Covers Play, Akka, and Reactive Streams

# Reactive
## Web Applications

Manuel Bernhardt

FOREWORD BY James Roper

SAMPLE CHAPTER

MANNING

# *Reactive Web Applications*

by Manuel Bernhardt

**Chapter 1**

*brief contents*

# Part 1

# Getting started with reactive web applications

This part of the book will get you started with reactive web applications by providing you with the foundation you need to understand the concepts discussed later in the book. You'll learn how reactive web applications came to be and why they matter, and then you'll get your hands dirty by building a simple reactive web application. You'll also get a quick introduction to the concepts behind functional programming as well as to the Play Framework, should you not be familiar with those topics already.

# 1 Did you say reactive?

***This chapter covers***

- Reactive applications and their origin
- Why reactive applications are necessary
- How Play helps you build reactive applications

Over the past few years, web applications have started to take an increasingly important role in our lives. Be it large applications such as social networks, medium-sized ones such as e-banking sites, or smaller ones such as online accounting systems or project management tools for small businesses, our dependency on these services is clearly growing. This trend is now transitioning to physical devices, and the information technology research and advisory firm Gartner predicts that the Internet of Things will grow to an installed base of 26 billion units by 2020.[1]

Reactive web applications are an answer to the new requirements of high availability and resource efficiency brought by this rapid evolution. Cloud computing and the subsequent emergence of cloud services have shifted web application development from an activity wherein one application tries to solve all kinds of

[1] Gartner, "Gartner Says the Internet of Things Installed Base Will Grow to 26 Billion Units By 2020" (December 12, 2013), www.gartner.com/newsroom/id/2636073.

problems to a process of identifying and connecting to adequate cloud services and only solving those problems that have not been solved beforehand satisfactorily.

We need a new set of tools to help us efficiently deal with the challenges that come with this evolution. The Play Framework has been designed from the ground up to make it possible to build reactive web applications that are capable of providing real-time behavior to users even under high load and in a decentralized setting. At the time of this writing, Play is the only full-stack reactive web application framework available on the Java virtual machine. Embraced by large companies such as Morgan Stanley, LinkedIn, and The Guardian, as well as many smaller players, Play is available as free, open source software, ready to be downloaded to your computer.

In this chapter, we'll look into what reactive web applications are, why you'd want to build such applications, and why the Play Framework is a good tool for this purpose. We'll start by disambiguating the meaning of the word "reactive" and look into how new trends in hardware design and software architecture call for a reconsideration of how to use computational resources. Finally, we'll explore why failure handling plays a crucial role in this context, and how it can be achieved.

## 1.1 Putting reactive into context

If you're reading this book, chances are that you've heard of concepts such as reactive applications, reactive programming, reactive streams, or the Reactive Manifesto. Even though we can probably agree that all those terms sound a lot more exciting when prepended with *reactive*, you may wonder what *reactive* means in those different contexts. Let's find out by looking at the origins of the word in relation to computer systems.

### 1.1.1 Origins of reactive

The concept of reactive systems isn't new. In their paper "On the Development of Reactive Systems"[2] (published in 1985), David Harel and Amir Pnueli round up several dichotomies to characterize complex computer systems and propose a novel dichotomy: *transformative* versus *reactive* systems. Transformative systems accept a known set of inputs, transform those inputs, and produce outputs. For example, a transformative system may prompt the user for some input, and then for some more, depending on what the user provided, to finally provide a result. Think, for example, of a pocket calculator, which accepts numbers and performs basic operations to finally return a result when the equals key is pressed. Reactive systems, on the other hand, are continuously stimulated by the external environment, and their role is to continuously respond to these stimuli. For example, a wifi-enabled camera with motion-detection capabilities may notice a burglar enter a room and send an alert to the camera owner's mobile phone, letting them witness helplessly their room being emptied of its precious belongings, as well as later on, the police arriving on the scene.

[2] A PDF version of the article is available at http://mng.bz/p1n3.

A few years later, Gérard Berry refined this definition by introducing the distinction between *interactive* and reactive programs. Whereas interactive programs set the speed at which they interact with the environment themselves, reactive programs are capable of interacting with the environment at the speed dictated by the environment.[3]

Thus, reactive programs

- Are available to continuously interact with their environment
- Run at a speed that is dictated by the environment, not the program itself
- Work in response to external demand

Coming back to present times, the preceding modus operandi of reactive programs looks a lot like how web applications operate or should be operating. Though appealing in theory, it takes quite some effort to fulfill these criteria, and possibly serious hardware resources, depending on the number of users and the nature of what they demand. It's perhaps the lack of widespread high-performance hardware capable of delivering real-time interaction at scale that explains why we haven't heard much of reactive systems until recently, when a set of core aspects that characterize reactive systems were published under the name *Reactive Manifesto*.

### 1.1.2 The Reactive Manifesto

The first version of the Reactive Manifesto was published in June 2013, and it describes a software architecture with the name *Reactive Applications*. Reactive applications are defined by a set of characteristics, or *traits* as they're called in the manifesto (those traits have nothing to do with Scala's `traits`), that altogether make up for applications that behave in the same way as the reactive programs we talked about earlier: continuously available and readily responding to external demand. Although the Reactive Manifesto may seem like it's describing an entirely new architectural pattern, its core principles have long been known in industries that require real-time behavior from their IT systems, such as financial trading.

The following four traits make up reactive applications:

- *Responsive*—React to users
- *Scalable*—React to load
- *Resilient*—React to failure
- *Event-driven*—React to events

A *responsive* application will satisfy the user's expectations in terms of availability and real-time behavior. Real-time, or near real-time, means that the application will respond within a short or very short time. The time interval between the request and response is called *latency*, and it's one of the key measurements when it comes to assessing how well a system performs.

[3] Gérard Berry, "Real-Time Programming: General Purpose or Special-Purpose Languages," *Information Processing* 89 (Elsevier Science Publishers, 1989): 11-18.

In order to continuously interact with their environment, reactive applications must be able to adjust to the load they're facing. Sudden traffic bursts may affect an application; for example, a popular tweet with a link to a news article could cause a rush on a news website. To this end, an application must be *scalable*: it must be able to make use of increased computational capacity when necessary. This means it must be able to make efficient use of the hardware on a single machine (which may have one or more CPU cores), and also be able to function across several computation nodes at its disposal, depending on the load.

**NOTE** We use the term "computation node" or simply "node" to refer to a resource on which a web application runs. In practice, this may be a physical computer, a virtual machine, or even a logical node on a Platform-as-a-Service provider.

Because even the simplest of software systems are prone to failure (whether software-related or hardware-related), reactive applications need to be *resilient to failure* to meet the demand of continuous availability. The capability of an application to get back on its feet should it encounter a problem is arguably even more important when it comes to scalable systems, which are more complex in nature and distributed, because the likeliness of hardware or network failure is increased.

*Event-driven* applications based on *asynchronous* communication can help you achieve the previously listed traits. In this setup, the system (or subsystem) reacts to discrete events such as HTTP requests without monopolizing computational resources as it waits for an event to occur. This natural level of concurrency yields better latency than traditional synchronous method calls. Another consequence of writing event-driven programs is that components are loosely coupled, making the software much more maintainable in the longer term.

