Reactive Programming with Scala and Akka

Today's web-based applications need to scale quickly to tackle the demands of modern users. Reactive programming is the solution developed to ensure the fault tolerance and robust scaling that is essential for professional applications. Reactive programming in Scala and Akka provides a great platform to develop low latency, resilient, concurrent Internet-scale applications on a Java Virtual Machine.

This comprehensive guide will help you get to grips with the concepts of reactive programming in order to build a robust distributed system in Scala and Akka. Written in two parts, you will first take a walkthrough of the reactive, asynchronous, and functional concepts of Scala before focusing on Akka and getting to grips with the details of real-world use cases.

By progressively working through the Akka concepts, you will not only be able to write your own distributed system, but also appreciate the hidden complexity within the Akka ecosystem.

Who this book is written for

If you are a developer who is passionate about building fault-tolerant, scalable distributed applications using Scala and Akka, then this book will give you a jump start. You should be familiar with Scala, but no prior knowledge of Akka and reactive programming is required.

What you will learn from this book

- Explore functional programming using Scala
- Design an asynchronous, non-blocking shopping cart application using Futures
- Understand the Akka actor model and the relationship between actors and threads
- Use the Actor Supervision feature to build a fault tolerant and resilient application
- Create your own distributed system framework using an Akka cluster
- Take a look under the hood to gain perspective on the Akka engine
- See a comprehensive case study of a key value store with concurrent reads and writes
- Model a finite state machine using state-driven actors

In this package, you will find:

- The author's biography
- A preview chapter from the book, Chapter 1 'Introducing Reactive Programming'
- A synopsis of the book’s content
- More information on Reactive Programming with Scala and Akka
About the Authors

Prasanna Kumar Sathyanarayanan is a Scala and distributed systems enthusiast living in Bangalore and has more than 10 years of experience.

In his daily job, he designs and writes concurrent code with Scala and Akka to implement big data systems.

He is a BTech graduate from Anna University. His interests include distributed systems, data engineering, functional programming, and machine learning.

Prasanna likes to contribute to open source projects. In his leisure time, he likes long rides on his bicycle and nature photography. He posts blogs at blog.prassee.me.

Suraj Atreya holds a master’s degree in computer science from the University of Edinburgh and has more than 7 years of industry experience. He has written software ranging from low-level 8-bit wireless sensor networks to systems scaling hundreds of nodes within a distributed system environment.

Fascinated by the early Pentium-1 computer that his mother presented him, he started learning LOGO and BASIC. Later, he grew up writing C and C++. His interests include functional programming, distributed systems, and designing data pipelines.

Suraj currently works for Glassbeam and loves using Scala and Akka for day-to-day data hacking. He has contributed to NeverWinterDP, a piece of open source data pipeline software. He text-mines in his free time and experiments with Haskell. He writes blogs at surajatreyac.github.io.
Modern Internet applications not only have a need to scale quickly, but also need to do so in a fault-tolerant and robust manner. Akka provides a great toolkit for developing low-latency, resilient, and concurrent Internet scale applications on the JVM.

This book offers a comprehensive coverage of a large number of feature sets in Akka, with carefully crafted examples for developers who are beginners and experts alike.

Dedicated chapters are provided so that you can jump-start your Scala and functional programming skills. By progressively working through the basics of Akka, you will not only be able to write your own distributed system, but also appreciate the hidden complexity within the Akka ecosystem.

What this book covers

Chapter 1, Introducing Reactive Programming, discusses the driving principles of reactive programming with an example and related methods of building a reactive application.

Chapter 2, Functional Reactive Programming in Scala, is where you learn how to think in a new paradigm called reactive programming, built using functional programming constructs.

Chapter 3, Asynchronous Programming in Scala, helps you identify the need for asynchronous behavior in your application, and explores Futures to an extent that will help you write an asynchronous non-blocking web crawler.

Chapter 4, Akka for Reactive Programming, introduces the Akka framework. This chapter gives a broad, high-level introduction to what Akka is capable of doing and it also shows how actors and threads are related and how the dispatcher plays a key role in ActorSystem.
Chapter 5, *Building Resilient Application with Akka*, introduces Akka's internal constructs, which are very necessary for building applications that can benefit from reactive design.

Chapter 6, *Akka Cluster*, shows the different components that make up an Akka cluster and how Akka supports building resilient distributed systems.

