

Akka

IN ACTION

Raymond Roestenburg
Rob Bakker
Rob Williams





Akka in Action

by Raymond Roestenburg

Rob Bakker

Rob Williams

brief contents

1	■	Introducing Akka	1
2	■	Up and running	28
3	■	Test-driven development with actors	49
4	■	Fault tolerance	66
5	■	Futures	92
6	■	Your first distributed Akka app	118
7	■	Configuration, logging, and deployment	147
8	■	Structural patterns for actors	166
9	■	Routing messages	188
10	■	Message channels	213
11	■	Finite-state machines and agents	233
12	■	System integration	254
13	■	Streaming	281
14	■	Clustering	322
15	■	Actor persistence	354
16	■	Performance tips	388
17	■	Looking ahead	416

Introducing Akka

In this chapter

- Why scaling is hard
- Write once, scale anywhere
- Introduction to the actor programming model
- Akka actors
- What is Akka?

Up until the middle of the '90s, just before the internet revolution, it was completely normal to build applications that would only ever run on a single computer, a single CPU. If an application wasn't fast enough, the standard response would be to wait for a while for CPUs to get faster; no need to change any code. Problem solved. Programmers around the world were having a free lunch, and life was good.

In 2005 Herb Sutter wrote in *Dr. Dobbs's Journal* about the need for a fundamental change (link: <http://www.gotw.ca/publications/concurrency-ddj.htm>). In short: a limit to increasing CPU clock speeds has been reached, and the free lunch is over.

If applications need to perform faster, or if they need to support more users, they will have to be *concurrent*. (We'll get to a strict definition later; for now let's

simply define this as *not single-threaded*. That's not really correct, but it's good enough for the moment.)

Scalability is the measure to which a system can adapt to a change in demand for resources, without negatively impacting performance. *Concurrency* is a means to achieve scalability: the premise is that, if needed, more CPUs can be added to servers, which the application then automatically starts making use of. It's the next best thing to a free lunch.

Around the year 2005 when Herb Sutter wrote his excellent article, you'd find companies running applications on clustered multiprocessor servers (often no more than two to three, just in case one of them crashed). Support for concurrency in programming languages was available but limited and considered black magic by many mere mortal programmers. Herb Sutter predicted in his article that "programming languages ... will increasingly be forced to deal well with concurrency."

Let's see what changed in the decade since! Fast-forward to today, and you find applications running on large numbers of servers in the cloud, integrating many systems across many data centers. The ever-increasing demands of end users push the requirements of performance and stability of the systems that you build.

So where are those new concurrency features? Support for concurrency in most programming languages, especially on the JVM, has hardly changed. Although the implementation details of concurrency APIs have definitely improved, you still have to work with low-level constructs like threads and locks, which are notoriously difficult to work with.

Next to scaling up (increasing resources; for example, CPUs on existing servers), *scaling out* refers to dynamically adding more servers to a cluster. Since the '90s, nothing much has changed in how programming languages support networking, either. Many technologies still essentially use RPC (remote procedure calls) to communicate over the network.

In the meantime, advances in cloud computing services and multicore CPU architecture have made computing resources ever more abundant.

PaaS (Platform as a Service) offerings have simplified provisioning and deployment of very large distributed applications, once the domain of only the largest players in the IT industry. Cloud services like AWS EC2 (Amazon Web Services Elastic Compute Cloud) and Google Compute Engine give you the ability to literally spin up thousands of servers in minutes, while tools like Docker, Puppet, Ansible, and many others make it easier to manage and package applications on virtual servers.

The number of CPU cores in devices is also ever-increasing: even mobile phones and tablets have multiple CPU cores today.

But that doesn't mean that you can afford to throw any number of resources at any problem. In the end, everything is about cost and efficiency. So it's all about effectively scaling applications, or in other words, getting bang for your buck. Just as you'd never use a sorting algorithm with exponential time complexity, it makes sense to think about the cost of scaling.

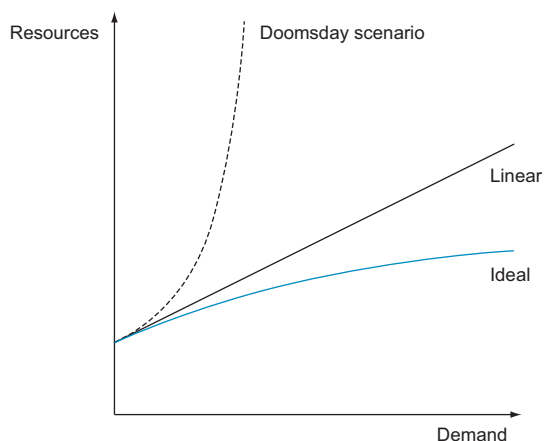


Figure 1.1 Demand against resources

You should have two expectations when scaling your application:

- The ability to handle any increase of demand with finite resources is unrealistic, so ideally you'd want the required increase of resources to be growing slowly when demand grows, linear or better. Figure 1.1 shows the relationship between demand and number of required resources.
- If resources have to be increased, ideally you'd like the complexity of the application to stay the same or increase slowly. (Remember the good ol' free lunch when no added complexity was required for a faster application!) Figure 1.2 shows the relationship between number of resources and complexity.

Both the number and complexity of resources contribute to the total cost of scaling.

We're leaving a lot of factors out of this back-of-the-envelope calculation, but it's easy to see that both of these rates have a big impact on the total cost of scaling.

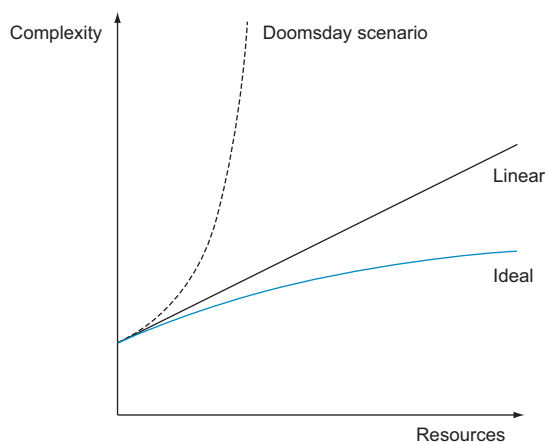


Figure 1.2 Complexity against resources

One doomsday scenario is where you'd need to pay increasingly more for more underutilized resources. Another nightmare scenario is where the complexity of the application shoots through the roof when more resources are added.

This leads to two goals: complexity has to stay as low as possible, and resources must be used efficiently while you scale the application.