## 1.1.3 Reactive programming

Reactive programming is a programming paradigm based on data flows and the propagation of changes. Consider, for example, the spreadsheet represented in table 1.1.

The cell `C1` is defined programmatically in the following way:

```
= A1 * B1
```

**Table 1.1 A simple spreadsheet demonstrating the concept of reactive programming**

|   | A | B | C |
|---|---|---|---|
| 1 | 6 | 7 | 42 |
| 2 |   |   |   |
| 3 |   |   |   |

If we were to run the preceding example in spreadsheet software, as soon as either the value of `A1` or `B1` was changed, the result in `C1` would change accordingly. The programming language behind the spreadsheet thus allows us to define relations between the data that result in the propagation of changes across the spreadsheet.

In order to implement a real-time spreadsheet application, such as the one in Google Drive, we'd build on top of lower-level concepts such as events: when the user changes the value of cell `A1`, an event is fired. All the cells interested in the content of `A1`, such as cell `C1` containing our expression, would act on this event by reevaluating themselves and displaying a new value. This process is entirely hidden from the user, who is only concerned with describing the high-level relation among cell values.

In terms of web application development, this technique is increasingly being used for front-end application development: tools such as KnockoutJS, AngularJS, Meteor, and React.js all make use of this paradigm. The developers only need to describe how changes in the data propagate through the user interface; they don't need to concern themselves with the nitty-gritty details of declaring listeners on specific DOM elements, thus greatly simplifying how reactive user interfaces can be implemented. We'll look into reactive user interfaces in chapter 8.

Similar abstractions, wherein events play a central role, can also be found on the server side. A new initiative called *Reactive Streams*, which we'll talk more about in chapter 9, aims at providing a standard interface for working with asynchronous stream processing on the JVM.

### 1.1.4 The emergence of reactive technologies

Over the years, a number of technologies and frameworks have been developed that share common aspects and can be broadly classified as reactive technologies. Building reactive applications takes more than simply using reactive technologies, as you'll see later, but technologies must satisfy a number of prerequisites to enable reactive behavior, most notably the capacity for *asynchronous* and *event-driven* code execution.

Microsoft's Reactive Extensions (Rx; https://rx.codeplex.com/) is a library for composing asynchronous and event-based programs, available on the .NET platform and other platforms such as JavaScript. Node.js (http://nodejs.org) is a popular platform for building asynchronous, event-driven applications in JavaScript. On the JVM, a number of libraries enable these capabilities, such as Apache MINA (https://mina.apache.org) and Netty (http://netty.io).

Those low-level technologies all offer basic tools for building asynchronous and event-driven applications, but it takes a bit more work to get to the state of a full-blown web application that also has to deal with concerns such as code organization, view templates, inclusion and organization of client-side resources such as stylesheets and JavaScript files, database connectivity, security, and so on. Many so-called full-stack web application frameworks exist, but few of them also include reactive technologies, and very few are built from the ground up using reactive technologies, embracing reactive principles at their core. Full-stack frameworks concern themselves with all the layers required to build and deploy an application: client-side UI technology (or a means to integrate it), server-side business logic, authentication, integration of database access, and various libraries for the most common tasks (such as remote web service calls). In a reactive application, all these layers must furthermore cooperate by following the same principles of asynchronous communication and error recovery.

On the JVM, the only mature full-stack reactive web application framework to this day is the Play Framework. Other full-stack frameworks such as Lift (http://liftweb.net) provide a good alternative for building web applications, but they haven't been designed with asynchronicity, failure resilience, and scalability as primary goals.

Play is built on top of Netty and leverages its reactive behavior by using asynchronous stream handling provided by *Reactive Streams* (see figure 1.1).

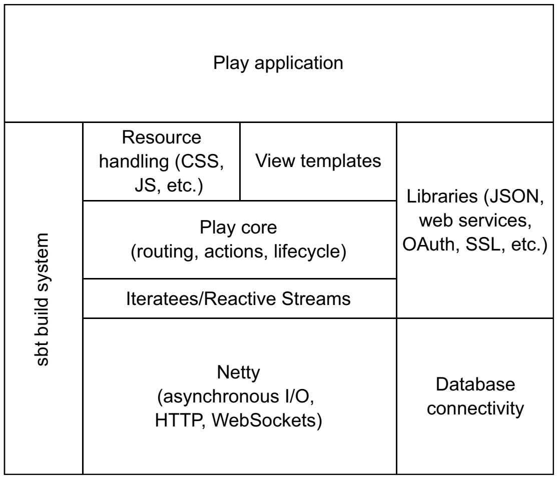

Play application
sbt build system
Resource handling (CSS, JS, etc.)
View templates
Libraries (JSON, web services, OAuth, SSL, etc.)
Play core (routing, actions, lifecycle)
Iteratees/Reactive Streams
Netty (asynchronous I/O, HTTP, WebSockets)
Database connectivity

**Figure 1.1 High-level architecture of the Play Framework**

Play deals with the typical concerns of web application development such as client-side resource handling, project compilation, and packaging by making use of the sbt build tool. It comes with a number of useful libraries to address common concerns such as JSON handling and web service access and offers access to databases though a range of plugins. Throughout the rest of this book, you'll learn how to use the Play Framework as an effective tool to build reactive web applications.

Let's now take a closer look at how web applications work and how they make use of computational resources to understand why the asynchronous, event-driven behavior of reactive web applications is necessary.

## 1.2 Rethinking computational resource utilization

To understand the why and how of reactive applications, we need to take a quick look at computers. They have certainly evolved a lot over the past decades, especially in terms of CPU clock speed (MHz to GHz) and memory (kilobytes to gigabytes). The most significant change, however, which has happened in the past few years, is that although the clock speed of CPUs isn't increasing very much, the number of cores each CPU has is changing. At the time of writing, most computers have at least 4 CPU cores, and there are already vendors offering CPUs with 1024 cores. On the other hand, the overall architecture of computers and the mechanism by which programs are executed haven't undergone a significant evolution, so some of the limitations of this architecture, such as the *von Neumann bottleneck*,[4] become more of a problem nowadays. To understand how this evolution affects web application development, let's take a look at the two most popular web server architectures.

[4] John Backus, "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs," *Communications of the ACM* 21 (8) (August 1978): 613-41.

### 1.2.1 Threaded versus evented web application servers

Roughly speaking, there are two categories of programming models in which web servers can be placed. In the *threaded* model, large numbers of threads take care of handling the incoming requests. In an *evented* model, a small number of request-processing threads communicate with each other through message passing. Reactive web application servers adopt the evented model.

#### THREADED SERVERS

A threaded server, such as Apache Tomcat, can be imagined as a train station with multiple platforms.[5] The station chief (acceptor thread) decides which trains (HTTP requests) go on which platform (request processing threads). There can be as many trains at the same time as there are platforms. Figure 1.2 illustrates how a threaded web server processes HTTP requests.