Chapter 7, *Finite State Machines with Akka*, covers finite state machines and how they are helpful in modeling systems. It also briefly describes Akka's agents and how to use them to connect to an actor.

Chapter 8, *Akka Unit Testing*, deals with various ways of writing unit tests in Scala and Akka. Unit testing is vital for every code base to help you design and build code in the right manner.

Chapter 9, *Distributed Systems and Key-Value Stores*, discusses some challenges faced while designing a highly scalable system. It also covers the CAP theorem, which is one of the main theorems behind the design of distributed systems.
Introducing Reactive Programming

Welcome to this book about reactive programming in Scala. This chapter will introduce the concepts of reactive programming by walking through the ideas that are rooted in the use of this new paradigm. This chapter also lays the foundation text for the rest of the chapters of this book.

In this chapter, we will discuss the foundations of reactive programming and we will approach reactive programming at a high level. We will also discuss the reason you need to learn this new style of programming and how it will help in your next project. Here is an outline of this chapter:

• A gentle introduction to discuss reactive programming from a high-level perspective
• The importance of reactive programming and what is driving reactive programming these days.
• Discussion of the four tenets of reactive programming:
  ○ Responsive
  ○ Resilient
  ○ Elastic
  ○ Message-driven
• Practices that complement reactive programming, such as functional programming, asynchronous programming, and RESTful services
• Some real-world examples of reactive applications
Introducing Reactive Programming

- Scala as a choice for reactive programming. We will explain why Scala is the choice while writing reactive code.
- Our tools. These will be the various tools and development environment that we will use to explain reactive concepts throughout the book.

Introduction

Reactive programming is a rage now. Although there is a reactive manifesto written, it is mostly a set of desired properties outlined in general terms. With many companies trying to tackle data-centric problems, the need for robustness, flexibility, and resilience becomes crucial. With growing data needs, building applications that are responsive and scalable becomes a challenge too.

Many organizations that deal with a huge amount of data address all of these challenges day in and day out. Millions of people tweeting across the world at once and simultaneously delivering each tweet to its respective followers all in near-real time is a huge challenge. This requires not only a strong backend design but also adherence to near-real-time constraints. A lag of a few seconds on an e-commerce site, such as Amazon, is enough to drive a customer away and make them jump on to another retailer to purchase a product. Keeping these common patterns in mind, the reactive manifesto was born. This book takes you through the journey of reactive programming using Scala and Akka as the vehicle.

The rise of the data age

We are generating huge volumes of data these days and it is growing exponentially every year. This ranges from smartphones to various sensors. Somehow, we also need to analyze data speed from these devices for analytics, for example, to track daily workouts such as miles jogged and analyze how many calories were burnt. While driving to various places, with navigation apps such as Google Maps, navigation is reported and used by Google to predict the traffic for us.

It is certain that to make the most of data, we need to get many insights in a very short interval of time. This can be achieved only by analyzing it deeply. One more factor to consider here is that there could be a huge amount of data generated in a short span of time, and hence such data processing systems need to be scalable as well. Let's go back in time to see ourselves how we got to the present data age:

- **1997-1999**: During the late nineties:
  - The Internet had 280 million users
  - J2EE was still an emerging paradigm
  - Online banking was in its infancy — 5 years old, give or take
• 2000-2005: This time period wasn't very long ago, but the world of computing and the Internet looked quite different. J2EE, SOA, and XML were the buzzwords:
  - The Internet had 1 billion users
  - Facebook had 5.5 million users
  - YouTube was a newborn (February 2005)
  - Twitter wasn't alive yet (2006)
  - Netflix had yet to introduce video streaming (2007)

• 2006-present: In 2014 — at the time of writing this book — there are approximately 2,950,000,000 (2.95 billion) Internet users:
  - Facebook has approximately 1.3 billion users
  - Twitter has approximately 270 million users

If we compare the number of Internet users in the past decades with the number of users of the two aforementioned websites, we can see that they now handle as much traffic as the entire Internet used to.

Tenets of reactive programming
Reactive programming follows certain principles in order to develop applications. We call them tenets of reactive programming. While writing applications that are reactive, we take a bottom-up approach. The goal of this section is to explain how an application can achieve this. In this section, we will detail each of the tenets and their respective properties.

