible for distributed systems to simultaneously provide all three of the following guarantees:

- **Consistency**—All nodes see the same data at the same time.
- **Availability**—Every request is guaranteed to receive a response about whether it was successful.
- **Partition tolerance**—The system continues to function regardless of message failure or partial system failure.

Figure 1.5 shows a Venn diagram of these guarantees.

![Figure 1.5 The CAP Theorem Venn diagram](image-url)
As the figure shows, in distributed computing it’s not possible to have all your cake and eat it too. By design, distributed systems are asynchronous and loosely coupled, relying 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 you must decide whether to have higher availability or greater consistency.

**CONSISTENCY MODEL**

In distributed computing, a system supports a given consistency model if operations follow specific rules 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** or **linearizability** is the strongest method of consistency, guaranteeing that data reads reflect the latest writes across all processes. Strong consistency is incredibly expensive in terms of scale; therefore, you must avoid it at all costs in building reactive applications.

- **Eventual consistency** is a consistency model used in distributed computing. This model informally guarantees that if no new updates are made to a given data item, all accesses to that item **eventually** 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 **Basically Available Soft state Eventual consistency semantics** as opposed to more traditional atomicity, consistency, isolation, and durability (ACID) guarantees—a key point and one of the key factors that allow distribution.

- **Causal consistency** is a stronger consistency model that ensures that operations are processed in the expected order. Causal consistency is the strongest method of consistency achievable while retaining availability. More precisely, partial order over operations is enforced through metadata. If operation A occurs before operation B, for example, any data store that sees operation B must see operation A first. Three rules define potential causality:
  
  - **Thread of execution**—If A and B are two operations in a single thread of execution, 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, A → B.
  - **Transitivity**—For operations A, B, and C, if A → B and B → C, A → C. Thus, the causal relationship among 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 any transaction has multiple distributed parties.

Even Akka (as you see in the reactive architecture model of the shopping cart in the next section) doesn’t 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.

**Why Akka?**

If you look at building a system in terms of building a house, you clearly see that the tools are of utmost importance in guaranteeing success. A craftsman builder accumulates tools over the years, always getting more and better tools that prove to get the job done. Craftsman software programmers do the same thing, in that we learn constantly and gain access to new technologies for building software. Akka is an important item in a programmer’s toolkit because it’s a runtime and software library for building highly concurrent, distributed, resilient, 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, and creates new actors—all asynchronously, without locks.

**Akka’s value proposition**

Akka is a single, unified programming model that provides the following:

- **Simpler concurrency**—Code written with the illusion of single-threadedness, without locks or synchronized or atomic variables
- **Simpler distribution**—Code distributed by default, remote, or local configuration
- **Simpler fault tolerance**—Communication decoupled from failure through supervision

As a result of the challenges of concurrency, nondeterminism, consistency guarantees, and other rapid technology changes (such as multicore processors and pay-as-you-go cloud services), monolithic architectures don’t translate well to the modern world of distributed computing. Problems with concurrency, transaction management, scalability, and fault tolerance are rampant.

In the next section, we look at reactive architectures and show how they fare against these challenges.

**1.2.2 Reactive architecture: distributable by default**

Reactive applications adopt a radically different approach from monolithic ones. Rather than build an architecture based on a nondistributed environment and then try to retrofit it with locks for concurrency, load balancers, and so on, reactive applications assume a distributed environment.
From the ground up, reactive applications bake in the four key traits explained earlier in this chapter: message-driven, elastic, resilient, and responsive. Figure 1.6 shows the example shopping cart in a reactive application.

If this architecture seems to be more complicated than the monolithic one, that’s because it is. Distributed applications aren’t easy to build, but with the advent of Akka, the task is a bit easier than it once was. For brevity’s sake, we’re showing only the order microservice; structurally, the inventory, review, and other services would be identical.

