Reactive Design Patterns

Roland Kuhn
with Brian Hanafee and Jamie Allen
Foreword by Jonas Bonér

MANNING

Sample Chapter

Reactive Design Patterns

by Roland Kuhn

with Brian Hanafee and Jamie Allen

Chapter 1

brief contents

1 Why Reactive?

We start from the desire to build a system that is *responsive* to users. This means the system should respond to user input in a timely fashion under all circumstances. Because any single computer can fail at any time, we need to distribute such a system over multiple computers. Adding this fundamental requirement for distribution makes us recognize the need for new architecture patterns (or to rediscover old ones). In the past, we developed methods that allowed us to retain the illusion of single-threaded local processing while having it magically executed on multiple cores or network nodes, but the gap between that illusion and reality is becoming prohibitively large.[1] The solution is to make the distributed, concurrent nature of our applications explicit in the programming model, using it to our advantage.

This book will teach you how to write systems that stay responsive in the face of partial outages, program failure, changing loads, and even bugs in the code. You will see that this requires adjustments to the way you think about and design your applications. Here are the four tenets of the Reactive Manifesto,[2] which defines a common vocabulary and lays out the basic challenges that a modern computer system needs to meet:

- It must react to its users (*responsive*).
- It must react to failure and stay available (*resilient*).
- It must react to variable load conditions (*elastic*).
- It must react to inputs (*message-driven*).

[1] For example, Java EE services allow us to transparently call remote services that are wired in automatically, possibly even including distributed database transactions. The possibility of network failure or remote service overload, and so on, is completely hidden, abstracted away, and consequently out of reach for developers to meaningfully take into account.

[2] http://reactivemanifesto.org

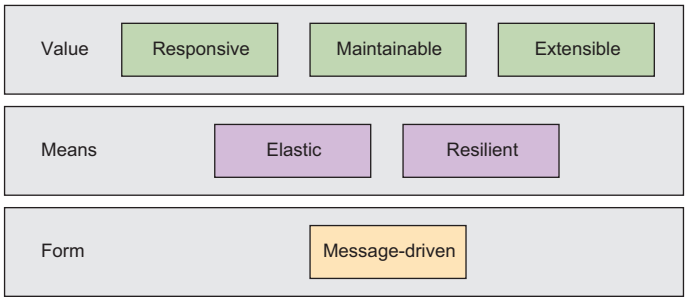

Figure 1.1 The structure of Reactive values

In addition, creating a system with these properties in mind will guide you toward better modularization, both of the runtime deployment and of the code itself. Therefore, we add two more attributes to the list of benefits: *maintainability* and *extensibility*. Another way to structure the attributes is shown in figure 1.1.

In the following chapters, you will learn about the reasoning of the Reactive Manifesto in detail, and you will get to know several tools of the trade and the philosophy behind their design, enabling you to effectively use these tools to implement reactive designs. The design patterns that emerge from these tools are presented in the third part of the book. To set the stage for diving into the manifesto, we will first explore the challenges of creating a Reactive application, using the example of a well-known email service: we will imagine a reimplementation of Gmail.

1.1 The anatomy of a Reactive application

The first task when starting such a project is to sketch an architecture for the deployment and draft the list of software artifacts that need to be developed. This may not be the final architecture, but you need to chart the problem space and explore potentially difficult aspects. We will start the Gmail example by enumerating the different high-level features of the application:

- The application must offer a view of the mailboxes to the user and display their contents.
- To this end, the system must store all emails and keep them available.
- It must allow the user to compose and send email.
- To make this more comfortable, the system should offer a list of contacts and allow the user to manage them.
- A good search function for locating emails is required.

The real Gmail application has more features, but this list will suffice for our purposes. Some of these features are more intertwined than the others: for example, displaying emails and composing them are both part of the user interface and share (or compete for) the same screen space, whereas the implementation of email storage is only distantly related to these two. The implementation of the search function will need to be closer to the storage than the front-end presentation.