Can you use the common tools of today (threads and RPC) to satisfy these two goals? Scaling out with RPC and scaling up with low-level threading aren't good ideas. RPC pretends that a call over the network is no different from a local method call. Every RPC call needs to block the current thread and wait for a response from the network for the local method call abstraction to work, which can be costly. This impedes the goal of using resources efficiently.

Another problem with this approach is that you need to know exactly where you scale up or scale out. Multithreaded programming and RPC-based network programming are like apples and pears: they run in different contexts, using different semantics and running on different levels of abstraction. You end up hardcoding which parts of your application are using threads for scaling up and which parts are using RPC for scaling out.

Complexity increases significantly the moment you hardcode methods that work on different levels of abstraction. Quick—what's simpler, coding with two entangled programming constructs (RPC and threads), or using just one programming construct? This multipronged approach to scaling applications is more complicated than necessary to flexibly adapt to changes in demand.

Spinning up thousands of servers is simple today, but as you'll see in this first chapter, the same can't be said for programming them.

1.1 *What is Akka?*

In this book we'll show how the Akka toolkit, an open source project built by Lightbend, provides a simpler, single programming model—one way of coding for concurrent and distributed applications—the *actor programming model*. Actors are (fitting for our industry) nothing new at all, in and of themselves. It's the way that actors are provided in Akka to scale applications both up and out on the JVM that's unique. As you'll see, Akka uses resources efficiently and makes it possible to keep the complexity relatively low while an application scales.

Akka's primary goal is to make it simpler to build applications that are deployed in the cloud or run on devices with many cores and that efficiently leverage the full capacity of the computing power available. It's a toolkit that provides an actor programming model, runtime, and required supporting tools for building scalable applications.

1.2 *Actors: a quick overview*

First off, Akka is centered on actors. Most of the components in Akka provide support in some way for using actors, be it for configuring actors, connecting actors to the network, scheduling actors, or building a cluster out of actors. What makes Akka unique

is how effortlessly it provides support and additional tooling for building actor-based applications, so that you can focus on thinking and programming in actors.

Briefly, actors are a lot like message queues without the configuration and message broker installation overhead. They're like programmable message queues shrunk to microsize—you can easily create thousands, even millions of them. They don't "do" anything unless they're sent a message.

Messages are simple data structures that can't be changed after they've been created, or in a single word, they're *immutable*.

Actors can receive messages one at a time and execute some behavior whenever a message is received. Unlike queues, they can also send messages (to other actors).

Everything an actor does is executed asynchronously. Simply put, you can send a message to an actor without waiting for a response. Actors aren't like threads, but messages sent to them are pushed through on a thread at some point in time. How actors are connected to threads is configurable, as you'll see later; for now it's good to know that this is not a hardwired relationship.

We'll get a lot deeper into exactly what an actor is. For now the most important aspect of actors is that you build applications by sending and receiving messages. A message could be processed locally on some available thread, or remotely on another server. Exactly where the message is processed and where the actor lives are things you can decide later, which is very different compared to hardcoding threads and RPC-style networking. Actors make it easy to build your application out of small parts that resemble networked services, only shrunk to microsize in footprint and administrative overhead.

The Reactive Manifesto

The Reactive Manifesto (<http://www.reactivemanifesto.org/>) is an initiative to push for the design of systems that are more robust, more resilient, more flexible, and better positioned to meet modern demands. The Akka team has been involved in writing the Reactive Manifesto from the beginning, and Akka is a product of the ideas that are expressed in this manifesto.

In short, efficient resource usage and an opportunity for applications to automatically scale (also called *elasticity*) is the driver for a big part of the manifesto:

- Blocking I/O limits opportunities for parallelism, so nonblocking I/O is preferred.
- Synchronous interaction limits opportunities for parallelism, so asynchronous interaction is preferred.
- Polling reduces opportunity to use fewer resources, so an event-driven style is preferred.
- If one node can bring down all other nodes, that's a waste of resources. So you need isolation of errors (resilience) to avoid losing all your work.

(continued)

- Systems need to be elastic: If there's less demand, you want to use fewer resources. If there's more demand, use more resources, but never more than required.

Complexity is a big part of cost, so if you can't easily test it, change it, or program it, you've got a big problem.

1.3 *Two approaches to scaling: setting up our example*

In the rest of this chapter, we'll look at a business chat application and the challenges faced when it has to scale to a large number of servers (and handle millions of simultaneous events). We'll look at what we'll call the *traditional approach*, a method that you're probably familiar with for building such an application (using threads and locks, RPC, and the like) and compare it to Akka's approach.

The traditional approach starts with a simple in-memory application, which turns into an application that relies completely on a database for both concurrency and mutating state. Once the application needs to be more interactive, we'll have no choice but to poll this database. When more network services are added, we'll show that the combination of working with the database and the RPC-based network increases complexity significantly. We'll also show that isolating failure in this application becomes very hard as we go along. We think that you'll recognize a lot of this.

We'll then look at how the actor programming model simplifies the application, and how Akka makes it possible to write the application once and scale it to any demand (thereby handling concurrency issues on any scale needed). Table 1.1 highlights the differences between the two approaches. Some of the items will become clear in the next sections, but it's good to keep this overview in mind.

Table 1.1 Differences between approaches

Objective	Traditional method	Akka method
Scaling	Use a mix of threads, shared mutable state in a database (Create, Insert, Update, Delete), and web service RPC calls for scaling.	Actors send and receive messages. No shared mutable state. Immutable log of events.
Providing interactive information	Poll for current information.	Event-driven: push when the event occurs.
Scaling out on the network	Synchronous RPC, blocking I/O.	Asynchronous messaging, nonblocking I/O.
Handling failures	Handle all exceptions; only continue if everything works.	Let it crash. Isolate failure, and continue without failing parts.

Imagine that we have plans to conquer the world with a state-of-the-art chat application that will revolutionize the online collaboration space. It's focused on business

users where teams can easily find each other and work together. We have tons of ideas on how this interactive application can connect to project management tools and integrate with existing communication services.

In good Lean Startup spirit, we start with an MVP (minimal viable product) of the chat application to learn as much as possible from our prospective users about what they need. If this ever takes off, we could potentially have millions of users (who doesn't chat, or work together in teams?). And we know that there are two forces that can slow our progress to a grinding halt:

- *Complexity*—The application becomes too complex to add any new features. Even the simplest change takes a huge amount of effort, and it becomes harder and harder to test properly; what will fail this time?
- *Inflexibility*—The application isn't adaptive; with every big jump in number of users, it has to be rewritten from scratch. This rewrite takes a long time and is complex. While we have more users than we can handle, we're split between keeping the existing application running and rewriting it to support more users.

We've been building applications for a while and choose to build it the way we have in the past, taking the traditional approach, using low-level threads and locks, RPC, blocking I/O, and, first on the menu in the next section, mutating state in a database.