Looking at the figure from the top down, you see the following:

- **Order service**—The first thing you notice is that the top of the figure doesn’t have a single centralized data store, like the monolithic example in figure 1.3. The order service is split into two sides: a command side and a query side, each supported by a clustered NoSQL data store and sitting atop its own JVM. This pattern is commonly referred to as

---

**Figure 1.6 Shopping cart modeled in reactive architecture style**

<table>
<thead>
<tr>
<th>Shopping cart</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1x Product one</td>
<td>$10.00</td>
</tr>
<tr>
<td>1x Product two</td>
<td>$5.25</td>
</tr>
<tr>
<td>2 items Total: $15.25</td>
<td></td>
</tr>
</tbody>
</table>

View cart Checkout
Command Query Responsibility Segregation (CQRS). We break this pattern into its key parts (C, Q, R, and S) in chapters 5, 6, and 7.

Each side of the order service is micro in nature and sits on top of a clustered Akka instance. The concept you should focus on for now is designing your application as a suite of small services, each running its own process, loosely coupled and communicating with a lightweight, message-driven process—in this case, Akka. These services are wrapped by the Akka microkernel, which offers a bundling mechanism and is distributable as a single payload. You don’t need a Java application server or a startup script.

- **Loosely coupled command-side microservice clusters**—The command side uses Akka persistence as its storage mechanism and Akka HTTP to process commands from the user interface (UI). Akka persistence provides durability to your application by persisting the internal state of each actor, allowing for recovery when the actor is started, restarted after a JVM crash or by a supervisor, or migrated in a cluster. Akka persistence is the foundation of resilience in the Reactive Manifesto.

  Akka HTTP provides an actor-based, asynchronous, lightweight, fast REST/HTTP layer for your application. We explore the command construct in detail in chapter 3.

- **Loosely coupled query-side microservice 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 via a consistency model, a common technique used in distributed computing to keep isolated systems synchronized. We discuss this model in relation to the CAP theorem earlier in this chapter. Consistency models can be *eventual* (eventually, all accesses to that item return the last updated value) or *causal* (the operations are processed in the expected order).

  We talk more about the importance of consistency models in chapters 3 and 5, but for the time being, you can think of consistency as being the logical glue that holds the command and query sides together. Consistency is essentially a contractual agreement that says that whatever happens on the command side makes its way to the query side, bound by some metrics such as content and time. We explore the query construct in detail in chapter 5.

In the next section, we look in depth at the principles of the Reactive Manifesto, which we introduce at the beginning of this chapter, and show how the reactive architecture is based on these principles.

### 1.2.3 Understanding the reactive architecture

The authoritative guide for reactive architectures is the Reactive Manifesto. Like many of us, Jonas Bonér, chief technology officer of Typesafe, grew increasingly frustrated with the way that modern applications were architected; he felt the need for a clear, concise way to
articulate the philosophies of good distributed design. As a result, the Reactive Manifesto was released on September 23, 2013 (V1); it was updated on September 16, 2014 (V2).

The manifesto centers on four attributes that lay the foundation for reactive applications, as shown in figure 1.7.

![Figure 1.7 Traits of the Reactive Manifesto (www.reactivemanifesto.org/images/reactive-traits.svg)](www.reactivemanifesto.org/images/reactive-traits.svg)

We discuss the four traits shown in the figure earlier in the chapter. In the following sections, we explore their implications for reactive design.

**MESSAGE-DRIVEN**

Message-driven architectures are loosely coupled, asynchronous, and nonblocking. Before we go any further, we’ll define what these terms mean, as they’re paramount to the concept:

- **Loosely coupled**—A system whose components depend on one another by the least amount practical
- **Asynchronous**—Capable of executing a task without waiting (nonblocking) for it to complete
- **Nonblocking**—Never waiting for a task to complete

This pattern results in no concrete dependencies and allows the use of a distributed domain model, which is crucial for scalability and which leads to lower latency and higher throughput. As a result, reactive architectures are naturally scalable, capable of elastically scaling in and out. This type of architecture mitigates financial risk, allowing for on-demand use of hardware and services such as those made popular by Amazon. When load is low, you spin down services, and when load spikes, you spin services back up. Because you’re paying only for what you use, you save money. We cover the details of distributed domain modeling in chapter 4.
Elastic architectures are key for distributed computing. They’re expected to expand and contract as load demands change through elasticity—the ability to add or remove nodes on the fly. This unique feature allows these architectures to scale in and out and up and down, without the need for the application to be redesigned or rewritten. Elasticity also mitigates risk in that hardware can be used on demand, which eliminates the need to keep a bank of unused servers waiting for a load spike to occur. The technique that allows for this elastic behavior is location transparency, which uses logical names to find network resources, removing the need to know the physical locations of the users and the resource. We cover elasticity in detail in chapters 2 and 3, which explore the Akka actor model.