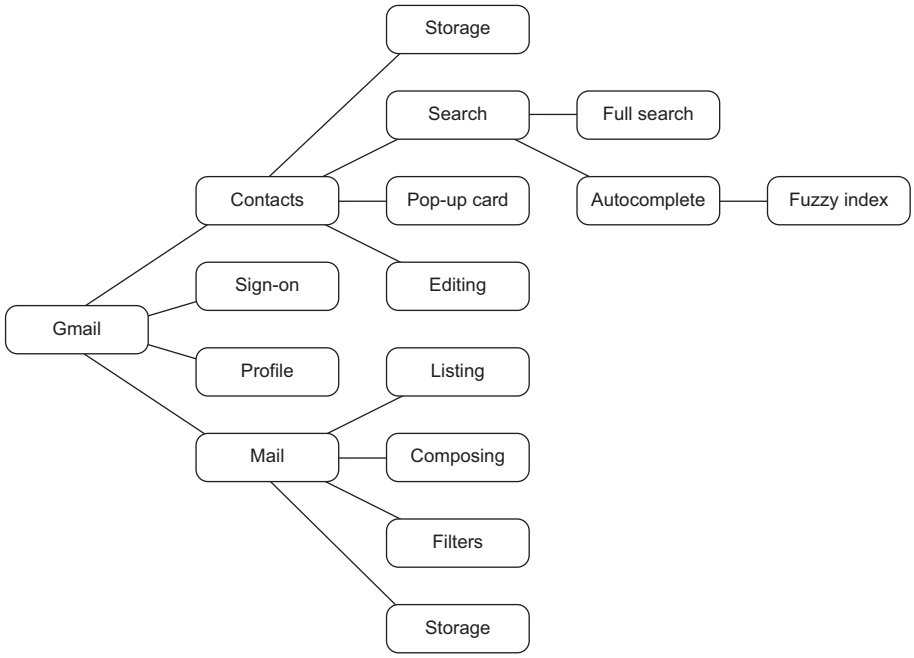

Figure 1.2 Partially decomposed module hierarchy of the hypothetical Gmail implementation

These considerations guide the hierarchical decomposition of Gmail's overall functionality into smaller and smaller pieces. More precisely, you can apply the *Simple Component pattern* as described in chapter 12, making sure you clearly delimit and segregate the different responsibilities of the entire application. The *Error Kernel pattern* and the *Let-It-Crash pattern* complement this process, ensuring that the application's architecture is well suited to reliable failure handling—not only in case of machine or network outages, but also for rare failure conditions in the source code that are handled incorrectly (a.k.a. *bugs*).

The result of this process will be a hierarchy of components that need to be developed and deployed. An example is shown in figure 1.2. Each component may be complex in terms of its function, such as the implementation of search algorithms; or it may be complex in its deployment and orchestration, such as providing email storage for billions of users. But it will always be simple to describe in terms of its responsibility.

1.2 Coping with load

The resources necessary to store all those emails will be enormous: hundreds of millions of users with gigabytes of emails each will need exabytes[3] of storage capacity. This magnitude of persistent storage will need to be provided by many distributed

[3] One exabyte is 1 billion gigabytes (using decimal SI prefixes; using binary SI prefixes, one EB is roughly 1.07 billion GB).

machines. No single storage device offers so much space, and it would be unwise to store everything in one location. Distribution makes the dataset resilient against local perils like natural disasters; but, more important, it also allows the data to be accessed efficiently from a larger region. For a worldwide user base, the data should be globally distributed as well. It would be preferable to have the emails of a Japanese user stored in or close to Japan (assuming that is where the user logs in from most of the time).

This insight leads us to the *Sharding pattern* described in chapter 17: you can split up the overall dataset into many small pieces—or *shards*—that you then distribute. Because the number of shards is much smaller than the number of users, it is practical to make the location of each shard known throughout the system. In order to find a user's mailbox, you only need to identify the shard it belongs to. You can do that by equipping every user with an ID that expresses geographical affinity (for example, using the first few digits to denote the country of residence), which is then mathematically partitioned into the correct number of shards (for example, shard 0 contains IDs 0–999,999; shard 1 contains IDs 1,000,000–1,999,999; and so on).

The key here is that the dataset naturally consists of many independent pieces that can easily be separated from each other. Operations on one mailbox never affect another mailbox directly, so the shards also do not need to communicate among themselves. Each serves only one particular part of the solution.

Another area in which the Gmail application will need a lot of resources is in the display of folders and emails to the user. It would be impossible to provide this functionality in a centralized fashion, not only for reasons of latency (even at the speed of light, it takes noticeable time to send information around the globe) but also due to the sheer number of interactions that millions of users perform every second. Here, you will also split the work among many machines, starting with the users' computers: most of the graphical presentation is rendered within the browser, shifting the workload very close to where it is needed and in effect sharding it for each user.