1.4 Traditional scaling

We start on one server. We set out to build the first version of the chat application, and come up with a data model design, shown in figure 1.3. For now we'll just keep these objects in memory.

A Team is a group of Users, and many Users can be part of some Conversation. Conversations are collections of messages. So far, so good.

We flesh out the behavior of the application and build a web-based user interface. We're at the point where we can show the application to prospective users and give demos. The code is simple and easy to manage. But so far this application only runs in memory, so whenever it's restarted, all Conversations are lost. It can also only run on one server at this point. Our web app UI built with [insert shiny new JavaScript library] is so impressive that stakeholders want to immediately go live with it, even though we repeatedly warn that it's just for demo purposes! Time to move to more servers and set up a production environment.

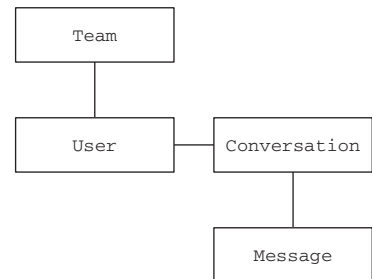


Figure 1.3 Data model design

1.4.1 Traditional scaling and durability: move everything to the database

We decide to add a database to the equation. We have plans to run the web application on two front-end web servers for availability, with a load balancer in front of it. Figure 1.4 shows the new setup.

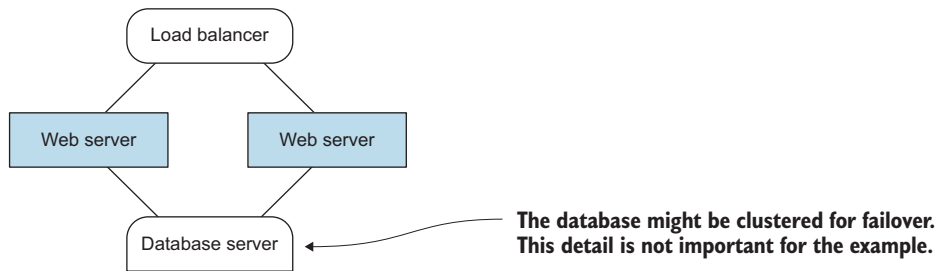


Figure 1.4 Load balancer/failover

The code is becoming more complex because now we can't just work with in-memory objects anymore; how would we keep the objects consistent on the two servers? Someone on our team shouts *"We need to go stateless!"* and we remove all feature-rich objects and replace them with database code.

The state of the objects doesn't simply reside in memory on the web servers anymore, which means the methods on the objects can't work on the state directly; essentially, all important logic moves to database statements. The change is shown in figure 1.5.

This move to statelessness leads to the decision to replace the objects with some database access abstraction. For the purpose of this example, it's irrelevant which one;

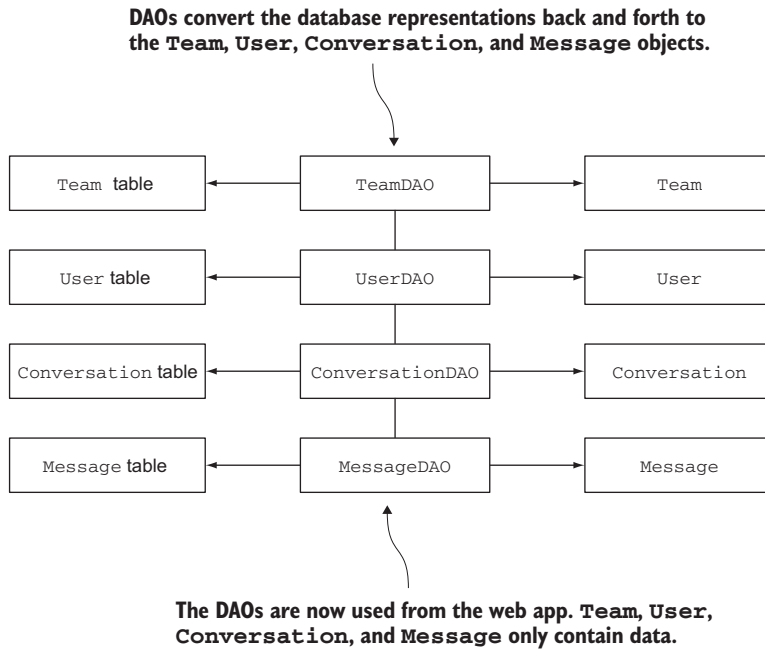


Figure 1.5 Data access objects

in this case, we're feeling a bit retro and use DAOs (data access objects, which execute database statements).

A lot of things change:

- We don't have the same guarantees anymore that we had before when we, for instance, called a method on the `Conversation` to add a `Message`. Before, we were guaranteed that `addMessage` would never fail, since it was a simple operation on an in-memory list (barring the exceptional case that the JVM runs out of memory). Now, the database might return an error at any `addMessage` call. The insert might fail, or the database might not be available at that exact moment because the database server crashes or because there's a problem with the network.
- The in-memory version had a sprinkling of locks to make sure that the data wouldn't get corrupted by concurrent users. Now that we're using "Database X," we'll have to find out how to handle that problem, and make sure that we don't end up with duplicate records or other inconsistent data. We have to find out how to do exactly that with the Database X library. Every simple method call to an object effectively becomes a database operation, of which some have to work in concert. Starting a `Conversation`, for instance, at least needs both an insert of a row in the `Conversation` and the message table.
- The in-memory version was easy to test, and unit tests ran fast. Now, we run Database X locally for the tests, and we add some database test utilities to isolate tests. Unit tests run a lot slower now. But we tell ourselves, "At least we're testing those Database X operations too," which were not as intuitive as we expected—very different from the previous databases we worked with.

We probably run into performance problems when we're porting the in-memory code directly to database calls, since every call now has network overhead. So we design specific database structures to optimize query performance, which are specific to our choice of database (SQL or NoSQL, it doesn't matter). The objects are now a sad anemic shadow of their former selves, merely holding data; all the interesting code has moved to the DAOs and the components of our web application. The saddest part of this is that we can hardly reuse any of the code that we had before; the structure of the code has completely changed.

The "controllers" in our web application combine DAO methods to achieve the changes in the data (`findConversation`, `insertMessage`, and so on). This combination of methods results in an interaction with the database that we can't easily predict; the controllers are free to combine the database operations in any way, as in figure 1.6.

The figure shows one of the possible flows through the code, for adding a `Message` to a `Conversation`. You can imagine that there are numerous variations of database access flows through the use of the DAOs. Allowing any party to mutate or query records at any point in time can lead to performance problems that we can't predict, like deadlocks and other issues. It's exactly the kind of complexity we want to avoid.