Resilient

Reactive applications don’t use traditional fault-tolerance techniques. Instead, they embrace the notion of resilience. Merriam-Webster defines resilience as

- 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 such as the bulkhead pattern, which allow the application to self-heal. An example might be the shopping cart.

Imagine a scenario in which the shipping module fails temporarily. In a reactive system, the user can still interface with the shopping cart, adding and deleting items, while the shipping module (in the background) identifies failure and repairs itself. We discuss resilience in chapters 3, 4, and 5, where we deal with the distributed domain model and other error-recovery concepts.

Responsive

Finally, reactive applications are responsive. Users aren’t interested in what your application does under the covers; they expect it to work the same way in high-load and low-load situations and in failover and nonfailover modes. Today’s applications are expected to be real-time, engaging, and collaborative, capable of responding to a user’s actions without hesitation. If the shipping module fails, for example, the application continues to respond. Reactive applications use stateful clients, streaming, and observable models, among other things, to provide a rich, collaborative environment for the user. We cover these concepts in great detail in chapters 7 and 8.

We’ve introduced a large number of concepts in this section. Most important, we showed you how to use asynchronous message passing and share-nothing designs. Don’t worry; we cover all these concepts in detail throughout the book. For now, all that matters is that you understand the general concepts of a reactive architecture.

In the next section, we dig a little deeper into the specifics of implementing these two architectures in the shopping cart example. We look at the details of placing an online order in
both monolithic and reactive shopping carts, showing the distinct advantage of the reactive paradigm and the message-driven trait of reactive applications, which is distinctly different from the monolithic approach.

### 1.2.4 Monolithic shopping cart: creating an order

As we note earlier in this chapter, the architectural problems of concurrency, scalability, and fault tolerance are significant in monolithic applications, but other challenges arise as well. One such challenge is the value of the data that 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 understand this problem, take a close look at a customer adding items to the example shopping cart in monolithic and reactive designs.

In a monolithic architecture, you typically build your shopping cart application in a client-server fashion, using CRUD to manage the current state of your domain model. A customer browses available inventory, chooses four items, adds shipping information, and then checks out. Meanwhile, many other things are happening in the background. Images are being fetched; reviews are being loaded, daily deals are being presented, and so on. The application needs to deal with all these activities. To keep the example simple, we focus in this section on the problem of data persistence in monolithic applications.

This order is most likely wrapped in a single transaction via an ORM implementation that inserts the values into three tables: order, order_item, and shipping_information. The information stored represents the current state of the shopping cart order, as shown in figure 1.8.

![Figure 1.8 CRUD shopping cart current state after create](image-url)
At some point in the future, before the order is shipped, the customer decides that he no longer wants one of the items he ordered. He logs back into the shopping cart application, fetches his order, and deletes the unwanted item. At that point, the order consists of three items that cost a total $47 (figure 1.9). The notion that item 2 was deleted is lost.

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 customers before orders ship. Therein lies the rub!

**THE PROBLEM: USER INTENT NOT CAPTURED**

Because the domain model by way of CRUD stores only current state, the deleted data is lost. The development team has to add this task to a future sprint and implement an audit log that tracks deleted items. Even worse, after the log is implemented, the team can track only deletions from that point forward, which has substantial implications for the value of the data.

You should look to capture the intent of your users because, from a business perspective, customer behavior is paramount. Rather than model your domain as a current state model, you 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 you’re so familiar with, does capture behavior, but the behavior that it captures is system behavior in the form of creating, reading, updating, and deleting, which doesn’t tell you much about your users and compromises the value of your data. Most systems today rely on this model primarily because of the general acceptance of the relational database management system (RDBMS) as the center of web architecture. Fortunately, this way to view persistence isn’t the only way.