How do tenets blend together?
The following four tenets exhibit the fundamental traits upon which an application should be built:

• Message-driven
• Scalable
• Resilient / Fault tolerant
• Responsive
Introducing Reactive Programming

The goal of a reactive application is to be **responsive**, however, this is impossible to achieve without addressing the **scalable** and **resilient** properties, and these in turn depend on the bottommost and the fundamental **message-driven** property.

The tenets of reactive programming

An example use case

Discussing the tenets of reactive programming can be very abstract, so to make it a bit interesting, let’s take an example of an e-commerce (imaginary) site called **RetKart**; it sells things online. This imaginary site serves as a vehicle to discuss all of these tenets in order to get architectural-level ideas to achieve it as a responsive site.

Responsive

Responsiveness is a crucial factor for ensuring consistently positive user experience. The experience of an application under various conditions, such as failure of an external system or a spike in traffic, depends on the three traits of a reactive application: resilience, scalability, and message-driven.

For instance, RetKart is highly popular and there is an increase in the number of visitors for the site. Hence, the goal is to provide a consistently positive user experience, regardless of certain factors such as:

- The device where the site is being accessed, whether it is a desktop, tablet, or mobile device
- High availability; that is, the system should always be available under very peak user traffic
Consistency is critical for e-commerce sites to deliver, given the various options that exist for the purpose of accessing the site and placing an order. A poor experience is not forgotten or ignored simply because the experience happened online rather than in a brick-and-mortar store.

In our case, let's assume that a site offers a seasonal sale on a stipulated date. It becomes obvious that many people start visiting the site to check out the offers on various items, and as a consequence, the site experiences a major spike in traffic.

Let's now consider the user experience. If a user browses the site on the aforementioned day, maximum responsiveness is expected and the user would like to place many orders (before the item runs out of stock!). This should be the same positive experience for another user at the same time. As discussed in the example, while dealing with such traffic, the e-commerce site should provide a responsive, positive experience.

**Resilient / fault tolerant**

Most applications are designed and developed for acceptable traffic, but things can—and do—go wrong. It seems that very frequently, there could be a report of a major application failure or another coordinated breach of systems by hackers, which results in downtime, data loss, and also damaged reputation. A resilient system applies proper design and architecture principles in order to ensure responsiveness under all sorts of traffic and unexpected conditions.

Today, applications are composed (rather than built) of a number of other applications, integrated via web services and other network protocols. An application built today may depend on a large number of external services, such as OAuth for authentication and a payment gateway. It may also serve a large number of external clients, both people and other systems.

Resiliency is one of the weakest links of the four tenets. Modern applications must be resilient at their core in order to stay responsive under a variety of real-world, less-than-ideal conditions. Performance, endurance, and security all complement resiliency.

Assume that RetKart has scaled out for a massive 100 nodes. Now, there is a high probability of some of the nodes failing at any point in time (this can happen due to various reasons, and discussing the exact reason for failure is beyond the scope of this book). The application should be agile enough to handle these failures and mitigate them. It should also be able to resume work from where it left off once it detects a node as online again. This gives a different perspective of failures.
Introducing Reactive Programming

Reactive applications should always treat a failure as a first-class event, which means that we expect failures to occur—"failures" in the sense that in a cluster of nodes, when one node fails, the another node has to continue its work.

The respect failure deserves
Failure is often scary in any application, but often we try to avert failure. When an application fails, it either stops working (idle) or exits. In a reactive system, failure is considered a first-class event in the life cycle of an application.

Scalable
The previous section discussed how responsiveness plays a role in achieving a positive experience for users. A scalable system is another important trait needed to ensure responsiveness under various load conditions.

For any e-commerce site, such as RetKart, the largest spikes in traffic are when they open a discount sale so that they can increase their sales. When RetKart sees that they are expecting a surge in traffic, they can provision and add more nodes. These extra nodes should be able to handle the traffic and the increase in throughput must be linear.

Storage scaling
During peak traffic, RetKart needs to process huge amounts of data, such as user information, product portfolios, product recommendations… the list goes on and on. However, one thing that stands out from an architectural point of view is the rate of growth of data.

The reason we are narrowing down to a NoSQL database installed on a cluster of nodes is that it scales better than a traditional RDBMS. The following diagram shows a typical configuration of storage nodes:
RetKart’s storage cluster

A very common and easy way to start scaling out is by adding more nodes to the cluster. Adding new storage nodes can also be done on the fly without affecting the underlying infrastructure, as shown the next diagram. Modern NoSQL databases such as Cassandra and HBase often come with distributed coordination, and thus it alleviates the burden of monitoring the nodes.