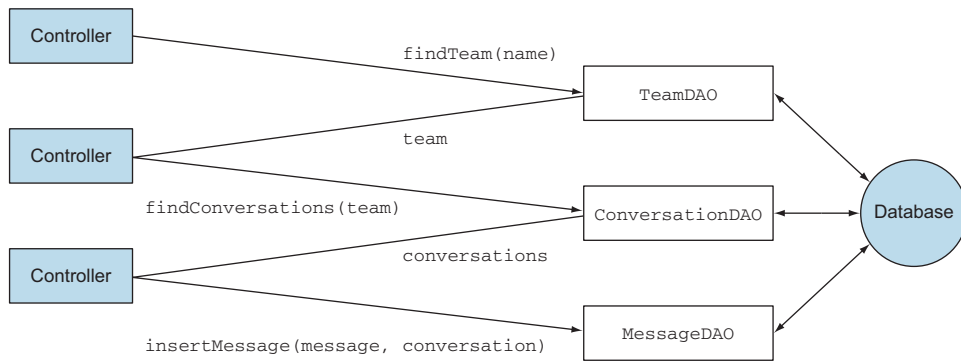


Figure 1.6 DAO interaction

The database calls are essentially RPC, and almost all standard database drivers (say, JDBC) use blocking I/O. So we’re already in the state that we described before, using threads and RPC together. The memory locks that are used to synchronize threads and the database locks to protect mutation of table records are really not the same thing, and we’ll have to take great care to combine them. We went from one to two interwoven programming models.

We just did our first rewrite of the application, and it took a lot longer than expected.

THIS IS A DRAMATIZATION The traditional approach to build the team chat app goes sour in a catastrophic way. Although exaggerated, you’ve probably seen projects run into at least some of these problems (we definitely have seen similar cases first-hand). To quote Dean Wampler from his presentation “Reactive Design, Languages, and Paradigms” (<https://deanwampler.github.io/polyglotprogramming/papers/>):

In reality, good people can make almost any approach work, even if the approach is suboptimal.

So is this example project impossible to complete with the traditional approach? No, but it’s definitely suboptimal. It will be very hard to keep complexity low and flexibility high while the application scales.

1.4.2 Traditional scaling and interactive use: polling

We run in this configuration for a while and the users are increasing. The web application servers aren’t using a lot of resources; most are spent in (de-)serialization of requests and responses. Most of the processing time is spent in the database. The code on the web server is mostly waiting for a response from the database driver.

We want to build more interactive features now that we have the basics covered. Users are used to Facebook and Twitter and want to be notified whenever their name is mentioned in a team conversation, so they can chime in.

We want to build a `Mentions` component that parses every message that's written and adds the mentioned contacts to a notification table, which is polled from the web application to notify mentioned users.

The web application now also polls other information more often to more quickly reflect changes to users, because we want to give them a true interactive experience.

We don't want to slow down the conversations by adding database code directly to the application, so we add a message queue. Every message written is sent to it asynchronously, and a separate process receives messages from the queue, looks up the users, and writes a record in a notifications table.

The database is really getting hammered at this point. We find out that the automated polling of the database together with the `Mentions` component are causing performance problems with the database. We separate out the `Mentions` component as a service and give it its own database, which contains the notifications table and a copy of the users table, kept up to date with a database synchronization job, as shown in figure 1.7.

Not only has the complexity increased again, it's becoming more difficult to add new interactive features. Polling the database wasn't such a great idea for this kind of application, but there are no other real options, because all the logic is right there in the DAOs, and Database X can't "push" anything into the web server.

We've also added more complexity to the application by adding a message queue, which will have to be installed and configured, and code will have to get deployed. The message queue has its own semantics and context to work in; it's not the same as the database RPC calls, or as the in-memory threading code. Fusing all this code together responsibly will be, once again, more complex.

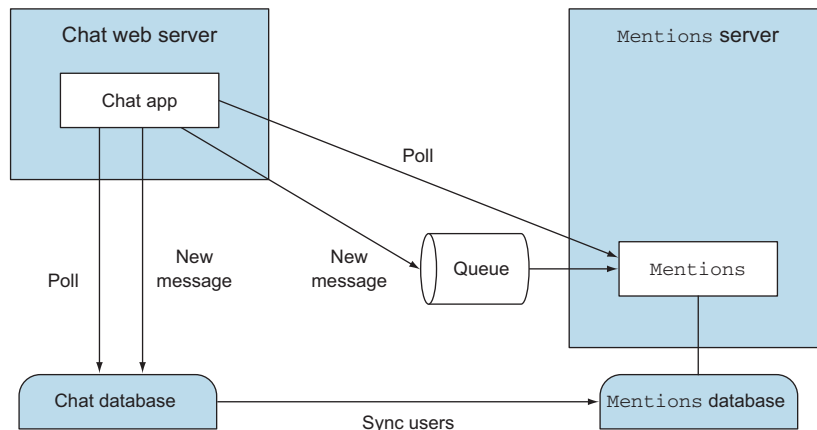


Figure 1.7 Service component

1.4.3 Traditional scaling and interactive use: polling

Users start to give feedback that they would love a way to find contacts with *typeahead* (the application gives suggestions while the user types part of a contact's name) and automatically receive suggestions for teams and current conversations based on their recent email conversations. We build a `TeamFinder` object that calls out to several web services like Google Contacts API and Microsoft Outlook.com API. We build web service clients for these, and incorporate the finding of contacts, as in figure 1.8.

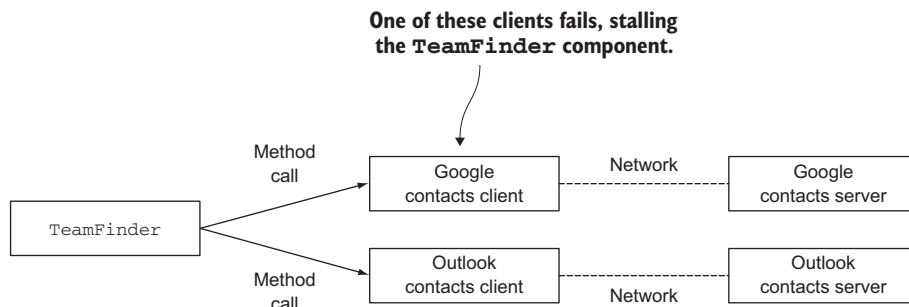


Figure 1.8 Team finder

We find out that one of the services fails often and in the worst possible way—we get long timeouts, or traffic has slowed down to only a few bytes per minute. And because the web services are accessed one after the other, waiting for a response, the lookup fails after a long time even though many valid suggestions could have been made to the user from the service that worked just fine.