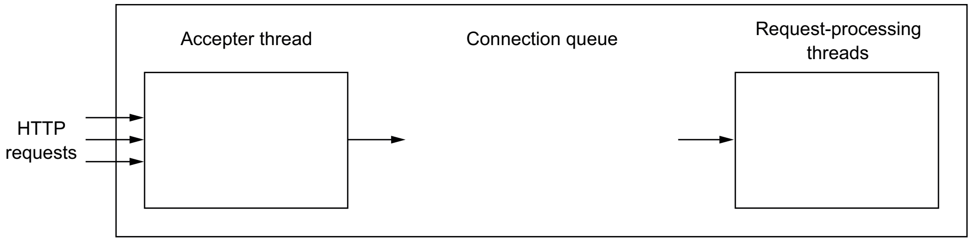


**Figure 1.2 Threaded web server**

As implied by the name, threaded web servers rely on using many threads as well as on queuing. The analogy between trains and threaded web application servers is depicted in table 1.2.

**Table 1.2 Imagining threaded web application servers as train stations**

| Train station | Threaded server |
|---|---|
| More trains come in than there are platforms; trains have to queue up and wait. | More HTTP requests reach the server than there are worker threads; users connecting to the application have to wait. |
| Trains hanging around at the platform for too long may be cancelled. | HTTP requests taking too long to process are cancelled; the user may see a page with *HTTP Error 408 - Request timeout.* |
| Too many trains queuing up in the station can cause huge delays and passengers to go home. | Too many requests queuing up can cause users to leave the site. |

[5] See Julian Doherty, "How Your Web Server Works," http://madlep.com/How-your-web-server-works-/.

### EVENTED SERVERS

To explain how evented servers work, let's take the example of a waiter in a restaurant.

A waiter can take orders from several customers and pass them on to multiple chefs in the kitchen. The waiter will divide their time between the different tasks at hand and not spend too much time on a single task. They don't need to deal with the whole order at once: first come the drinks, then the entrees, later the main course, and finally dessert and an espresso. As a result, a waiter can effectively and efficiently serve many tables at once.

As I write this book, Play is built on top of Netty. When building an application with Play, developers implement the behavior of the chefs that cook up the response, rather than the behavior of the waiters, which is already provided by Play.

The mechanism of an evented web server is shown in figure 1.3.

In an evented web server, incoming requests are sliced and diced into events that represent the various smaller pieces of work involved in handling the whole request, such as parsing the request body, retrieving a file from disk, or making a call to another web service. The slicing and dicing is done by *event handlers*, which may trigger I/O actions, resulting in new events later on. Say, for example, that you wanted to issue a request for the size of a file on the web server. In this case, the event handler dealing with the request will make an asynchronous call to the disk. When the operating system is done figuring out the size of the file, it emits an interrupt, which results in a new event. When it's the turn of that event to be handled, you'll get a response with the size. While the operating system is taking care of figuring out the size of the file, the event loop can process other events in the queue.

One important implication of the evented programming model is that the time spent on tasks needs to be small. If a chef insisted on cooking the whole order when a waiter simply wanted to place it, there would be many angry unserved customers once the waiter finally got back from the kitchen. The evented model only works if the entire pipeline is *asynchronous*: orders, or HTTP requests, are processed without *blocking*. The term *nonblocking I/O* is often used to refer to input-output operations that don't hold up the current execution thread while doing their work, but instead send a notification when the work is done.

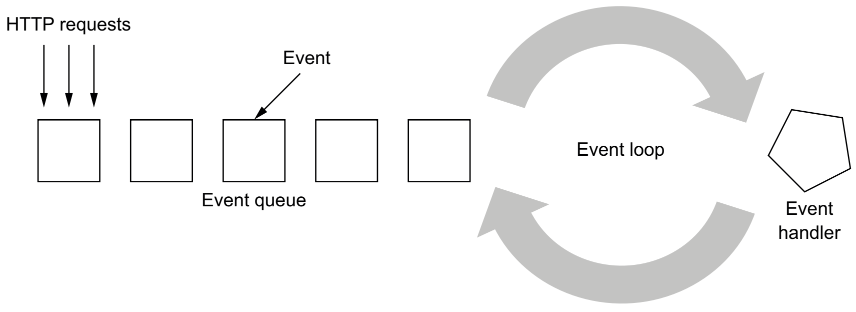


**Figure 1.3 Evented web server**

#### MEMORY UTILIZATION IN THREADED AND EVENTED WEB SERVERS

Evented web servers make much better use of hardware resources than threaded ones. Instead of having to spawn thousands or tens of thousands of "train track" worker threads to deal with large numbers of incoming requests, only a few "waiter" threads are necessary. There are two advantages to working with a smaller number of threads: reduced memory footprint and much improved performance due to reduced context switching, thread management time, and scheduling overhead.

Each thread created on the JVM has its own stack space, which is by default 1 MB. The default thread pool size of Apache Tomcat is 200, which means that Apache Tomcat needs to be assigned over 200 MB of memory in order to start. In contrast, you can run a simple Play application with 16 MB of memory. And although 200 MB may not seem like a lot of memory these days, let's not forget that this means that 200 MB are required to process 200 incoming HTTP requests at the same time, without taking into account the memory necessary to perform additional tasks involved in handling these requests. If you wanted to cater to 10,000 requests at the same time, you'd need a lot of memory, which may not always be readily available. The threaded model has difficulty scaling up to a larger number of concurrent users because of its demands on available memory.

In addition to utilizing a lot of memory, the threaded approach results in inefficient use of the CPU.

### 1.2.2 *Developing web applications fit for multicore architectures*

Threaded web servers rely on multiple thread pools to distribute the available CPU resources among incoming requests. This mechanism is mostly hidden from developers, letting developers work as though there were only one main thread. Arguably, developing against an abstraction that hides away the increased complexity of dealing with multiple threads may appear simpler at first. Indeed, programming contracts such as the Servlet API provide the illusion that there's only one main thread of execution answering an incoming HTTP request and all the resources in the world to answer it. But the reality is somewhat different, and this leaky abstraction brings its own set of drawbacks.[6]

#### SHARED MUTABLE STATE AND ASYNCHRONOUS PROGRAMMING

If you've built web applications served by a threaded server, chances are that you've found yourself facing the side effects of a race condition caused by the use of *shared mutable state*. Threads on the JVM, while running in parallel, do not run in isolation: they have access to the same memory space, open file handles, and other shared resources as other threads. One classic example of the problems caused by this behavior is a Java servlet making use of the `DateFormat` class:

```
private static final DateFormat dateFormatter = new SimpleDateFormat();
```

[6] Joel Spolsky, "The Law of Leaky Abstractions," http://www.joelonsoftware.com/articles/LeakyAbstractions.html.

The problem with the preceding line is that `DateFormat` is not thread-safe. When called by two threads concurrently, it doesn't act differently depending on what thread is calling it, and makes use of the same variables to hold its internal state. This leads to unpredictable behavior and to bugs that are usually hard to understand and analyze. Even experienced developers spend a lot of time trying to understand race conditions, deadlocks, and other strange, funny, or despairing side effects brought about by this unfortunate situation. This isn't to say that applications written in an evented way are immune to the phenomenon of shared mutable state—for the most part, application developers decide whether or not to make use of mutable data structures and what level of exposition to give them. But the design of frameworks such as Play and languages such as Scala discourages developers from making use of shared mutable state.