Scaling RetKart’s storage cluster

**Compute scaling**

Storage scaling does not really give tangible results for an application unless we scale our computation too. To put things straight, imagine a Linux box running on a dual-core processor with 4 GB RAM. It is connected to a 500 GB HDD and is using it to run a MapReduce job. There is a resource mismatch here. Running a MapReduce job generally requires a good processing speed. This example also applies to the RetKart system for scaling; that is, unless the computation is also scaled, the scaling storage provides little or no improvement in performance.
Scaling of the computing capability is not easy to build. From an architectural perspective, the system should be able to handle peak loads, must be responsive, and must be flexible to scale. The crux of reactive computing is that as a school of thought, it teaches us to model applications that are modular and react to an event.

Let's say that RetKart is getting a large number of orders during its peak traffic. Let's assume that under normal conditions, each node can handle 100,000 requests/sec at any given instant and they run their business with three nodes. This amounts to 20,000 requests per sec per node.

Assuming that RetKart announces a discount sale on a Christmas Eve, the number of orders is obviously going to increase. RetKart expects about 1,000,000 requests/sec. Clearly, there is a 10x jump in the number of requests/sec. If they go ahead with their existing infrastructure, they might be able to serve only 50 percent of the request, that is, only half of the requests.

The response times for users will be slower, and hence it will drive away the customers. To compensate for the difference, RetKart provisions additional five nodes. Now, a total of 10 nodes are sufficient to handle 1,000,000 requests/sec.

**Message-driven**

A message-driven architecture is the foundation of reactive applications. A message-driven application may be event-driven, actor-based, or a combination of the two. An event-driven system is based on events that are monitored by zero or more observers. Events are not directed to a specific address but rather subscribed to.
Actor-based concurrency is an extension of the message-passing architecture, where messages are directed to a recipient. This recipient happens to be an actor. Messages may cross thread boundaries or be passed to another actor's mailbox on a different physical server. This enables elastic scaling out on demand as actors can be distributed across the network, yet communicate with each other. The main difference between messages and events is that messages are directed while events happen.

Actor-based applications revolve around asynchronous messages passing between multiple actors. Each actor has a mailbox for receiving messages and can handle different types of messages. An actor instance has an isolated state rather than a shared state meant for storing context between messages. Actors can pass messages back and forth, or even pass messages to themselves; an actor can, for example, pass a message to itself in order to finish processing a long-running request after it has serviced other messages in its queue first. A huge benefit of actor-based concurrency is that scaling out computation across network boundaries is even easier. This is because messages are directed to actors (via remote actors). This is a powerful concept that makes it easy to build scalable applications that are also easy to design, build, and maintain. The developers only need to think about how messages flow between actors.
Another major benefit of an actor-based architecture is the loose coupling of components. The caller doesn’t block a thread that is waiting for a response. Therefore, the caller can quickly move on to some other work. The actor can send a message to another actor or reply back to the sender. This opens up many possibilities, such as distributing routines across a cluster of machines, because the call stack doesn’t couple applications with a single space in the memory, and the actor model makes the deployment topology.

Let’s correlate how a message-driven architecture can be implemented in the RetKart example. As we have already discussed, such a large site is divided into modules for scalability and fault tolerance. For the sake of simplicity, let’s assume that each request that enters the RetKart site is handled by an actor. Therefore, many actors are created on the fly whenever a new request comes in. On normal business days, the peak traffic is 100,000 requests/sec/node. Assuming that each actor handles one request, each node will have to handle 100,000 requests/sec. Actors are lightweight, and therefore we can use them to our advantage. Actors can also scale out when more nodes are added to handle more requests, such as on a Christmas Eve. If more nodes are added, then actors can be scaled out for these extra nodes. All of the interaction occurs as asynchronous messages. This is explained in the aforementioned diagram. This example does not cover in entirety how a site works but serves as a vehicle for an understanding of the tenets of reactive programming without delving into the details.

Thus, we have seen so far how the goal of the RetKart application to provide a positive user experience can be achieved with each tenet of reactive programming. The following section extends this context by providing some background of other practices that complement reactive programming.