Even worse, though we collected our database methods in DAOs and the contacts lookup in a `TeamFinder` object, the controllers are calling these methods like any other. This means that sometimes a user lookup ends up right between two database methods, keeping connections open longer than we want, eating up database resources. If the `TeamFinder` fails, everything else that's part of the same flow in the application fails as well. The controller will throw an exception and won't be able to continue. How do we safely separate the `TeamFinder` from the rest of the code?

It's time for another rewrite, and it doesn't look like the complexity is improving. In fact, we're now using four programming models: one for the in-memory threads, one for the database operations, one for the Mentions message queue, and one for the contacts web services.

How do we move from 3 servers to, say, 10, and then to 100 servers, if this should be required? It's obvious that this approach doesn't scale well: we need to change direction with every new challenge.

In the next section, you'll find out if there's a design strategy that doesn't require us to change direction with every new challenge.

1.5 Scaling with Akka

Let's see if it's possible to deliver on the promise to use only actors to meet the scaling requirements of the application. Since it's probably still unclear to you what actors are, exactly, we'll use objects and actors interchangeably and focus on the conceptual difference between this approach and the traditional approach.

Table 1.2 shows this difference in approaches.

Table 1.2 Actors compared to the traditional approach

Goal	Traditional approach	Akka approach (actors)
Make conversation data durable, even if the application restarts or crashes.	Rewrite code into DAOs. Use the database as one big shared mutable state, where all parties create, update, insert, and query the data.	Continue to use in-memory state. Changes to the state are sent as messages to a log. This log is only reread if the application restarts.
Provide interactive features (Mentions).	Poll the database. Polling uses a lot of resources even if there's no change in the data.	Push events to interested parties. The objects notify interested parties only when there's a significant event, reducing overhead.
Decoupling of services; the Mentions and chat features shouldn't be interfering with each other.	Add a message queue for asynchronous processing.	No need to add a message queue; actors are asynchronous by definition. No extra complexity; you're familiar with sending and receiving messages.
Prevent failure of the total system when critical services fail or behave outside of specified performance parameters for any given time.	Try to prevent any error from happening by predicting all failure scenarios and catching exceptions for these scenarios.	Messages are sent asynchronously; if a message isn't handled by a crashed component, it has no impact on the stability of the other components.

It would be great if we could write the application code once, and then scale it any way we like. We want to avoid radically changing the application's main objects; for example, how we had to replace all logic in the in-memory objects with DAOs in section 1.4.1.

The first challenge we wanted to solve was to safekeep conversation data. Coding directly to the database moved us away from one simple in-memory model. Methods that were once simple turned into database RPC commands, leaving us with a mixed programming model. We have to find another way to make sure that the conversations aren't lost, while keeping things simple.

1.5.1 Scaling with Akka and durability: sending and receiving messages

Let's first solve the initial problem of just making Conversations durable. The application objects must save Conversations in some way. The Conversations must at least be recovered when the application restarts.

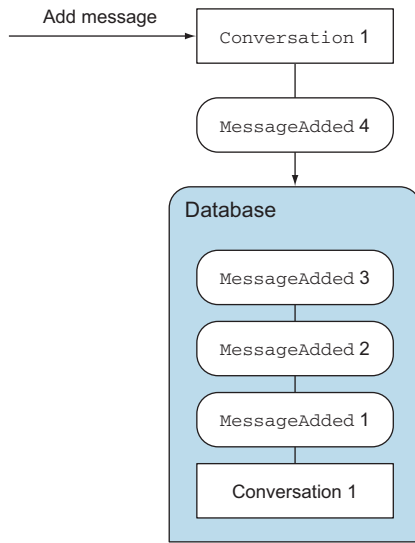


Figure 1.9 Persist conversations

Figure 1.9 shows how a `Conversation` sends a `MessageAdded` to the database log for every message that's added in-memory.

The `Conversation` can be rebuilt from these objects stored in the database whenever the web server (re)-starts, as shown in figure 1.10.

Exactly how this all works is something we'll discuss later. But as you can see, we only use the database to recover the messages in the conversation. We don't use it to express our code in database operations. The `Conversation` actor sends messages to

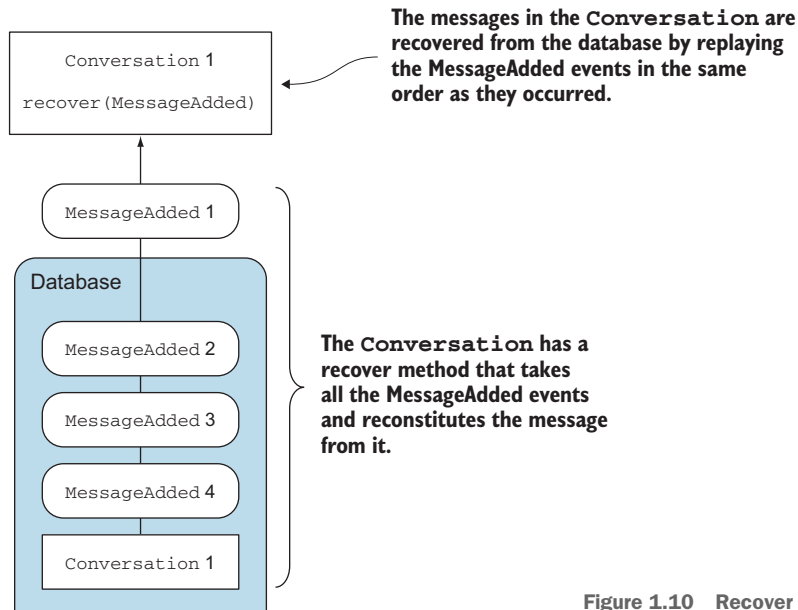


Figure 1.10 Recover conversations

the log, and receives them again on startup. We don't have to learn anything new; it's just sending and receiving messages.

CHANGES KEPT AS A SEQUENCE OF EVENTS

All changes are kept as a sequence of events, in this case `MessageAdded` events. The current state of the `Conversation` can be rebuilt by replaying the events that occurred to the in-memory `Conversation`, so it can continue where it left off. This type of database is often called a *journal*, and the technique is known as *event sourcing*. There's more to event sourcing, but for now this definition will do.

What's important to note here is that the journal has become a uniform service. All it needs to do is store all events in sequence, and make it possible to retrieve the events in the same sequence as they were written to the journal. There are some details that we'll ignore for now, like serialization—if you can't wait, go look at chapter 15 on actor persistence.