#### LANGUAGE DESIGN AND IMMUTABLE STATE

Languages and tools favoring the use of immutable state make it easier to develop web applications that have to deal with concurrent access. The Scala programming language is designed to use immutable values by default, rather than mutable variables. Although it's possible to write programs in an immutable fashion in Java, a lot more boilerplate is involved than in Scala. For example, declaring an immutable value in Scala is done like this:

```
val theAnswer = 42
```

The same result would be achieved in Java by explicitly prepending the `final` keyword:

```
final int theAnswer = 42
```

This may seem like a minor difference, but over the course of writing a large application, it means that the `final` keyword needs to be used many, many times. When it comes to more-complex data structures, such as lists and maps, Scala provides these data structures in both their immutable and mutable versions, favoring the immutable one by default:

```
val a = List(1, 2, 3)
```

Java, on the other hand, doesn't provide immutable data structures in its collection library. You'd have to use third-party libraries such as Google's Guava library (https://github.com/google/guava) to get a useful set of immutable data structures.

#### LOCKS AND CONTENTION

To avoid the side effects caused by concurrent access to non-thread-safe resources, locks are used to let other threads know that a resource is currently busy. If all goes well, the thread holding the lock will release it and thus inform other possibly waiting threads that they may now access the resource in turn. In some situations, however, threads may wait for one another to release a lock and be stuck in a *deadlock*. If a thread holds on to a resource for too long, this may cause resource *starvation* from the viewpoint of

**The Scala programming language**

One of the main design goals of the Scala programming language is to enable developers to tackle the complexity of programming multicore and distributed systems. It does so by favoring immutable values and data structures over mutable ones, providing functions and higher-order functions as first-class citizens of the language, as well as easing the use of an expression-oriented programming style. For this reason, this book's examples are written in Scala rather than Java. (It should, however, be noted that Play, Akka, and Reactive Streams all have Java APIs.) We'll review the core concepts of functional programming with Scala in chapter 3.

other threads. When the load on a web application that relies on locks surges, it isn't unusual to observe *lock contention*, which results in decreased performance for the whole application.

The new many-core architecture that CPU vendors have moved toward doesn't make locks look any better. If a CPU offers over 1,000 real threads of execution, but the application relies on locks to synchronize access to a few regions in memory, one can only imagine how much performance loss this mechanism will entail. There is a clear need for a programming model that better suits the multithread and multicore paradigm.

### THE APPARENT COMPLEXITY OF ASYNCHRONOUS PROGRAMMING

For a long time, writing asynchronous programs hasn't been popular among developers because it can seem more difficult than writing good old synchronous programs. Instead of the ordered sequence of operations in a synchronous program, a request-handling procedure may end up being split into several pieces when written in an asynchronous fashion.

One of the popular ways of writing asynchronous code is to make use of callbacks. Because the program's flow of execution isn't blocked when waiting for an operation to complete (such as retrieving data from a remote web service), the developer needs to implement a callback method that's executed once the data is available. Proponents of the threaded programming model would argue that when the processing is a bit more complicated, this leads to a style of code known as "callback hell."

**Listing 1.1 Example of nested callbacks in JavaScript**

**The main function composes a list of items and their prices.**

**First callback function handles the retrieval of the items**

**Second callback function is called for each item**

**Third callback method handles the retrieval of the price of one item**

```
var fetchPriceList = function() {
    $.get('/items', function(items) {
        var priceList = [];
        items.forEach(function(item, itemIndex) {
            $.get('/prices', { itemId: item.id }, function(price) {
                priceList.push({ item: item, price: price });
                if ( priceList.length == items.length ) {
                        return priceList;
                    }
                }).fail(function() {
                    priceList.push({ item: item });
                    if ( priceList.length == items.length ) {
                        return priceList;
                    }
                });
            }
        }).fail(function() {
            alert("Could not retrieve items");
        });
    }
```

**Fourth callback method performs error handling when a price can't be retrieved**

**Fifth callback method performs error handling if the items can't be retrieved**

It's easy to imagine that if you had to retrieve data from more sources, the level of callback nesting would be further increased and the code harder to understand and maintain. There are dozens of articles about callback hell and even one domain name (http://callbackhell.com) dedicated to this issue, and it's often encountered in larger Node.js (http://nodejs.org) applications.

But writing asynchronous applications doesn't need to be that hard. Callbacks, for all of their merits, are an abstraction that's too low-level to write complex asynchronous flows. JavaScript is only slowly catching up on tools and abstractions enabling a more human approach to asynchronous programming, but languages such as Scala have been designed with these abstractions in mind, leveraging well-known functional programming principles that make it possible to approach the problem from a different angle.

#### NOVEL WAYS OF WRITING ASYNCHRONOUS PROGRAMS

Tools inspired by functional programming concepts, such as Java 8 lambdas or Scala's first-order functions, greatly simplify the handling of multiple callbacks (as compared to the rather meager options that the JavaScript language provides). On top of this tooling built into the programming languages, abstractions such as *futures* and *actors* are powerful means to write and compose asynchronous request-handling pipelines, largely eliminating the phenomenon of callback hell.

Switching from an imperative, synchronous style of writing applications to a more functional and asynchronous style doesn't happen overnight. We'll discuss the tools, techniques, and mental model of asynchronous programming in chapters 3 and 5.

By adopting an evented request-handling model, Play can make much better use of a computer's resources. But what happens if, despite having an extremely performant request-processing pipeline, you hit the hardware limits of your server? Let's find out how Play can help you *scale horizontally* to several servers.

### 1.2.3 The horizontal application architecture

When developing a web application, a few fundamental choices have to be made that have a profound impact on how the web application can be operated. Unfortunately, web applications are often developed without considering what happens to the application after the code has been shipped and deployed on the production server. This can lead to profound limitations, such as when it comes to running the application on more than just one computer. If the application wasn't designed for this operational mode from the start, chances are that it won't be practicable to run it this way without significant changes to the code. In the following discussion, we'll explore a few deployment models and consider their benefits and disadvantages. We'll also look at the advantages of the so-called *horizontal deployment model* enabled and embraced by reactive applications.

#### SINGLE-SERVER DEPLOYMENTS

The single-server deployment is a very common deployment model. Web applications are deployed on a single computer, and often the database is deployed on that same computer, as shown in figure 1.4.

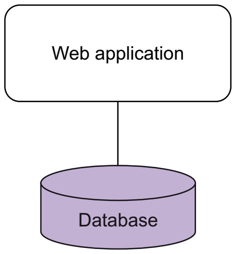


**Figure 1.4 Traditional application deployment model**

This deployment is widely used because of its relative simplicity, but it comes with a few important limitations. When the load on the server exceeds the capabilities of the hardware, or when the hardware fails, or when security or application upgrades need to be installed, the unavoidable result is that the application becomes unavailable. The usage load that this kind of setup can handle depends to a great extent on the hardware—when there's a need for more performance, a more powerful computer with more memory and faster CPUs is necessary. The process of increasing the load a server can handle by switching to more performant hardware is called *vertical scaling*.