**Event Sourcing**, or persisting a sequence of events (behaviors)—not to be confused with message-driven, which means reacting to a message—provides a means by which you can...
capture the real intent of your users. In an Event Sourcing system, all data operations are viewed as a sequence of events that are recorded to an append-only store. In this section, we provide two examples that best showcase the capabilities of Event Sourcing: the canonical example of a bank account register and the CRUD shopping cart redone as a reactive shopping cart.

What’s the difference between messages, commands, and events?

The distinction between messages, commands, and events is important, and one that we need to make before going too far into our architecture discussion. Messages can come in two flavors: abstract and concrete. Following are two simple examples:

- You can think of an abstract message as being a blank sheet of paper—a structure to capture a conversation between two parties. The paper in and of itself isn’t a conversation until something is written on it. To start the discussion, suppose that we write on the paper a request to borrow a book from you. The abstract message becomes (has been implemented as) a command—a request to do something. In response, you write back that you’ve mailed the book. At this point, the abstract message has become (has been implemented as) an event—a notification that something has occurred. In this example, the command and event are forms of messages. In computing lingo, they implement the message interface.

- A concrete message is like an envelope—a container that has a payload. That payload can be anything. In the preceding example, the payload is either a command or an event. The distinction is that a message is concrete, like a command and an event.

1.2.5 Event sourcing: A banking example

In a mature business model, the notion of tracking behavior is quite common. Consider the bank accounting system shown in figure 1.10. This system allows customers to make deposits, write checks, make withdrawals, transfer monies to another account, and so on.

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

Figure 1.10 Bank account register transaction log with five transactions
The figure shows a typical bank account register in which the account holder deposits $10,000, writes a check for $4,000, makes an ATM withdrawal, writes another check, and makes another deposit.

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

Suppose that the bank persisted only current state for the account. When the account holder tries to reconcile her account, she notices a discrepancy. She double-checks her reconciliation and concludes that the bank made a mistake. She quickly calls the bank and states her case, and the bank officer promptly replies, “I’m sorry; we have no record of that transaction. We only store the last update to your balance.”

That scenario is ludicrous. Although it’s an extreme example of losing the user’s intent—changes in the balance—unfortunately, this situation happens in a CRUD-based monolithic application.

1.2.6 Reactive shopping cart: creating an order with Event Sourcing

Another way to look at events (which you might think of as being transactions) is to look at them as notifications that something has happened. Events are indicative or evidential in mood, as they state recorded facts. We discuss in depth details such as these for Event Sourcing, especially in relation to CQRS and commands in general, in chapters 5 and 6.

For now, we dig back into the shopping cart example by modeling it in an Event Sourcing fashion, as shown in figure 1.11.

![Figure 1.11 A reactive shopping cart stores events.](image-url)
As you can see, the workflow addresses the same concerns as the earlier CRUD example, with crucial differences:

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

As figure 1.11 shows, the reactive shopping cart has no current state of the order or line items that persist. Instead, it stores in order a sequence of deltas that capture user behavior. Note the indicative tense of an event, such as item added or shipping information added. Events are things that have happened, which is an important distinction compared with commands. You can reject a command because it’s a request to do something, whereas you can’t reject an event because it represents something that has already occurred.

To wrap up the discussion of the differences between monolithic and reactive applications, figure 1.12 illustrates creating an order in shopping carts created in both types of applications.
The rest of this section shows what happens when at some point in the future, before the order is shipped, the customer decides that he no longer wants one of the items he ordered. The customer logs back into the shopping cart application, fetches his order, and deletes the unwanted item, as shown in figure 1.13.
Again, the workflow is similar to the CRUD example, with subtle but crucial differences:

- No total is generated.
- The delete is a distinct delta that persists at the end of the event stream.

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 customers before orders ship. From a data perspective, this situation is one in which a reactive application shines:

- You have everything you need to craft the report because you capture the intent of the user in the form of events, rather than the current state of the model.
- Deletes aren’t updates to current state, as with a CRUD solution; they’re simply events captured in a user’s behavior workflow.
- In a reactive system, deletes are explicit and verifiable, whereas in a CRUD solution, they’re implicit and require tracking.