SPREADING OUT THE DATA: SHARDING CONVERSATIONS

The next problem is that we're still putting all our eggs in one server. The server restarts, reads all conversations in memory, and continues to operate. The main reason for going stateless in the traditional approach is that it's hard to imagine how we would keep the conversations consistent across many servers. And what would happen if there were too many conversations to fit on one server?

A solution for this is to divide the conversations over the servers in a predictable way or to keep track of where every conversation lives. This is called *sharding* or *partitioning*. Figure 1.11 shows some conversations in shards across two servers.

We can keep using the simple in-memory model of `Conversations` if we have a generic event-sourced journal and a way to indicate how `Conversations` should be partitioned. Many details about these two capabilities will be covered in chapter 15. For now, we'll assume that we can simply use these services.

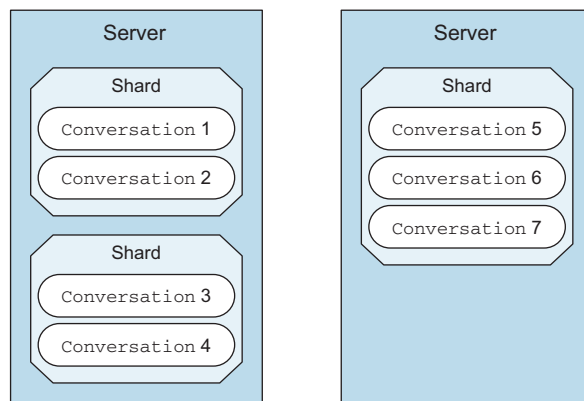


Figure 1.11 Sharding

1.5.2 *Scaling with Akka and interactive use: push messages*

Instead of polling the database for every user of the web application, we could find out if there's a way to notify the user of an important change (an event) by directly sending messages to the user's web browser.

The application can also send event messages internally as a signal to execute particular tasks. Every object in the application will send an event when something interesting occurs. Other objects in the application can decide if an event is interesting and take action on it, as in figure 1.12.

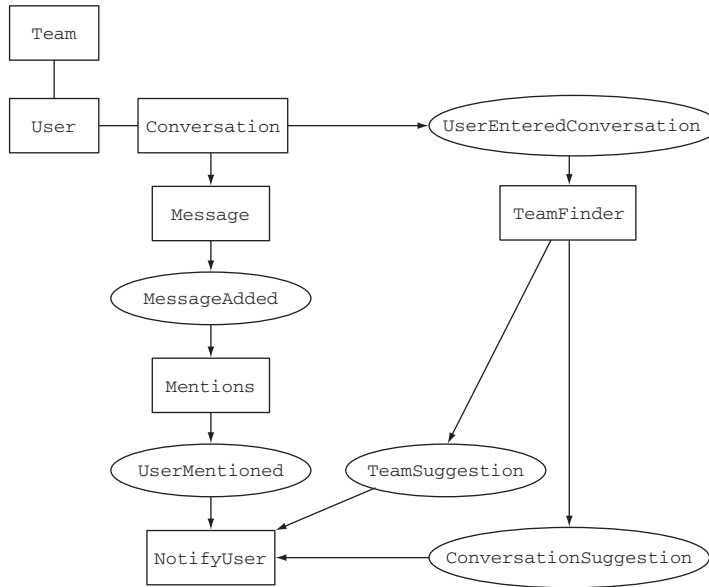


Figure 1.12 Events

The events (depicted as ellipses) decouple the system where there used to be undesired coupling between the components. The Conversation only publishes that it added a Message and continues its work. Events are sent through a publish-subscribe mechanism, instead of the components communicating with each other directly. An event will eventually get to the subscribers, in this case to the Mentions component. It's important to note that, once again, we can model the solution to this problem by simply sending and receiving messages.

1.5.3 *Scaling with Akka and failure: asynchronous decoupling*

It's preferable that users be able to continue to have Conversations even if the Mentions component has crashed. The same goes for the TeamFinder component: existing conversations should be able to continue. Conversations can continue to publish events while subscribers, like the Mentions component and the TeamFinder object, crash and restart.

The `NotifyUser` component could keep track of connected web browsers and send `UserMentioned` messages directly to the browser when they occur, relieving the application from polling.

This event-driven approach has a couple of advantages:

- It minimizes direct dependencies between components. The conversation *doesn't know about the* `Mentions` *object and could not care less what happens with the event*. The conversation can continue to operate when the `Mentions` object crashes.
- The components of the application are loosely coupled in time. It doesn't matter if the `Mentions` object gets the events a little later, as long as it gets the events eventually.
- The components are decoupled in terms of location. The `Conversation` and `Mentions` object can reside on different servers; the events are just messages that can be transmitted over the network.

The event-driven approach solves the polling problem with the `Mentions` object, as well as the direct coupling with the `TeamFinder` object. In chapter 5 on futures, we'll look at some better ways to communicate with web services than sequentially waiting for every response. It's important to note that, once again, we can model the solution to this problem by simply sending and receiving messages.

1.5.4 The Akka approach: sending and receiving messages

Let's recap what we've changed so far: `Conversations` are now stateful in-memory objects (actors), storing their internal state, recovering from events, partitioned across servers, sending and receiving messages.

You've seen how communicating between objects with messages instead of calling methods directly is a winning design strategy.

A core requirement is that messages are sent and received in order, one at a time to every actor, when one event is dependent on the next, because otherwise we'd get unexpected results. This requires that the `Conversation` keeps its own messages secret from any other component. The order can never be kept if any other component can interact with the messages.

It shouldn't matter if we send a message locally on one server or remotely to another. So we need some service that takes care of sending the messages to actors on other servers if necessary. It will also need to keep track of where actors live and be able to provide references so other servers can communicate with the actors. This is one of the things that Akka does for you, as you'll soon see. Chapter 6 discusses the basics of distributed Akka applications, and chapter 13 discusses clustered Akka applications (in short, groups of distributed actors).

The `Conversation` doesn't care what happens with the `Mentions` component, but on the application level we need to know when the `Mentions` component doesn't work anymore to show users that it's temporarily offline, among other things. So we need some kind of monitoring of actors, and we need to make it possible to reboot these if

The supervisor watches the objects and takes actions when they crash.