**Practices complementing reactive programming**

In this section, we are going to touch upon some of the concepts that complement reactive programming. Though this is a side topic, it would be nice to know about it before we go to further sections.
Functional programming

Functional programming has been around since the 1970s with the introduction of ML, developed at the University of Edinburgh. Later on, other languages came in, such as Haskell, which was hugely popular within academia. But these days, with languages, functional programming is entering the mainstream. With languages such as Scala and Clojure, which runs on JVM, many Java applications can reside side by side with other JVM-compatible languages. The Erlang and Lisp family of languages, like Clojure, are other highly popular functional programming languages.

Functional programming is defined as programming with pure functions in which functions are first-class citizens. A pure function is a function that has no side effects, such as writing to a screen, reading from a file, mutating the value of a variable, and so on. The renewed interest in functional programming is due to increased awareness of no shared state, immutability with no side effects, and therefore lockless algorithms, which give rise to concurrency. This concurrency can be exploited by multicore processors.

Functional languages such as Haskell and Scala allow expressions to be evaluated in a lazy manner. This means that expressions are not evaluated unless they are referenced for the first time. Typically, most languages execute expressions eagerly. This means that expressions are evaluated during the start of a program and not necessarily referenced the first time.

Asynchronous programming

By way of illustration, we will imagine a program that consists of four conceptually distinct tasks that must be performed to complete a program. Note that we are using the word "task" in the non-technical sense of something that needs to be done. The first model that we will look at is the single-threaded synchronous model, as shown in this diagram:
Introducing Reactive Programming

This is the simplest style of programming. Each task is performed one at a time, with one finishing completely before another is started. If the tasks are always performed in a definite order, the implementation of a later task can assume that all earlier tasks have finished without errors, with all their outputs available.

In contrast, the multi-threaded synchronous model illustrated in the next diagram performs each task in a separate thread of control. The threads are managed by the operating system and run concurrently.

The tasks may be interleaved together on a single processor. The point is that in the multi-threaded model, the details of the execution are handled by the OS, and the programmer simply thinks in terms of independent instruction streams, which may run simultaneously. Although this diagram is simple, in practice, multi-threaded programs can be quite complex because of the need for threads to coordinate with one another. Thread communication and coordination is more complicated and can be difficult to get right. Some programs implement parallelism using multiple processes instead of multiple threads. Although the programming details are different, conceptually, it is the same model as what you just saw.

Data streams

An uninterrupted flow of data sequence can be thought of as a data stream. On a very small scale, data streams can be anything from reading files from a disk to serving content such as media on YouTube to millions of viewers across the Internet. Data streams pose a very different challenge, and it is not normally seen during batch processing.

Typically, there is a source from which data is produced and a sink to which data is delivered. Between the source and the sink, data transformation takes place. The following diagram explains a typical flow of data in a pipelined fashion.
Reactive streams are an upcoming standard for asynchronous stream processing with back pressure. Currently, the reactive streams standard targets only JVM.

Data is stored in the sink, such as a database, after a series of transformations. Transformations may validate the data or enrich it, pulling more data from additional data stores. Back pressure is applied in the opposite direction of the data flow to protect the system from halting.

**Micro-services**

Micro-services is a modern architectural approach that allows components to be modularized and exposed as services. This is also fine with components being asynchronous. Hence, they can also be composed together. A website, for example, can be composed of many small micro-services. Different components that serve only a small subset of the overall service can be put together to form one big website. Micro-services are also easy to deploy since each micro service is modular and any change made to that service does not affect others.

Micro-services make use of lightweight containers whose services can be deployed independently and can be updated as and when needed. This enables you to organize the development effort around multiple teams. If any service has a memory leak, then only that service will be affected. The other services will continue to handle requests.
Introducing Reactive Programming

In comparison, one misbehaving component of a monolithic architecture can bring down the entire system. However, with a micro-service-driven architecture, the components can be debugged in isolation and fixed.

A typical micro-services organization

A popular approach to partitioning the system is by using nouns or resources. This kind of service is responsible for all operations that work on entities/resources of a given type. For example, the RetKart site explained in the previous section could have an inventory service that keeps track of whether some products are in stock. Other micro-services, such as a shipping service and an accounting service, come together to make one big website, which we call a storefront. The web frontend communicates with the micro-services.

RESTful services
Every major development language now includes frameworks for building RESTful web services. As such, it is important for web developers and architects to have a clear understanding of REST and RESTful services.