#### REPLICATED DEPLOYMENTS

For applications that need better availability or performance, a popular setup involves *replication* of the data across two computers, as shown in figure 1.5.

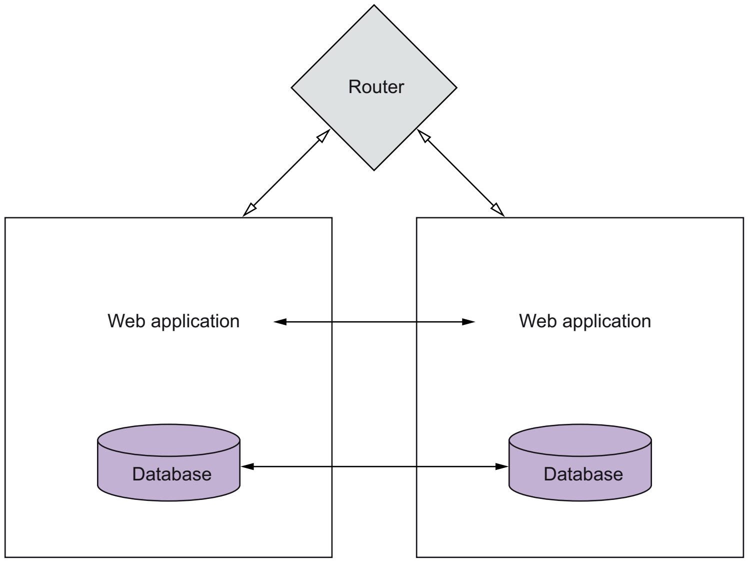


**Figure 1.5 Replicated application deployment model**

In this kind of setup, both the database and the server-side state, such as server-side user sessions or caches, need to be replicated (by making use of Apache Tomcat's clustering capabilities or similar functionality). On the database level, master-to-master replication can be employed. This solution makes it possible to update one deployment after another, thus allowing uptime during upgrades. But the complexity involved in correctly configuring this kind of setup more often than not limits the number of replicas to two. From a developer's perspective, the web application is still developed as though it were running on a single computer, and the underlying framework or application server takes care of replicating server-side state.

The complexity inherent in a multi-machine setup isn't eliminated but instead pushed to the application server. This makes it more difficult to deal with error states elegantly (without annoying the user too much), given that the error happens at a different level than the application itself and isn't a first-class concern of the application.

#### HORIZONTAL DEPLOYMENTS

In a horizontal architecture, as shown in figure 1.6, the same version of the web application is deployed across many nodes.

Those nodes may be physical computers or virtual machines, and an important characteristic about them is that they don't know anything about each other and don't share any state. This *share-nothing* principle is at the core of so-called *stateless* architectures, wherein each node is self-contained, and its presence or absence doesn't affect other nodes in any way (except, perhaps, with increased or decreased load, depending on the traffic). The advantage of such an architecture is that the application can be scaled easily by adding new nodes to a front end router, and rolling updates can be performed by bringing up new nodes with the new version and then switching the

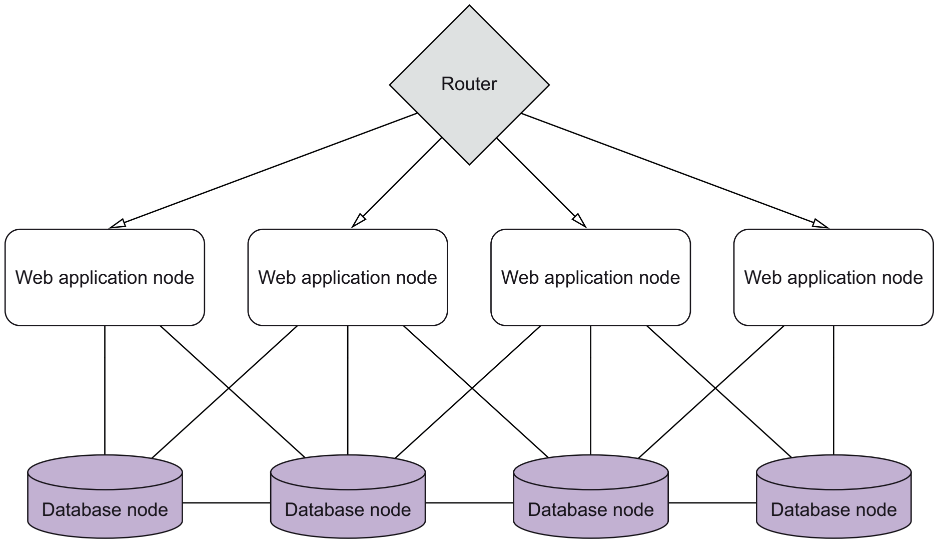


**Figure 1.6 Horizontal architecture model**

routing layer to point to those new nodes. These so-called *hot redeploy* mechanisms are popular with Platform-as-a-Service (PaaS) providers such as Heroku.

On the storage layer, a good counterpart to a share-nothing web application layer is a storage technology that supports some form of clustering. NoSQL databases such as MongoDB, Cassandra, Couchbase, and new versions of relational databases (such as WebScaleSQL; http://webscalesql.org) are a good fit for such scalable front end layers.

One consequence of using a horizontal architecture is that a user may be connected randomly to one of the front end nodes by the routing layer instead of always ending up on the same node. Given that there's no shared state between nodes, a server-side session (the default in the Servlet Standard and in frameworks built on top of it) can't be used. The Play Framework embraces the share-nothing philosophy at its core and provides a client-side user session based on cookies, which we'll talk about in chapter 8.

Thanks to its low memory footprint, Play is also a good candidate for multi-node deployments through PaaS or on other cloud-based platforms, where the amount of memory available to a single node is typically much lower than on a dedicated server.

## 1.3 Failure-handling as first-class concern

When the New York Stock Exchange (NYSE) opened at 9:30 AM on August 1, 2012, the automatic trading software of the Knight Capital Group (KCG) started trading stocks automatically, as it had been built to do and had done for many years. A few days earlier, a new version of the application had been rolled out on the servers, enabling customers of the company to participate in the Retail Liquidation Program at the NYSE. But on this August 1, things would be a little different: in the 45 minutes from when the market opened until it was shut down, the application generated a loss of 440 million USD. Oops.[7]

Building applications that don't fail is extremely difficult, and if those applications are meant to be built at a reasonable pace it's close to impossible. Instead of avoiding failure, reactive systems are designed and built from the ground up to embrace failure, leveraging the principle of *supervision*, which if employed might have prevented the fate of KCG. Reactive systems detect failure on their own and spring back into shape automatically, or degrade in such a way as to minimize catastrophic failure.

To cope with failure up front, it's important to understand what can go wrong. Let's look a bit closer at why failure is inevitable (you may not be convinced just yet that this is the case) and at what techniques can be used to cope with it.

### 1.3.1 Failure is inevitable

Unlike the development teams of the onboard shuttle group, which built the software that ran the space shuttles at a rate of a few lines of code per day,[8] most development

[7] Doug Seven, "Knightmare: A DevOps Cautionary Tale," http://dougseven.com/2014/04/17/knightmare-a-devops-cautionary-tale.

