Reactive Applications with Akka.NET

Anthony Brown

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Reactive Applications with Akka.net
Version 12

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

welcome

Thank you for purchasing the MEAP for *Reactive Applications with Akka.NET*, I hope that this book lays the solid foundations needed to ensure that you're able to make the most of the reactive manifesto to create applications and services which are truly capable of standing the trials and tribulations of a wide audience.

As software developers, we're at an interesting time thanks to the significant growth in popularity of computers of all shapes and sizes, whether they're in traditional devices such as laptops and desktops, smart entertainment devices such as TVs or in the booming Internet of Things market. This increase in demand is leading to the requirement for new and innovative solutions to be able to handle such a demand.

As you work through the book, you'll see how the Reactive Manifesto and reactive concepts fit into this new era of software development. In part 1 of the book we'll see an overview of reactive and why it's needed over the coming years as well as a more in depth look at how we're able to design new systems with the reactive traits in mind. From there, we'll introduce Akka.Net, an actor model implementation in .Net which allows us to write applications in the reactive style. Following these introductions, we'll start to build up an understanding of the fundamentals of writing applications using Akka.Net before we look at how we're able to apply these principles in the applications we write, all thanks to Akka.Net and the Akka.Net ecosystem.

Reactive Applications with Akka.NET has been written with a focus on those with little to no experience with any of Akka.Net, the actor model or reactive systems but experience of the difficulties you might encounter when you try and make applications which are resilient and scalable. I hope the book helps you on the journey to a thorough understanding of reactive applications and how using Akka.Net you might be able to alleviate some of these difficulties you've experienced in the past.

Since the title is available through MEAP, I implore you to leave comments such that I'm able to ensure that you and others have a solid understanding of what reactive is all about.

Once again, thank you,
—Anthony Brown

brief contents

1

Why reactive?

1.1 What does it mean to be reactive?

Over the past several decades, computers and the internet have moved from a position of relative obscurity into being a core component of many aspects of modern life. You now rely on the internet for all manner of day-to-day tasks, including shopping and keeping in contact with friends and family. This proliferation of computers and devices capable of accessing the internet has increased pressure on software developers to create applications that are able to withstand this near-exponential growth, meaning that you need to develop applications that are able to meet the demands of a modern populous who have grown to become dependent upon technology. Demands range in scope from a desire to provide information to users as soon as it's available, to the need to ensure that the services you provide are resilient to any issues they might encounter due to increased usage or an increased likelihood of failure, which may be caused by factors entirely outside of your control. When this is twinned with the demands of a rapidly evolving company, trying to beat the competition to find new gaps in an ever-changing marketplace, you're left needing to build applications that are not only able to stand satisfy demands imposed by users, but are also designed to be sufficiently malleable that they can be rapidly adapted and modified to fill potential gaps in the market.

In response to this, technology companies working across a broad range of different domains began to notice common patterns that were able to fulfill these new requirements. Common trends began to emerge, which were clearly visible to companies building the next generation of modern applications with a strong focus on huge datasets, up to the petabyte scale in certain instances, which needed to be analyzed and understood in record time, with results being delivered to users at near-instantaneous speeds. After following these principles, these systems were seen to be more robust, more resilient and more open to change. These principles were collected together and form the outcomes one can expect when you develop applications by implementing the Reactive Manifesto: a set of common shared traits that

exemplify a system design capable of standing up to the challenges we are exposed to when building applications to fulfill users' demands.

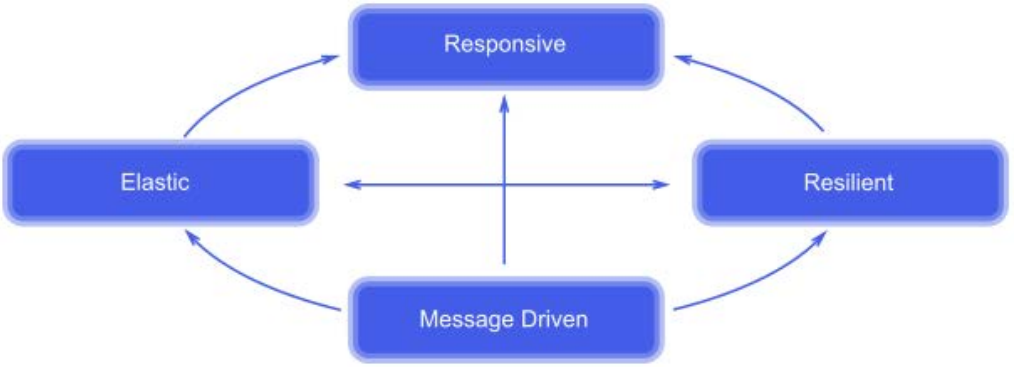
1.1.1 The heart of the Reactive Manifesto