Representational State Transfer (REST), which is an architectural style for networked hypermedia applications, is primarily used to build web services that are lightweight, maintainable, and scalable. A service based on REST is called a RESTful service. REST is not dependent on any protocol, but almost every RESTful service uses HTTP as its underlying protocol.
Features of a RESTful service

Every system uses resources. These resources can be pictures, video files, web pages, business information, or anything that can be represented in a computer-based system. The purpose of a service is to provide a window for its clients so that they can access these resources. Service architects and developers want this service to be easy to implement, maintainable, extensible, and scalable. In general, RESTful services should have the following properties and features.

Representations

The focus of a RESTful service is on resources and how to provide access to these resources. A resource can easily be thought of as an object as in OOP. A resource can consist of other resources. RESTful services are designed by identifying the resources and determining how they are related to each other.

Messages

The client and service talk to each other via messages. Clients send a request to the server and the server replies with a response. Apart from the actual data, these messages also contain some metadata about the message.

URIs

RESTful systems should have a uniform interface. HTTP 1.1 provides a set of methods, called verbs, for this purpose.

Statelessness

A RESTful service is stateless and does not maintain the application state for any client. A request cannot be dependent on a past request and a service treats each requests independently. HTTP is a stateless protocol by design, and you need to do something extra to implement a stateful service using HTTP links between resources.

Caching

Caching is the concept of storing generated results and using the stored results instead of generating them repeatedly if the same request arrives in the near future. This can be done on the client, the server, or any other component between them, such as a proxy server.

Some known REST APIs in Scala are as follows:

- Scalatra
- Finatra
- The Play framework
Some technical use cases of reactive applications
In this section, we are going to explore some use cases where reactive programming can be put into practice.

Internet of Things
Internet of Things (IoT) has been a hot topic these days. Simply put, the idea is connecting any device to the Internet or a network. This includes everything from cell phones, coffee makers, and washing machines to headphones, lamps, wearable devices, and almost anything else that you can think of. IoT is so trendy these days that the analyst firm Gartner says that by 2020, there will be over 26 billion connected devices! That’s a lot of connections (some even estimate this number to be much higher—over 100 billion).

There are many day-to-day examples that can change our lives with IoT. Let’s imagine the following things for a moment:

- Imagine you are on your way to a meeting, your car could have access to your calendar and already know the best route to take. If the traffic is severe, your car might send a text to the other party notifying them that you will be late.
- Your alarm clock wakes up you at 6:00 a.m. and then notifies your coffee maker to start brewing coffee for you.
- The office printer knows when it is running low on supplies and automatically reorders more.
- Your phone is connected to your account and a payment is made at a point-of-sale terminal by scanning a QR code.

The data collected from these various use cases could be analyzed and valuable metrics could be determined, which is where reactive programming plays a vital role. Reactive streams play a major role in collecting data from the devices. The data thus obtained is processed on the server side for analytics.

Reactive electricity measure
Next up is a schematic diagram of reactive electricity measure. Here, a sensor is attached to the electric meter, which monitors the reading. The sensor measures the volts passed and converts the analog data into digital bytes.
The sensor is connected to a minicomputer (a single-board computer, such as Raspberry Pi or Arduino) that receives the bytes. To make things much more complex, more sensors are connected to the minicomputer so that it forms a grid of sensors.

![Diagram of reactive electric measure](image)

A schematic diagram of reactive electric measure

The minicomputer sends this sensor data as a stream to the Analytics server that is deployed on the cloud. The analysis server can be written as a micro-service using Spray (an open-source toolkit for building REST services on top of HTTP) exposes its services via REST. In this case, it is obvious that the minicomputer should be connected to the Internet. The data obtained from each of the sensors can be analyzed for various measures, including:

- The number of units consumed per month
- The total hours of power outages per month
- The peak load and low load on a daily basis

The data points obtained from these observations can be plotted on a nice graph and streamed out in a web page.

**Tumblr's architecture**

Tumblr is a blogging platform and is used by millions of people across the globe. Tumblr's architecture is especially appealing because not only is the content of Tumblr appealing, but also lots of people share it concurrently. Thereby, this architecture has a lot of scalability challenges. According to Tumblr, they use Scala for development because they feel it scales well for their use case, and since they wanted to use JVM, Scala was the preferred choice over Java. They also use Finagle, created by Twitter; this is a protocol-agnostic, asynchronous RPC system.
Introducing Reactive Programming

Tumblr gets more than 50 million posts per day, and each post needs to be distributed to hundreds more. To handle such a scale, the hardware should be equally capable. Tumblr has 500 million page views per day, with more than 15 billion page views per month. At a given instant, they get more than 40,000 concurrent requests per second. To achieve this scale, Tumblr uses hardware that comprises 500 web servers, 200 database servers, 30 memcached servers, 22 Redis servers, and so on.