The web browser will need to get the raw information from a server, ideally one that is close by to minimize network round-trip time. The task of connecting a user with their mailbox and routing requests and responses accordingly is one that can also easily be sharded. In this case, the browser's network address directly provides all needed characteristics, including an approximate geographic location.

One noteworthy aspect is that in all the aforementioned cases, resources can be added by making the shards smaller, distributing the load over more machines. The maximum number is given by the number of users or used network addresses, which will be more than enough to provide sufficient resources. This scheme will need adjustment only when serving a single user requires more computing power than a single machine can provide, at which point a user's dataset or computing problem needs to be broken down into smaller pieces.

This means that by splitting a system into distributable parts, you gain the ability to scale the service capacity, using a larger number of shards to serve more users. As long as the shards are independent from each other, the system is in theory infinitely

scalable. In practice, the orchestration and operation of a worldwide deployment with millions of nodes requires substantial effort and must of course be worth it.

1.3 Coping with failures

Sharding datasets or computational resources solves the problem of providing sufficient resources for the nominal case, when everything is running smoothly and networks are operational. In order to cope with failures, you need the ability to keep running when things go wrong:

- A machine may fail temporarily (for example, due to overheating or kernel panic) or permanently (electrical or mechanical failure, fire, flood, and so on).
- Network components may fail, both within a computing center as well as outside on the internet—including the case that intercontinental overseas cables go down, resulting in a split of the internet into disconnected regions.
- Human operators or automated maintenance scripts may accidentally destroy parts of the data.

The only solution to this problem is to replicate the system—its data or functionality—in more than one location. The geographical placement of the replicas needs to match the scope of the system; a global email service should serve each customer from multiple countries, for example.

Replication is a more difficult and diverse topic than sharding because intuitively you mean to have the same data in multiple places—but keeping the replicas synchronized to match this expectation comes at a high cost. Should writing to the nearest location fail or be delayed if a more distant replica is momentarily unavailable? Should it be impossible to see the old data on a distant replica after the nearest one has already signaled completion of the operation? Or should such inconsistency just be unlikely or very short-lived? These questions will be answered differently between projects or even for different modules of one particular system. Therefore, you are presented with a spectrum of solutions that allows you to make trade-offs between operational complexity, performance, availability, and consistency.

We will discuss several approaches covering a wide range of characteristics in chapter 13. The basic choices are as follows:

- *Active–passive replication*—Replicas agree on which one of them can accept updates. Fail-over to a different replica requires consensus among the remaining ones when the active replica no longer responds.
- *Consensus-based multiple-master replication*—Each update is agreed on by sufficiently many replicas to achieve consistent behavior across all of them, at the cost of availability and latency.
- *Optimistic replication with conflict detection and resolution*—Multiple active replicas disseminate updates and roll back transactions during conflict or discard conflicting updates that were performed during a network partition.

- *Conflict-free replicated data types*—This approach prescribes merge strategies such that conflicts cannot arise by definition, at the cost of providing only eventual consistency and requiring special care when creating the data model.

In the Gmail example, several services should provide consistency to the user: if a user successfully moves an email to a different folder, they expect it to stay in that folder regardless of which client they use to access their mailboxes. The same goes for changes to a contact's telephone number or the user's profile. For these data, you could use active–passive replication to keep things simple by making the failure response actions coarse-grained—that is, on a per-replica scope. Or you could use optimistic replication under the assumption that a single user will not concurrently make conflicting changes to the same data item—but keep in mind that this is a fair assumption only for human users.

Consensus-based replication is needed within the system as an implementation detail of sharding by user ID, because the relocation of a shard must be recorded accurately and consistently for all clients. It would lead to user-visible distortions like an email disappearing and then reappearing if a client were to flip-flop between decommissioned and live replicas.

1.4 Making the system responsive

The previous two sections introduced reasons for distributing the system across several machines, computing centers, or possibly even continents, matching the scope and reliability requirements of the application. The foremost purpose of this exercise is to build an email service for end users, though, and for them the only metric that counts is whether the service does what they need when they need it. In other words, the application must respond quickly to any request a user makes.

The easiest way to achieve this is, of course, to write an application that runs locally and that has all emails stored on the local machine as well: going across the network to fetch an answer will always take longer and be less reliable than having the answer close by. There is, thus, a tension between the need to distribute and the need to stay responsive. All distribution must be justified, as in the Gmail example.