At the core of the Reactive Manifesto is a common understanding that applications designed according to these traits should be able to respond to changes in their environment as quickly as is physically possible. A change in the environment could include any number of variables, whether it's a response to a change in the data of another component in the system, an increase in the error rate when attempting to process data or communicate with an external system, or an increase in the amount of data flowing through the system across component boundaries.

The implication of this is that the most important property of a modern application is that it should be responsive, and should respond to requests from users as quickly as possible. For example, in the context of a web application, the user should expect to see changes as soon as possible, whether this is by the application pushing data changes directly down to the user's web browser, or ensuring that it is able to be retrieved as quickly as possible when the user next requests the change. The term "responsive" is quite broad, and the definition of responsive in one domain may be vastly different to responsive in another context, so some consideration should be applied to what responsive means when applied to your applications. Many of the examples in this book apply to either web applications or real-time data solutions. These two cases alone include a number of potential interpretations of what responsive means. But you can exemplify these two cases as such: a web application should be responsive by responding to an HTTP request in the shortest possible time, whereas a data streaming solution should ensure data flows at a constant rate in an effort to prevent a stalled stream, which may have knock-on effects for other components earlier in the stream.

In order to achieve this level of responsiveness, you need to ensure that the systems you design are able to handle greater scale. Let's consider the example of a web application again. If this receives more web requests than the server is capable of handling, then it is inevitable that the incoming requests will start to be queued up, until the resources are available to service the requests. This queuing then leads to an increase in the response time for users, ultimately making the application less responsive. Similarly, in the case of a streaming data solution, you need to ensure that if more events start to flow through the stream, then you should have the ability to process them within a fixed amount of time; otherwise, it may cause the results for subsequent events to be delayed. But it's not enough to constantly provide more computing power; while computing power has dropped in price significantly in recent years, it's still far from cheap. As such, you should be able to respond to periods of inactivity or reduced throughput by negatively adjusting your provisioned compute resources so that you don't have to maintain or pay for these unnecessary resources. This can ultimately be categorized as the need for systems to be designed with elasticity in mind, that is, have the ability to expand when needed, but shrink back down to a bare minimum set of resources for the system to continue to operate when not.

In parallel to elasticity, it's important that systems are equally resilient, that is, they're able to react to a failure, whether it's a failure that originated from within the system, which you have some degree of control over, or it originates from other systems external to yours and over which you have no control. Within a streaming data solution, this might translate into the ability for your stream processing system to handle a situation in which you start to receive bad or invalid data from an incoming data source. For example, if this was an Internet of Things device sending sensor data, then your stream processing solution should be able to handle incoming data that may contain invalid sensor readings caused by a faulty sensor. If your application started to fail, then this would likely cause knock-on failures in other components within your system. Therefore, it's important that for an application to be resilient it should focus on the containment of errors in the smallest possible area of the application. Following this containment, you should then seek to recover from these failures automatically, without the need for manual intervention. This notion of resilience then ensures that the client does not end up being burdened with the responsibility for handling any failures that may occur within the system.



Finally, in order to drive the concepts we've seen thus far, message-driven systems are the core component that links everything together. By using messaging as the basis of communication between components within the system, you're able to perform work asynchronously and in a completely non-blocking manner. This ensures that you're able to perform more work in parallel, leading to an increase in overall responsiveness. By using message passing as the basis of communication, you're also able to redirect and divert messages at runtime as appropriate, allowing you to reroute a message from a failing component to one that is able to service the request. For example, if you had two servers, each of which was capable of servicing a request, then by using message passing you're able to change which server ultimately receives the request if one server becomes unavailable to service it. Similarly, if you notice one server has become a bottleneck, then you're able to divert a message to another server which is able to service the request. This means that you're able to dynamically add or remove new instances and automatically redirect the messages to the target instance.