**Tumblr's internal firehose**

Tumblr has an internal firehose for distributing a real-time activity stream for various applications within the data pipeline. The activity stream might include millions of users creating or deleting posts, liking or unliking, reblogging, and so on. Such vast amounts of data need to be distributed without choking up the system.

Thus, Kafka, which acts as a central repository for distributing activity streams, is used to achieve this high throughput. Kafka is an open source Apache project originally created by LinkedIn. It is a publish-subscribe messaging system wherein a group of clients may subscribe to a set of producers. A group of clients may belong to the same consumer group, and only those clients who are part of the same consumer group will receive the messages. Since Kafka maintains a write-ahead log, the messages are written to an append-only log file. Hence, the clients don't see duplicate data. Furthermore, multiple clients belonging to the same consumer group can process independently of each other, thus exploiting parallelism.

Tumblr's firehose model is built around Kafka. It is easy to rewind the stream up to a certain period. Therefore, any client that needs to read data can also specify from which point in time it needs to be fed with activity streams. All clients consume messages using HTTP streams.

HBase is just used to shorten billions of URLs and for historical data. It is used where a million writes per second are required.

**Tools of the trade**

This book is all about Scala, and hence getting to know the tools and development environment would be of great help before we get started with the next chapter. This book is conscious about presenting the concepts with suitable case study and code examples. The code presented here is shared on GitHub so as to help you play with the examples presented in the book. Detailing all the concepts of Scala and Akka is beyond the scope of this book. However, the intent is to briefly cover the concepts necessary to get acquainted.
It is a good idea to download all of these tools and get ready with a workable environment before you jump into the rest of the chapters. The following sections explain the tools and how to use them to create a skeleton project to get you started.

**The Scala programming language**

Scala has full support for functional programming as well as a very strong static type system. This allows programs written in Scala to be very concise and thus smaller in size than other general-purpose programming languages. Also, Scala encourages both object-oriented and functional ways of writing code. Hence, a programmer can take advantage of both of these paradigms. The first part of the book covers the concepts built around the core Scala language. We also assume that you have written some beginner-level Scala code so you can grasp the code. If not, there are a lot of resources to learn Scala from. Here, we have listed a few:

- **Scala documentation**: This is the official tutorial from typesafe (http://docs.scala-lang.org/tutorials/)
- **Scala school by Twitter**: This is from Twitter. They have compiled a nice HTML page that describes the fundamental concepts at https://twitter.github.io/scala_school/

We are going to use Scala version 2.11.x5, which is the latest stable version at the time of writing this book.

**Akka**

Akka is an actor-based toolkit and runtime part of the typesafe reactive platform. It is for building highly concurrent, distributed, and fault-tolerant actor-based applications on the JVM. Akka has a number of other incredible features for building reactive applications, such as supervisor hierarchies for resilience and distributed workers for scalability. We will take a deep dive into Akka in the second part of this book. As with Scala, we will assume that you have written some Akka code. If not, to get a fundamental understanding, refer to the Akka official documentation.

The Akka official documentation is at http://doc akka.io/docs/akka/2.3.9/scala.html.

We are going to use Akka version 2.3.9, which is the latest stable version at the time of writing this book.
Introducing Reactive Programming

SBT

Simple Build Tool (SBT) is the official build tool for the Scala language. It is written in Scala and can be used to compile both Java and Scala code. SBT is not just a build tool; it also provides the basic framework for a development environment. Its integrated Scala REPL, the continuous compile mechanism, and the fast compile server make it a necessary tool in any Scala developer's toolkit. SBT, like Maven, provides a command prompt to build a project. Throughout this book, for all the code examples and illustrative case studies, we will use SBT 0.13.5 to define the build and manage dependencies.

In the next section, we will create a skeleton Scala project by defining a simple build.sbt file and briefly the various sections of the build file.