Where distribution is necessary, you encounter new challenges in the quest for responsiveness. The most annoying behavior of many distributed applications today is that their user interaction grinds to a halt when network connectivity is poor. Interestingly, it seems much simpler to deal with the complete absence of a connection than with a trickling flow of data. One pattern that is helpful in this context is the *Circuit Breaker pattern* discussed in detail in chapter 12. With this tool, you can monitor the availability and performance of a service that you are calling on for some function so that when the quality falls below a threshold (either too many failures or too long a response latency), the circuit breaker trips, forcing a switch to a mode where that service is not used. The unavailability of parts of the system needs to be considered from the beginning; the Circuit Breaker pattern addresses this concern.

Another threat to responsiveness arises when a service that the application depends on becomes momentarily overloaded. A backlog of requests will accumulate, and while these are processed, response latencies will be much longer than normal. This situation can be avoided by employing *flow control,* as described in chapter 16. In the Gmail example, there are several points at which circuit breakers and flow control are needed:

- Between the front end that runs on the users' devices and the web servers that provide access to back-end functionality
- Between the web servers and back-end services

The reason for the first point has already been mentioned: the desire to keep the user-visible part of the application responsive under all conditions, even if sometimes the only thing it can do is signal that the server is down and that the request will be completed at a later time. Depending on how much functionality can or should practically be duplicated in the front end for this *offline mode*, some areas of the user interface may need to be deactivated.

The reason for the second point is that the front end would otherwise need to have different circuit breakers for different kinds of requests to the web server, each circuit breaker corresponding to the specific subset of back-end services needed by one kind of request. Switching the entire application to offline mode when only a small part of the back-end services are unavailable would be an unhelpful over-response. Tracking this in the front end would couple its implementation to the precise structure of the back end, requiring the front-end code to be changed whenever the service composition of the back end was altered. The web-server layer should hide these details and provide its clients with responses as quickly as possible under all circumstances.

Take, for example, the back-end service that provides the information shown on the contact card that pops up when hovering the pointer over an email sender's name. This is a nonessential function, considering the overall function of Gmail, so the web server may return a temporary failure code for such requests while that back-end service is unavailable. The front end does not need to track this state; it can merely refrain from showing the pop-up card and retry the request when interaction with the user triggers it again.

This reasoning applies not only at the web server layer. In a large application that consists of hundreds or thousands of back-end services, it is imperative to confine the treatment of failure and unavailability in this fashion; otherwise, the system would be unreasonable in the sense that its behavior could no longer be understood by humans. Just as functionality is modularized, the treatment of failure conditions must be encapsulated in comprehensible scopes as well.

1.5 Avoiding the ball of mud

The Gmail application at this point consists of a front-end part that runs on the user's device, back-end services that provide storage and functionality, and web servers that act as entry points into the back end. The latter serve an important purpose beyond the responsiveness discussed in the previous section: they decouple the front end from the back end architecturally. Having this clearly defined ingress point for client requests makes it simpler to reason about the interplay between the part of the application that runs on the users' devices and the part that runs on servers in the cloud.

The back end so far consists of a multitude of services whose partitioning and relationships resulted from the application of the Simple Component pattern. By itself, this pattern does not provide the checks and balances that keep the architecture from devolving into a large mess where every service talks with almost every other service. Such a system would be hard to manage even with perfect individual failure handling, circuit breakers, and flow control; it certainly would not be possible for a human to understand it in its entirety and confidently make changes to it. This scenario has informally been called the *big ball of mud*.

With the problem lying in the unrestrained interaction between arbitrary back-end services, the solution is to focus on the communication paths within the entire application and to specifically design them. This is called *message flow* and is discussed in detail in chapter 15.

The service decomposition shown in figure 1.2 is too coarse-grained to serve as an example for a "ball of mud," but an illustration for the principle of message-flow design would be that the service that handles email composition probably should not talk directly to the contact pop-up service: if composing an email entails showing the contact card of someone mentioned in the email, then instead of making the back end responsible for that, the front end should ask for the pop-up, just as it does when the user hovers the mouse pointer over an email header. In this way, the number of possible message-flow paths is reduced by one, making the overall interaction model of back-end services a little simpler.

Another benefit of carefully considering message flow lies in facilitating testing and making it easier to ensure coverage of all interaction scenarios. With a comprehensive message-flow design, it is obvious which other services a component interacts with and what is expected from the component in terms of throughput and latency. This can be turned around and used as a canary in the coal mine: whenever it is difficult to assess which scenarios should be tested for a given component, that is a sign that the system is in danger of becoming a big ball of mud.