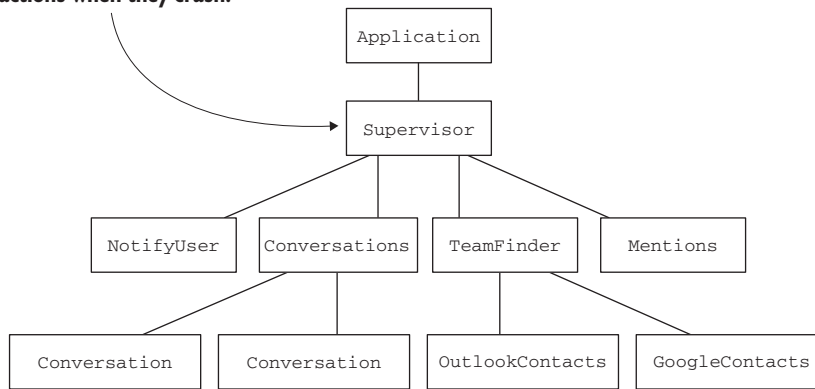


Figure 1.13 High-level structure

necessary. This monitoring should work across servers as well as locally on one server, so it will also have to use sending and receiving messages. A possible high-level structure for the application is shown in figure 1.13.

The supervisor watches over the components and takes action when they crash. It can, for example, decide to continue running when the Mentions component or the TeamFinder doesn't work. If both Conversations and NotifyUser stop working completely, the supervisor could decide to restart completely or stop the application, since there's no reason to continue. A component can send a message to the supervisor when it fails, and the supervisor can send a message to a component to stop, or try to restart. As you'll see, this is conceptually how Akka provides error recovery, which is discussed in chapter 4 on fault tolerance.

In the next section, we'll first talk about actors in general, and then talk about Akka actors.

1.6 **Actors: one programming model to rule up and out**

Most general-purpose programming languages are written in sequence (Scala and Java being no exception to the rule). A concurrent programming model is required to bridge the gap between sequential definition and parallel execution.

Whereas parallelization is all about executing processes simultaneously, concurrency concerns itself with defining processes that *can* function simultaneously, or *can* overlap in time, but don't necessarily *need* to run simultaneously. A concurrent system is not by definition a parallel system. Concurrent processes can, for example, be executed on one CPU through the use of time slicing, where every process gets a certain amount of time to run on the CPU, one after another.

The JVM has a standard concurrent programming model (see figure 1.14), where, roughly speaking, processes are expressed in objects and methods, which are executed on threads. Threads might be executed on many CPUs in parallel, or using

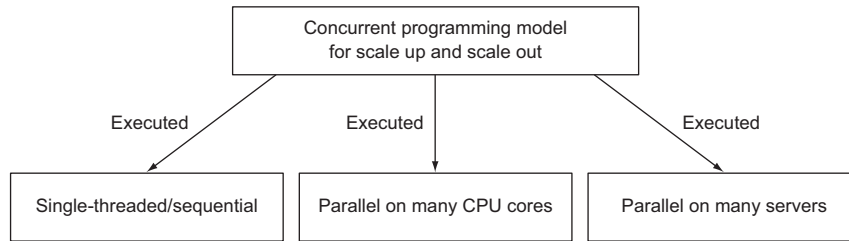


Figure 1.14 Concurrent programming model

some sharing mechanism like time slicing on one CPU. As we discussed earlier, threads can't be applied directly to scaling out, only to scaling up.

The concurrent programming model that we're after should function for one CPU or many, one server or many servers. The actor model chooses the abstraction of sending and receiving messages to decouple from the number of threads or the number of servers that are being used.

1.6.1 An asynchronous model

If we want the application to scale to many servers, there's an important requirement for the programming model: it will have to be *asynchronous*, allowing components to continue working while others haven't responded yet, as in the chat application (see figure 1.15).

The figure shows a possible configuration of the chat application, scaled to five servers. The supervisor has the responsibility to create and monitor the rest of the application. The supervisor now has to communicate over the network, which might fail, and every server could possibly crash as well. If the supervisor used synchronous

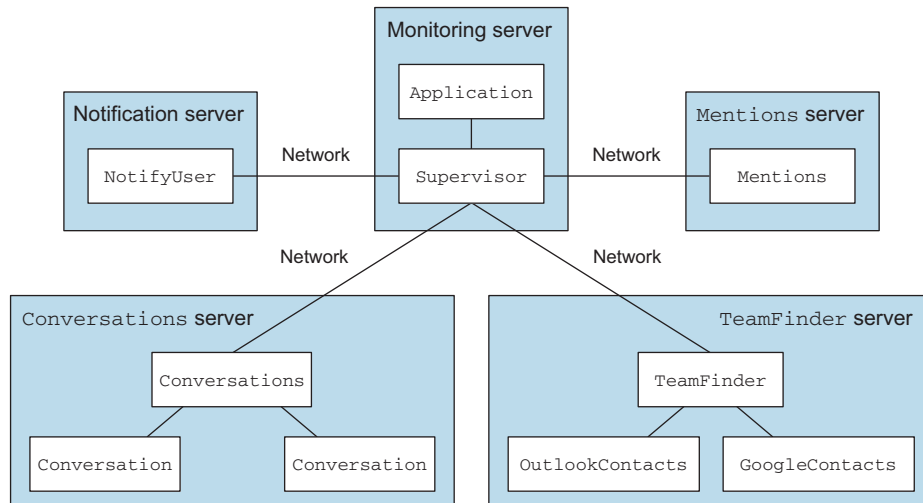


Figure 1.15 Scaled out

communication, waiting for every response of every component, we could get in the problematic situation where one of the components doesn't respond, blocking all other calls from happening. What would happen, for instance, if the conversations server is restarting and not responding to the network interface yet, while the supervisor wants to send out messages to all components?

1.6.2 Actor operations

Actors are the primary building blocks in the actor model. All the components in the example application are actors, shown in figure 1.16. An actor is a lightweight process that has only four core operations: create, send, become, and supervise. All of these operations are asynchronous.

THE ACTOR MODEL—NOT NEW The actor model is not new and has actually been around for quite a while; the idea was introduced in 1973 by Carl Hewitt, Peter Bishop, and Richard Steiger. The Erlang language and its OTP middleware libraries, developed by Ericsson around 1986, support the actor model and have been used to build massively scalable systems with requirements for high availability. An example of the success of Erlang is the AXD 301 switch product, which achieves a reliability of 99.9999999%, also known as *nine nines* reliability. The actor model implementation in Akka differs in a couple of details from the Erlang implementation, but has definitely been heavily influenced by Erlang, and shares a lot of its concepts.

SEND

An actor can only communicate with another actor by sending it messages. This takes *encapsulation* to the next level. In objects we can specify which methods can be publicly called and which state is accessible from the outside. Actors don't allow any access to internal state, for example, the list of messages in a conversation. Actors can't share mutable state; they can't, for instance, point to a shared list of conversation messages and change the conversation in parallel at any point in time.