You can see how these concepts work together, with messaging being the core building block that powers the resilience, elasticity, and responsiveness of the application. You can also see that elasticity and resilience are a shared concern; once you have the underlying infrastructure in place for resilience, this also provides the necessary logic for elasticity. Once all of these concepts are linked together, you're left with applications that are ultimately responsive.

1.1.2 Reactive systems vs reactive programming

The concepts surrounding the Reactive Manifesto are far from new, and ultimately stretch back over several decades. The manifesto is itself a formalization of a significant amount of domain knowledge from varying organizations. Due to the relatively broad concepts covered in the Reactive Manifesto, there is some overlap between two somewhat related programming concepts: reactive programming and reactive systems.

Reactive programming, like the programming model offered by Reactive Extensions, offers a smaller-scale overview of reactive programming, tailored to how data flows in a single application. Typical applications are driven by a threaded execution model, in which operations are performed sequentially in an order that you've defined, leaving you to deal with many of the underlying flow control primitives needed to synchronize data. In contrast, reactive programming is driven by the execution of code only when new data is available; typically, this is in the form of events arising from a data source. One such example of this would be a timer that ticks once every 5 minutes. Using typical programming patterns, you'd have to set up a loop that continuously polls until the minimum time period has elapsed before you progress through your application flow. But, when dealing with reactive programming, you create handlers that receive an event and are executed whenever a new event is received.

Reactive systems, however, focus on applying the same concepts on a much larger scale that deals with the integration of multiple distinct components. Many of the applications you build today are no longer basic programs, taking in an input and producing an output; instead, they are complex systems made up of an array of components, where each component could itself be an entire system. This level of interconnectedness brings with it complexities. Systems might not be running on the same physical hardware, and in fact may not even be collocated, with one system existing thousands of miles from another. This means that you now need to consider what happens in the event of failures, or how other system components will respond in the event of sudden floods of information passing through the system. We saw when discussing the Reactive Manifesto that these are the core requirements for an overall system to remain responsive, and we saw the way to achieve these aims was through the use of a higher-level message passing-based API.

This is the core difference between reactive programming and a reactive system. Reactive programming involves the notion of events: data that is broadcast to everybody who is listening to that event. This can be compared to reactive systems. These are message-driven, which brings with it individually addressable components to support targeted messages. Akka.NET is one example of a tool that simplifies the building of larger-scale reactive systems,

which we'll be seeing throughout this book, whereas Reactive Extensions is an example of reactive programming, which you won't be considering in this book. The two concepts can, however, be combined, with reactive programming being built on top of a reactive system, or reactive programming existing within a single component of a reactive system. But the combination of these concepts won't be addressed in this book.

1.2 Applying Akka.NET

Akka.NET is positioned as a platform upon which reactive systems can be built. This opens the door to using it across multiple distinct domains. It has seen similar use in Internet of Things-based applications, e-commerce, finance, and many other domains. But it is the internal requirements of these applications that drive whether Akka.NET is an ideal fit. One of the common concerns across these types of application is the requirement to update components based on the results of operations from previous components. Akka.NET starts to become a powerful tool once you need immediate responses from multiple components all integrated together.

1.2.1 Where to use Akka.NET

One example of where Akka.NET would be an ideal fit is in the world of commercial air travel. Here, multiple distinct components all produce data at an incredible rate, which needs to be processed and delivered to the user as soon as possible. For example, while a passenger sits in the terminal waiting to board their flight, they wait to see which gate their flight will depart from. This is particularly important in large airports where it might take 20-30 minutes to walk between gates. But there are a vast number of systems that are all integrated together and dictate where a flight ultimately travels and ends up. There is also national air traffic control, which reroutes flights in the event of an emergency and to prevent in-air collisions between planes in a congested airspace. There is also the airport's air traffic control, responsible for directing planes towards the correct runway; in the case of a large airport, landing on a difference runway could force the plane to a different gate. There are also airport operations that may be forced to divert a flight to a different gate due to a scheduling issue with another airline, which prevents the plane from arriving at that gate. Similarly, there's also data from the airline's internal systems, which might force a gate change due to internal scheduling problems. A vast array of data sources all publish data that needs to be processed as quickly as possibly, so that passengers are immediately aware of any changes that might occur as part of the effort to ensure that aircraft are able to turn around and take off as soon as possible after landing.