1.6 Integrating nonreactive components

The final important aspect of creating an application according to Reactive principles is that it will, in most cases, be necessary to integrate with existing systems or infrastructure that does not provide the needed characteristics. Examples are device drivers that lack encapsulation (for example, by terminating the entire process in case of

failure), APIs that execute their effects synchronously and thereby block the caller from reacting to other inputs or timeouts in the meantime, systems with unbounded input queues that do not respect bounded response latency, and so on.

Most of these issues are dealt with using the *resource-management patterns* discussed in chapter 14. The basic principle is to retrofit the needed encapsulation and asynchronous boundaries by interacting with the resource within a dedicated Reactive component, using extra threads, processes, or machines as necessary. This allows these resources to be integrated seamlessly into the architecture.

When interfacing with a system that does not provide bounded response latency, it is necessary to retrofit the ability to signal momentary overload situations. This can to some degree be achieved by employing circuit breakers, but in addition you must consider what the response to overload should be. The *flow-control patterns* described in chapter 16 help in this case as well.

An example in the context of the Gmail application is a hypothetical integration with an external utility, such as a shared shopping list. Within the Gmail front end, the user can add items to the shopping list by extracting the needed information semiautomatically from emails. This function would be supported in the back end by a service that encapsulates the external utility's API. Assuming that the interaction with the shopping list requires the use of a native library that is prone to crash and bring down the process it is running in, it is desirable to dedicate a process to this task alone. This encapsulated form of the external API is then integrated via the operating system's interprocess communication (IPC) facilities, such as pipes, sockets, and shared memory.

Assuming further that the shopping list's implementation employs a practically unbounded input queue, you need to consider what should happen when latencies increase. For example, if it takes minutes for an item to show up on the shopping list, users will be confused and perhaps frustrated. A solution to this problem would be to monitor the shopping list and observe the latency from the Gmail back-end service that manages this interaction. When the currently measured latency exceeds the acceptable threshold, the service will either respond to requests with a rejection and a temporary failure code, or perform the operation and include a warning notice in the response. The front-end application can then notify the user of either outcome: in one case it suggests retrying later, and in the other it informs them about the delay.

1.7 Summary

In this chapter, we explored the Reactive landscape in the context of the principles laid out in the Reactive Manifesto and surveyed the main challenges facing you when building applications in this style. For a more detailed example of designing a Reactive application, please refer to appendix B. The next chapter takes a deep dive into the manifesto itself, providing a detailed discussion of the points that are condensed into a compressed form in appendix C.

SOFTWARE DEVELOPMENT

REACTIVE DESIGN PATTERNS

ROLAND KUHN
WITH BRIAN HANAFEE AND JAMIE ALLEN

Modern web applications serve potentially vast numbers of users—and they need to keep working as servers fail and new ones come online, users overwhelm limited resources, and information is distributed globally. A Reactive application adjusts to partial failures and varying loads, remaining responsive in an ever-changing distributed environment. The secret is message-driven architecture—and design patterns to organize it.

Reactive Design Patterns presents the principles, patterns, and best practices of Reactive application design. You'll learn how to keep one slow component from bogging down others with the Circuit Breaker pattern, how to shepherd a many-staged transaction to completion with the Saga pattern, how to divide datasets by Sharding, and much more. You'll even see how to keep your source code readable and the system testable despite many potential interactions and points of failure.

WHAT'S INSIDE

- The definitive guide to the *Reactive Manifesto*
- Patterns for flow control, delimited consistency, fault tolerance, and much more
- Hard-won lessons about what doesn't work
- Architectures that scale under tremendous load

Most examples use Scala, Java, and Akka. Readers should be familiar with distributed systems.

Dr. Roland Kuhn led the Akka team at Lightbend and coauthored the *Reactive Manifesto*. **Brian Hanafee** and **Jamie Allen** are experienced distributed systems architects.

"Does an excellent job explaining Reactive architecture and design, starting with first principles and putting them into a practical context."

—From the Foreword by Jonas Bonér
Creator of Akka

"If the Reactive Manifesto gave us a battle cry, this work gives us the strategic handbook for battle."

—Joel Kotarski, The Rawlings Group

"An engaging tour of distributed computing and the building blocks of responsive, resilient software."

—William Chan, LinkedIn

"This book is so reactive, it belongs on the left-hand side of the periodic table!"

—Andy Hicks, Tanis Systems

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

ISBN-13: 978-1-61729-180-7
ISBN-10: 1-61729-180-3
54999
9 781617 291807

MANNING US $ 49.99 | Can $65.99