[8] Charles Fishman, "They Write the Right Stuff" (December 31, 1996), http://www.fastcompany.com/28121/they-write-right-stuff.

teams will produce software that contains errors (and hopefully, at a higher rate of lines of code per day). Even when employing test-driven development methodologies and achieving a perfect code coverage score, chances are that the software will not be entirely error-free. Applications fail because of human mistakes all the time, and increasing software quality is an iterative process. The difficulty of building failure-tolerant applications is increased many times when it comes to distributed systems running on different computers.

At the ACM Symposium on the Principles of Distributed Computing in July 2000, Eric Brewer gave a keynote speech[9] in which he presented the CAP theorem. CAP stands for *consistency*, *availability*, and tolerance to network *partitions*.

The essence of the theorem is that in the presence of network partitions, depicted in figure 1.7, you can have either *consistency* of data across servers or *availability* of all servers, but not both at the same time.

Suppose we wanted to build an online trading platform to deal with a high volume of orders. To satisfy our expected load, we set up four servers, all connected to the

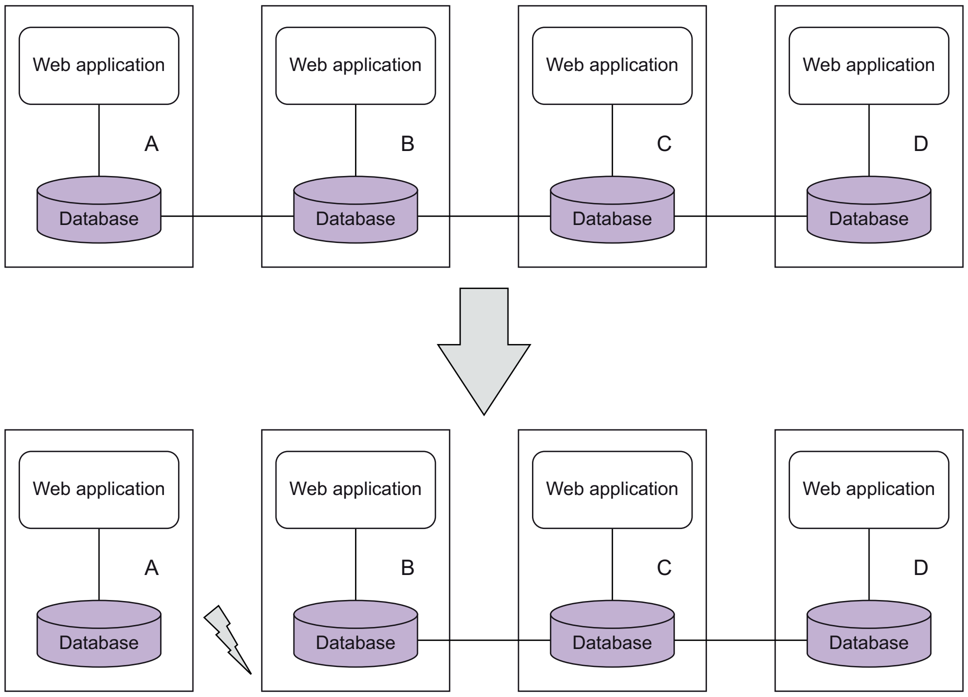


**Figure 1.7 Network partition on a system with four servers. When the partition occurs, server A is isolated from the other servers, yet the application running on the server is still reachable from the outside world. Changes occurring on this server will occur in isolation, and once the network partition is over, there may be data inconsistency between server A and servers B, C, and D.**

9 Eric Brewer, "Towards Robust Distributed Systems," www.cs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf.

internet, and additionally interconnected on a LAN. (Never mind that in practice such a setup wouldn't pass any of the security audits required for online trading. Let's just go with it for this example.) Each server is hosting a web application as well as a database that keeps data changes synchronous via replication on the LAN. When an order is placed on any of the nodes, this information is automatically propagated to all other server instances, thus ensuring the consistency of the data in our small cluster.

Now let's suppose that through an unfortunate turn of events, a member of the office cleaning personnel trips over the LAN cable of server A, thus disconnecting it from the internal network, but not from the internet. If a user now places an order via server A to buy a number of shares, and the order is successfully executed, nothing would prevent another user from placing a buy order for the same shares on any of the other nodes of the system and having it execute correctly. When the network recovers, we'd wind up with node A being in an inconsistent state, and we'd have quite a problem as a result of having sold the same shares twice.

Even if it can be argued that network partitions are rare, they still happen often enough that they can't just be overlooked. Technologies such as Amazon's DynamoDB were built with network partitions as a key part of their design.[10] Using the Command and Query Responsibility Segregation (CQRS) pattern in combination with Event Sourcing, which we'll discuss in chapter 7, is an increasingly popular mix of techniques for achieving *eventual consistency*—ensuring that even though a system may at first not be consistent at all times across all nodes, it will eventually *converge* so that all nodes see the latest version of an update.

To make things even more interesting, network partitions are just one of many things that can go wrong when working with distributed systems. In 1994, Peter Deutsch drafted out seven fallacies of distributed computing, and an additional one was added by James Gosling in 1997. The result is known as the eight fallacies of distributed computing:[11]

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous

As you can see from the length of this list, there are many reasons building a highly available system is difficult. In order for a system to be truly *resilient*, fault-tolerance can't be an afterthought—it must be handled right from the start.

[10] Giuseppe DeCandia et al., "Dynamo: Amazon's Highly Available Key-value Store," http://mng.bz/YY5A.

[11] "The Eight Fallacies of Distributed Computing," https://blogs.oracle.com/jag/resource/Fallacies.html.

## 1.3.2 *Building applications with failure in mind*

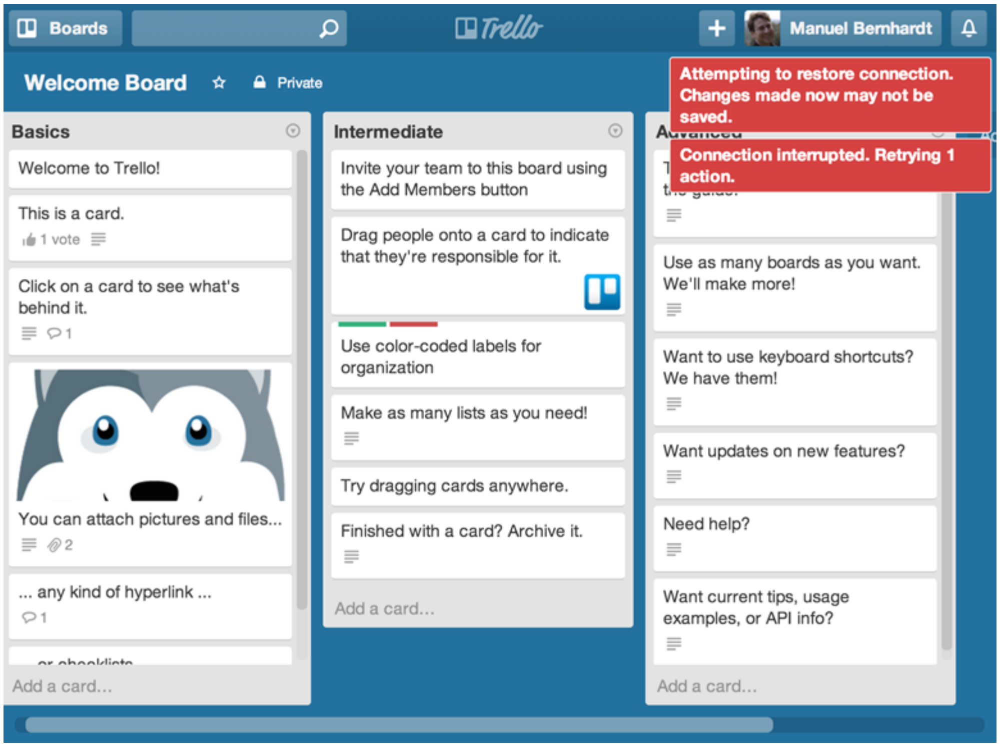
Though failure is unavoidable, there are ways to influence how a system fails and how quickly it recovers. Not every kind of failure needs to render an entire application unavailable.