Figure 1.14 compares monolithic and reactive carts dealing with a deleted item.
So far, we've shown you how a reactive architecture solves several issues in reacting to events and transactions: the first characteristic of reactive applications as defined by the Reactive Manifesto. In the next section, we show you what else reactive applications can react to.
1.3 What are reactive applications reacting to?

A coming tidal wave will have a significant effect on application design. This tidal wave is the Internet of Things (IoT). Some theorists believe that by 2020, the internet will be comprised of 30 billion interconnected devices, as shown in figure 1.15.4

Consider the impact of 30 billion devices. Currently, approximately 12 billion devices are interconnected, and you can already see the result: slow websites, longer down time than expected, email interruptions every couple of months. The effect of tripling the number of devices in seven years will be staggering. To make matters worse, an estimated 99 percent of the physical objects (homes, cars, buildings, wearables, and so on) that may someday join the internet are still unconnected.5

---

4 ABI Research, https://www.abiresearch.com/press/more-than-30-billion-devices-will-wirelessly-conne/
The IoT will have a dramatic effect 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, in turn affecting every company’s bottom line.

If modern programmers are going to be successful (not succumb to the fate of the light-colored peppered moth during the Industrial Revolution), they must learn new tools and techniques designed for this new environment, which embrace distributed systems and cloud computing. This is exactly what reactive applications are all about.

1.4 What you will learn in this book

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

Following are those philosophies, patterns, and technologies broken down by the traits of the Reactive Manifesto to give you a better sense of where we’re heading in the following chapters. Strap in; you’re in for a great ride.

1.4.1 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 (chapters 2 and 3)
- **Akka HTTP**—Embeddable HTTP stack built entirely on Akka actors
- **CQRS-ES**—A set of patterns communicating via event messages (chapters 5, 6, and 7)

1.4.2 Elastic

Elastic means being 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 automatically distributed across multiple nodes in the cluster
- **Akka Streams**—A streaming model that protects each consumer of data from being overwhelmed by its producer by propagating backpressure
- **CQRS**—An approach in which models used for commands (write) are different from models used to query (read)
- **Distributed domain-driven design (DDD)**—A distributed approach to software development for complex needs that connects the implementation to an evolving model
- **Elasticity**—The expansion or contraction of the system according to load
• *Event Sourcing*—Persisting a sequence of behaviors

### 1.4.3 Resilient

More than fault tolerance, *resilience* is 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*—Enabling 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 automatically distributed across multiple nodes in the cluster
- *Akka Streams*—A streaming model that protects each consumer of data from being overwhelmed by its producer by propagating backpressure
- *Failure detection*—Responsibility for detection of node failures or crashes in a distributed system
- *Modular/microservice architecture*—A way of designing software applications as suites of independently deployable services (covered in chapter 8)

### 1.4.4 Responsive

*Responsiveness* is the ability to respond regardless of circumstances.

- *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*—An embeddable HTTP stack entirely built on Akka actors
- *Akka Streams*—A streaming model that protects each consumer of data from being overwhelmed by its producer by propagating backpressure

### 1.4.5 Testing

- *Test-driven development (TDD)*—A software development process that relies on the repetition of a 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.
- *Behavioral-driven development (BDD)*—A software development process that combines the general techniques and principles of TDD with ideas from domain-driven design and object-oriented analysis and design to give software development and management teams shared tools and a shared process for collaborating on software development.
- *Test kit*—A test kit that provides all the means necessary to test actors asynchronously, mimicking how they behave in the real world at runtime.
• **Multi-JVM testing**—A process that supports running applications (objects with main methods) and ScalaTest tests in multiple JVMs at the same time; useful for integration testing in which multiple systems communicate.

### 1.5 Summary

- The Reactive Manifesto is
  - Message-driven
  - Elastic, expanding and contracting with load
  - Resilient in the face of failure
  - Responsive to users

- Traditional, monolithic architectures has pitfalls and limitations.
- Blocking limits concurrency, and, therefore, distribution and scale.
- Reactive design solves the distributed programming problems existing today.