Although not all systems are as complex, or rely on as many distinct data sources, as an airline, there is a clear pattern of integrating multiple components together into a larger system, while also catering for any difficulties that might be encountered in the process. An airline, for instance, needs to immediately respond to changes when they are published, in an effort to protect the safety and security of all passengers and staff involved.

1.2.2 Where not to use Akka.NET

Although Akka.NET makes it easier to build larger reactive systems, it brings with it some difficulties. We've already seen how complex systems can be, which ultimately forces you to consider these complexities. For example, you need to think about partial failures of system components that might impact other components, you need to consider data consistency and how that should be handled in the case of partial failures, and plenty of other issues. Akka.NET brings these difficulties and complexities to the surface as first-class principles, which ultimately means that you have to deal with them. In dealing with them, you also bring to the surface a number of other complexities, notably harder debugging and the requirement to think about concurrency. Therefore, for fairly simple web applications that have basic requirements, Akka.NET is unlikely to provide any significant benefits. This includes relatively basic CRUD (Create, Read, Update, Delete) applications that are backed by a basic database model.

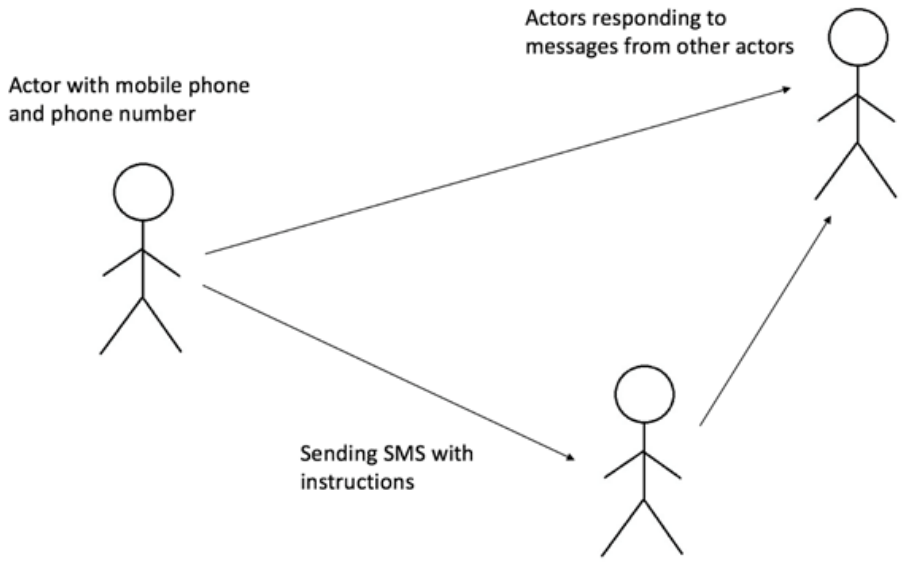
At its core, Akka.NET provides a concurrency model that is designed to allow multiple components to operate simultaneously. This means that when developing systems with Akka.NET, you need to think carefully about the data in your system. Although Akka.NET removes the possibility of concurrent access to shared data, there is still the opportunity for data races to occur, as well as the potential for deadlocks. For a system that doesn't need to operate concurrently, Akka.NET is likely to complicate matters and force more complications into your application, rather than simplifying it further.