### RESILIENT CLIENTS

Take, for example, the online service Trello, a project-management tool inspired by the Kanban methodology. Trello allows you to create cards and edit their content, drag and drop them from one list to another, and perform a lot more actions. When there's a problem with the network connection, be it on the client side or the server side, the Trello application doesn't simply stop responding but instead exhibits one of the most important behaviors of a reactive web application: resiliency. A user doesn't need to interrupt their work but can continue to use the service, and when the connection is recovered, the actions saved locally are transmitted back to the server. As shown in figure 1.8, users are constantly kept informed about the status of the application and made aware of situations in which their actions can't be saved properly.

### BULKHEADING

*Watertight bulkhead partitions*, used in shipbuilding for centuries, are an effective way to prevent a ship from sinking by compartmentalizing different sections. Should the ship



Attempting to restore connection. Changes made now may not be saved.

Connection interrupted. Retrying 1 action.

Welcome to Trello!

This is a card.

Click on a card to see what's behind it.

You can attach pictures and files...

... any kind of hyperlink ...

Invite your team to this board using the Add Members button

Drag people onto a card to indicate that they're responsible for it.

Use color-coded labels for organization

Make as many lists as you need!

Try dragging cards anywhere.

Finished with a card? Archive it.

Use as many boards as you want. We'll make more!

Want to use keyboard shortcuts? We have them!

Want updates on new features?

Need help?

Want current tips, usage examples, or API info?

Figure 1.8 **Failure handling and user interaction in Trello**

hit an iceberg, only the damaged compartments would be flooded—the whole ship would stay afloat if enough compartments remained intact. (It should be noted here that in the case of the Titanic, the bulkheads were not truly watertight, which explains why this mechanism did not work as designed.)

The bulkhead pattern can be used in web applications at different levels. For example, LinkedIn's home page features a lot of information kept in different sections: people you may know, people you recently visited, people who have viewed your profile, people who have viewed your updates, and so on. Those sections appear to be loaded at the same time but are in fact retrieved from various back-end services and composed together asynchronously.[12] When one of the back-end services is unavailable or takes a long time to load, the other sections aren't affected and are loaded on a first-come, first-served basis. Sections that can't get an answer from their service can render themselves differently or hide themselves entirely if necessary.

### SUPERVISION AND ACTORS

Supervision is one of the fundamental concepts used by reactive applications in order to be fault-tolerant.

When considering supervision, you might think of adult supervision of children or supervision in a work environment. In both cases, a hierarchical relationship exists—between parent and child or boss and employee. Though different in nature, these human relationships have a few common aspects:

- The supervisor (parent or boss) is responsible for the mistakes the supervised (child or employee) makes.
- The supervisor gets to know about the mistakes of the supervised (this may not always be true in reality, but let's assume it is for the sake of this explanation).
- The supervisor has to decide how to react to those mistakes.

Building on these three aspects, we can say that the core idea of supervision in the context of software systems is one of *separation of concerns*: the responsibility of executing a task is separated from the responsibility of deciding how failures are dealt with and ensuring that they are dealt with.

Joe Armstrong's thesis, "Making reliable distributed systems in the presence of software errors,"[13] introduces the Erlang computing language, designed with the idea that software, no matter how well it may be tested, always has mistakes in it. He goes on to introduce supervision as a means to counteract those mistakes when possible.

When implementing a given task, a developer may not always be able to predict all the errors that may arise. More often than not there's a degree of uncertainty when implementing an application. And even when an error condition is expected, the best reaction to the error may not be clear because it will depend on the current state of the systems. Software systems—especially distributed systems that combine many moving

[12] Yevgeniy Brikman, "Play at LinkedIn: Composable and streamable Play apps," http://www.slideshare.net/brikis98/composable-and-streamable-play-apps.

[13] A PDF of the article is available at http://mng.bz/uFsr.

parts—gain in robustness and resilience through experience and by seeing the system behave in reality. By isolating the risky parts of a task into supervised units of code, developers can acknowledge the sometimes unpredictable nature of these systems and factor the unpredictability right into their design.

A popular implementation of supervision can be found in the *actor programming model*, which is at the core of Erlang and is also represented on the JVM by Akka. It revolves around small units of software called *actors*, which, much like humans, can communicate with each other by sending and reacting to messages. Just as in human communication, messages are sent asynchronously, which means that an actor doesn't freeze and wait for a response to a message before it resumes its work.

Actors exist in a supervision hierarchy: each actor has a supervisor and can have one or several child actors for which it is responsible. Unhandled errors raised by a child actor are communicated to the parent actor, which decides to react in one way or another. We'll discuss actors in detail in chapter 6.

### 1.3.3 Dealing with load

Reactive web applications are designed to cope with varying loads. When building web applications, one critical piece of information that should flow into the design is the expected load in terms of requests per second that the application should be able to handle. This varies depending on the application: a meeting room–scheduling application on a company intranet isn't likely to generate as much interest (or be available to as many users) as a social media site for sharing funny video clips. Often, and especially in the early stages of a project, concerns about performance are dismissed as *premature optimization*, the attitude being, "We'll take care of it when we have enough users." In reality though, if a site gets popular, users won't gently and slowly visit the site turn by turn, giving developers time to come up with a way to increase capacity. Instead, the site may be featured on a popular news feed such as Hacker News, and suddenly tens of thousands of people will rush to it without warning. (Incidentally, Hacker News regularly features stories about the impact of being featured on Hacker News, sometimes including the amount of the Amazon Web Services bill.) The problem with such bursts in the number of visitors is that they are often unpredictable, and not being able to cope with them may well mean a website will lose one of its few chances to get noticed by the general public.

The capability of an application to perform well under load and to scale out to the necessary number of nodes (hardware servers or virtualized ones) can't be an afterthought. Unlike simple features such as the capability to log in using an existing Google, Facebook, or Twitter account, scalability is a *cross-cutting concern* and needs to be factored into the design right from the beginning. Reactive systems often make use of stateless architectures, which we discussed in the section "Horizontal deployments." Let's look at a few tools available for handling increased load on an application.