A skeleton SBT project

Let's take a brief look at how we normally define a build script for a Scala project:

```
/project-root
|_src
 |_main
 |  |_scala
 |  |_java
 |  |_resources
 |_test
 |  |_scala
 |  |_java
 |  |_resources
 |_project
 |_plugins.sbt
 |_build.properties
 |_Build.scala
 |_target
 build.sbt
```

This is a director hierarchy of a simple Scala project. This structure looks very similar to a Maven project, except for `build.sbt` and `plugins.sbt` under `project`:

- **project_root**: The directory under which the entire project is placed.
- **build.sbt**: This defines the project name, organization, and dependencies. To write a build, the script is written in a DSL-style notation.
• plugins.sbt: This defines the plugin dependencies required to build the project and plugins generally required to support other project types, such as web project, packaging, bundling as a single executable JAR file, and so on.

• build.properties: This defines the SBT version used to build the project.

• Build.scala: This is an alternate way of defining a build and exactly follows the same notations to define the build. Unlike build.sbt where a DSL is used, here the build is defined entirely in the Scala language. This is very useful for defining large builds wherein a project has multiple child projects and deals with a lot of dependencies. (Let’s park Build.scala aside for this chapter. We will resume looking at it in the next part of the book.)

The following are the steps for building a skeleton SBT project:

1. Writing build.sbt:
   - Let’s define build.sbt for a project. Under project_root, create a file named build.sbt with the following contents:
     ```scala
     name       := "rp-chapter1"
     organization   := "pack.rpis.chapter1"
     scalaVersion   := "2.11.5"
     version    := "0.1"
     libraryDependencies ++= Seq("org.scalatest" %%% "scalatest" % "2.0" % "test")
     ```
   - For the skeleton, we are not using any plugin dependencies (hence let’s leave them aside).
   - build.sbt shown here serves for a basic Scala project with just one test dependency.

2. Writing build.properties:
   - Let’s define a build.properties for a project by typing the following code:
     ```scala
     sbt.version = 0.13.5
     ```

3. Execute the build in SBT prompt:
   - Let’s assume that sbt is installed and Command Prompt is right in your project_root directory. Execute the following code:
     ```bash
     /project_root $ sbt
     ```
Introducing Reactive Programming

This should create the src and target folders and download the ScalaTest dependency. With this, we have successfully created a skeleton project. This will serve as a foundation for other illustrative case studies in the following chapters.

ScalaTest – a unit testing framework

No code is complete without unit testing. In the Scala world, we use the ScalaTest framework. The advantage of ScalaTest is that it embraces both TDD and BDD (which is beyond our scope). In this book, we will use TDD in the forthcoming chapters.

ScalaTest can easily be added to the Scala project with the following dependencies specified in the build.sbt file:

```scala
libraryDependencies ++= Seq("org.scalatest" %% "scalatest" % "2.2.4" % "test")
```

ScalaTest offers several different ways of writing tests. The simplest is the FunSpec parameter. It contains a standard storyboard that describes the reason for the existence of the test using a describe clause and subsequent tests that fulfill that description. The following code is a simple ScalaTest code that tests a case class:

```scala
import org.scalatest.{ShouldMatchers, FunSpec}

class AddressBookTest extends FunSpec with ShouldMatchers {
  describe("An AddressBook") {
    it("can add a Contact object toolkit the AddressBook") {
      val mjContact = new Contact("Michael", "Jackson", "mjackson@mj.me", "")
      val contacts = List(mjContact)
      val addressBook = new AddressBook(contacts)
      addressBook.contacts.head.fname shouldBe "Michael"
    }
  }
}

case class AddressBook(contacts: List[Contact])

case class Contact(fname: String, lname: String, email: String, address: String)
```
A word about IDEs
All major IDEs (Eclipse, Netbeans, and IDEA) support Scala via plugins, the official one being Scala IDE from TypeSafe (Eclipse-based). Though there are a lot of options, we leave it up to the reader to choose the one that suits them better, the reason being that the build process is the same irrespective of the IDE used. In the early days, stability of IDEs was a huge concern, but now the IDE communities have grown and support for the language has also greatly improved. Hence, for a quick start, having an IDE is always very beneficial.

Summary
We have reached the end of the first chapter. We discussed the driving principles of reactive programming with an example and the related methods for building a reactive application. This chapter highlighted the concepts required in order to understand more concepts that will be discussed in the forthcoming chapters.
Where to buy this book

You can buy Reactive Programming with Scala and Akka from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.