1.3 How does Akka.NET work?

Although the concept of Akka.NET and how it works might be new to many developers, the underlying principles have been in development for decades, in the form of the actor model. As part of the actor model, independent entities, known as actors, are responsible for performing work. You can have multiple different types of actor within a system, and each of these types can have multiple instances running within the system. Every actor runs independently of every other actor within the system, meaning that two running actors are not capable of directly interfering or interacting with one another. Instead, each actor is supplied with a mailbox, which receives messages, and an address, which can be used to receive messages from other actors within the system. Actors will sit idle and not do anything until a new message is received in the actor's mailbox; at this point, the actor is able to process the message using its internal behavior. Its behavior is the brain of the actor and defines how it should respond to each message it receives. If an actor receives more than one message, the messages are queued up in the order in which they were received, and the actor processes each message sequentially. Each actor will only process a single message at any one time, although multiple actors can process their respective message at the same time. This allows you to create highly concurrent applications, without having to concern yourself with the underlying multithreading infrastructure and code that is typically required when developing concurrent applications. It's important to also note that the actors are completely isolated, meaning that any internal information or data owned by an actor is not accessible by anything other than that actor.

You can think of actors as being similar to people with a mobile phone. Each person has an address through which they can be contacted; in this case, the address is the phone number of the user you are calling. You also have access to a unique address for the person initiating the phone call; once again this is a phone number. Once you've got that address, you're able to communicate with the person you want to talk to. You can do this by sending an SMS with some data in it. As humans, the data you might include in an SMS is typically a question if you want to know an answer, or a statement if you want to inform the other person of something. The SMS you send ends up in the other person's mailbox, where they're able to asynchronously deal with it when they have the resources and bandwidth available. Like actors, every person is an independent, isolated entity with no ability to directly access other people's information. If you want to find out what plans a friend has for the weekend, you don't have direct access to that information; instead, what you do is send them an SMS asking for the information. This is the same pattern you use when sending data between actors: rather than directly accessing an actor's data, you send the actor a message asking for it and await the response.

Similar to humans, actors are able to perform a number of operations upon receiving a new message. The simplest approach is to ignore a message; if it's particularly important, the other party will resend the message and attempt to retrieve a response a second time. Alternatively, it could choose to send a message elsewhere in response to receiving a message. The actor might not have all the information available to create a complete response, but it's able to contact other actors within the system, who might have the information available, after which the actor is able to act on the message. For a particularly intense or long-running task, an actor can also spawn another actor that is solely responsible for performing that task. This is similar to how people delegate work to other people if they lack the time needed to perform the task, or if they have other pressing matters to attend to. You can also choose how you respond to the next message you receive by modifying the internal state of the actor. This is analogous to hearing some new information from one person that then influences answers to questions you receive from other people.



The core takeaway when considering the actor model is that its core design principle is to form an abstraction over the top of low-level multithreading concepts to simplify the process of developing concurrent applications. Understanding this, combined with the isolated nature of individual actors, ensures that the systems you build on top of Akka.NET are able to line up with the traits of the Reactive Manifesto.

1.4 What will you learn in this book?

This book focuses on how Akka.NET provides abstractions, which you as a developer can use to reduce the complexity of reactive applications. This chapter has shown the core traits that make up the Reactive Manifesto; in later chapters, we'll see how you can apply concepts from Akka.NET to closely align your systems with the aims of the Reactive Manifesto. In each chapter, we'll also look at a short case study where we will see how the Akka.NET feature can be used in a real-world application to help model a solution, using concepts covered in that chapter. In addition to these shorter case studies, two larger case studies will also be presented, showing the application of the Reactive Manifesto to a small component of an e-commerce application, as well as the use of Akka.NET to model a reactive solution to an Internet of Things solution. We'll also see the inherent complexities involved in building larger systems, where the actor model of Akka.NET helps to bring these issues to the forefront so that you are forced to consider the ramifications of failure within your system or supporting systems.

In order to get the most out of this book, you should be a software developer with some experience of the .NET framework, and specifically the C# language. All examples presented in this book will use the C# Akka.NET API, which is equally usable by other .NET languages, but

more idiomatic APIs are available and while the concepts remain the same, there may be significant differences between APIs. An understanding of the tradeoffs that must be made when developing larger systems in .NET is also beneficial, as well as a grasp of the difficulties typically encountered when developing asynchronous applications.

1.5 Conclusion

In this chapter, you learned:

- The core reasoning behind the move to reactive systems
- The underlying programming model behind Akka.NET