### CAPACITY PLANNING WITH LITTLE'S LAW

Little's law is a formula from queuing theory often used for dimensioning telecommunication infrastructures (such as traditional telephone installations). When applied to the domain of web servers, it states that

```
L = λ * W
```

where

- $L$ is the average number of requests served at the same time
- $\lambda$ is the average rate at which requests arrive on the system
- $W$ is the average time it takes to process a request

In the case of a meeting room–scheduling application for a company intranet, if there's an average of one request per minute and each request takes 100 ms to process, the average number of concurrent requests will be approximately 0.0017. In other words, there's no need to worry about scaling out for this application.

On the other hand, the site for sharing funny videos may get 10,000 requests per second (many people like to watch those videos instead of working), and if the processing time is 100 ms, the application faces on average 1,000 concurrent requests. In this case we might want to adopt a number of design and deployment decisions that allow for handling 1,000 requests at the same time. If, for example, we know that one node in our system is capable of handling 100 concurrent requests, we'll need 10 such nodes to handle the entire load.

### DYNAMICALLY SCALING IN AND OUT

As I've already pointed out, it's hard to predict the effective number of users visiting a website. The time of the day, weather conditions, and mentions via social media services may influence how high the load on the funny video clip site will be. Instead of running at full capacity all the time, it may be worth saving some money by scaling up and down depending on the load.

One approach would be to measure the effective load on the site using a monitoring tool, and then shut down or start up nodes accordingly. But as heroic as it may sound, getting up at 3:00 AM when receiving an SMS alert that the load has increased, and going online to slide the Heroku slider to the right may not be a very good strategy for the health of the website operator. Instead, using Little's law in combination with scripts to automate this process seems more reasonable. We'll look at an example of elastically scaling a Play deployment with Clever Cloud in chapter 10.

### BACK PRESSURE PROPAGATION

One of the main features of the web application for sharing funny videos is to show those videos to visitors. If we were to store the videos on a third-party storage service, such as Amazon S3, and display them using a video player on our site, we'd have to stream the video to the client through our server. If, however, the client's bandwidth was not as good as that of our server (which is often the case, especially for mobile devices), we'd need to keep the video in memory on the server for the duration of the streaming. With many users watching videos at the same time, we'd certainly run out of memory very quickly. *Back pressure propagation* is a means of regulating the speed of streams by taking into account the effective consumption speed on the consumer side. Instead of keeping the entire video in memory on our server, a setup involving back pressure would allow us to modulate the speed at which the data is retrieved from Amazon S3 so we'd only buffer a small amount in memory on the server, fetching more of it as the video is played on the user's phone.

The Play Framework builds on the concept of back pressure and utilizes it for core concerns, such as request body parsing and WebSocket handling. The Reactive Streams initiative that we briefly mentioned in the context of reactive programming provides this capability, as you'll see in chapter 9.

### CIRCUIT BREAKERS

Sometimes, it may not be possible to scale a service out, such as when communicating with a legacy application (for example, a mainframe system in a banking environment). In this case, we may need a different approach for dealing with load bursts to protect the legacy service from overload and avoid cascading failures.

In an electric circuit, a circuit breaker is an automatic switch that's meant to protect the circuit from overload or short circuits. On an abstract level, it functions as illustrated in figure 1.9.

In the context of a web application, a circuit breaker is configured to check whether the service it protects responds within a certain time frame, and if the service takes longer to answer than this timeout, the circuit trips into an open state. After a certain amount of time (which can also be configured), the circuit goes into a half-open state and a new attempt is made to contact the service. If it responds within the intended time frame, the circuit is closed again; otherwise, it trips into the open state again and waits longer for the service to recover.

Circuit breakers are an effective way to protect legacy services from overloading. Play can easily leverage the circuit breaker implementation provided by Akka, this combination having been employed successfully in a project for the Walmart Canada site.[14]

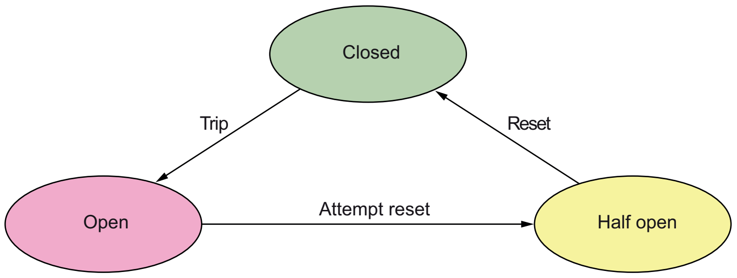


**Figure 1.9 Different states of a circuit breaker**

[14] Lightbend, "Walmart Boosts Conversions by 20% with Lightbend Reactive Platform," www.lightbend.com/resources/case-studies-and-stories/walmart-boosts-conversions-by-20-with-lightbend-reactive-platform.

## 1.4 Summary

In this chapter, you were introduced to reactive applications and why they matter. In particular, we looked at

- The meaning and origins of reactive applications and reactive technologies, including the Play Framework
- How threads are executed by a CPU and how an asynchronous, event-driven programming style embraced by evented servers makes better use of resources
- Different deployment models, including stateless, horizontal architectures that scale well under load
- The importance of failure handling and different methods that reactive applications employ to become resilient

In the next chapter, we'll get our hands dirty and build a small reactive web application with Play.

WEB DEVELOPMENT/SCALA

# Reactive Web Applications

Manuel Bernhardt

Reactive applications build on top of components that communicate asynchronously as they react to user and system events. As a result, they become scalable, responsive, and fault-tolerant. Java and Scala developers can use the Play Framework and the Akka concurrency toolkit to easily implement reactive applications without building everything from scratch.

**Reactive Web Applications** teaches web developers how to benefit from the reactive application architecture and presents hands-on examples using Play, Akka, Scala, and Reactive Streams. This book starts by laying out the fundamentals required for writing functional and asynchronous applications and quickly introduces Play as a framework to handle the plumbing of your application. The book alternates between chapters that introduce reactive ideas (asynchronous programming with futures and actors, managing distributed state with CQRS) and practical examples that show you how to build these ideas into your applications.

## What's Inside

- Reactive application architecture
- Basics of Play and Akka
- Examples in Scala
- Functional and asynchronous programming

For readers comfortable programming with a higher-level language such as Java or C#, and who can read Scala code. No experience with Play or Akka needed.

**Manuel Bernhardt** is a passionate engineer, author, and speaker. As a consultant, he guides companies through the technological and organizational transformation to distributed computing.

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit www.manning.com/books/reactive-web-applications

“You’ll come away with a solid understanding of how reactive web applications are architected, developed, tested, and deployed.”

—From the Foreword by James Roper, lead developer of the Play Framework

“Good theory and good practice, with powerful examples.”

—Steve Chaloner
Objectify

“How to be reactive in your application development… Eye-opening.”

—David Torrubia Íñigo
Fon Wireless, Ltd

“A complete and exhaustive source of best practices for large-scale, real-world reactive platforms.”

—Antonio Magnaghi, PhD
OpenMail

Free eBook
SEE INSERT

ISBN 13: 978-1-63343-009-9
ISBN 10: 1-63343-009-X
54499
9 781633 430099

MANNING $44.99 / Can $51.99 [INCLUDING eBOOK]