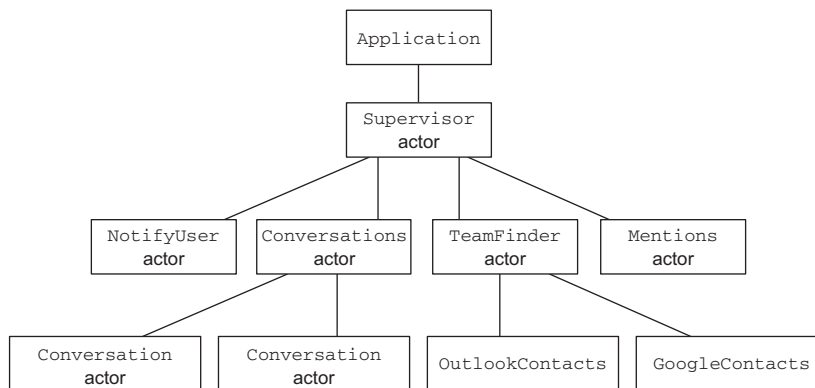


Figure 1.16 Components

The Conversation actor can't simply call a method on any other actor, since that could lead to sharing mutable state. It has to send it a message. Sending messages is always asynchronous, in what is called a *fire and forget* style. If it's important to know that another actor received the message, then the receiving actor should just send back an acknowledgement message of some kind.

The Conversation actor doesn't have to wait and see what happens with a message to the Mentions actor; it can send off a message and continue its work. Asynchronous messaging helps in the chat application to decouple the components; this was one of the reasons why we wanted to use a message queue for the Mentions object, which is now unnecessary.

The messages need to be immutable, meaning that they can't be changed once they're created. This makes it impossible for two actors to change the same message by mistake, which could result in unexpected behavior.

WHAT, NO TYPE SAFETY? Actors can receive any message, and you can send any message you want to an actor (it just might not process the message). This basically means that type checking of the messages that are sent and received is limited. That might come as a surprise, since Scala is a statically typed language and a high level of type safety has many benefits. This flexibility is both a cost (less is known about actors' type correctness at runtime) and a benefit (how would static types be enforced over a network of remote systems?). The last word hasn't been said on this, and the Akka team is researching how to define a more type-safe version of actors, which we might see details of in a next version of Akka. Stay tuned.

So what do we do when a user wants to edit a message in a Conversation? We could send an `EditMessage` message to the conversation. The `EditMessage` contains a modified copy of the message, instead of updating the message in place in a shared messages list. The Conversation actor receives the `EditMessage` and replaces the existing message with the new copy.

Immutability is an absolute necessity when it comes to concurrency and is another restriction that makes life simpler, because there are fewer moving parts to manage.

The order of sent messages is kept between a sending and receiving actor. An actor receives messages one at a time. Imagine that a user edits a message many times; it would make sense that the user eventually sees the result of the final edit of the message. The order of messages is only guaranteed per sending actor, so if many users edit the same message in a conversation, the final result can vary depending on how the messages are interleaved over time.

CREATE

An actor can create other actors. Figure 1.17 shows how the Supervisor actor creates a Conversations actor. As you can see, this automatically creates a hierarchy of actors. The chat application first creates the Supervisor actor, which in turn creates all

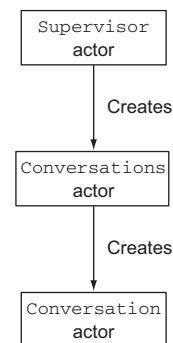


Figure 1.17 Create

other actors in the application. The Conversations actor recovers all Conversations from the journal. It then creates a Conversation actor for every Conversation, which in turn recovers itself from the journal.

BECOME

State machines are a great tool for making sure that a system only executes particular actions when it's in a specific state.

Actors receive messages one at a time, which is a convenient property for implementing state machines. An actor can change how it handles incoming messages by swapping out its behavior.

Imagine that users want to be able to close a Conversation. The Conversation starts out in a started state and becomes closed when a CloseConversation is received. Any message that's sent to the closed Conversation could be ignored. The Conversation swaps its behavior from adding messages to itself to ignoring all messages.

SUPERVISE

An actor needs to supervise the actors that it creates. The supervisor in the chat application can keep track of what's happening to the main components, as shown in figure 1.18.

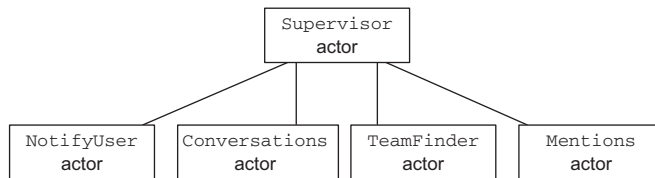


Figure 1.18 Supervise

The Supervisor decides what should happen when components fail in the system. It could, for example, decide that the chat application continues when the Mentions component and Notify actor have crashed, since they're not critical components. The Supervisor gets notified with special messages that indicate which actor has crashed, and for what reason. The Supervisor can decide to restart an actor or take the actor out of service.

Any actor can be a supervisor, but only for actors that it creates itself. In figure 1.19 the TeamFinder actor supervises the two connectors for looking up contacts. In this case it could decide to take the OutlookContacts actor out of service because it failed too often. The TeamFinder will then continue looking up contacts from Google only.

Outlook is taken out of service. It failed too often.

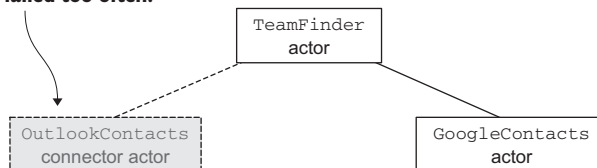


Figure 1.19 TeamFinder supervising contacts actors

Actors: decoupled on three axes

Another way to look at actors is how they're decoupled, on three axes, for the purpose of scaling:

- Space/Location
- Time
- Interface

Decoupling on exactly these three axes is important because this is exactly the flexibility that's required for scaling. Actors might run at the same time if there are enough CPUs, or might run one after the other if not. Actors might be co-located, or far apart, and in a failure scenario actors might receive messages that they can't handle.

- *Space*—An actor gives no guarantee and has no expectation about where another actor is located.
- *Time*—An actor gives no guarantee and has no expectation about when its work will be done.
- *Interface*—An actor has no defined interface. An actor has no expectation about which messages other components can understand. Nothing is shared between actors; actors never point to or use a shared piece of information that changes in place. Information is passed in messages.

Coupling components in location, time, and interface is the biggest impediment to building applications that can recover from failure and scale according to demand. A system built out of components that are coupled on all three axes can only exist on one runtime and will fail completely if one of its components fails.

Now that we've looked at the operations that an actor can perform, let's look at how Akka supports actors and what's required to make them actually process messages.

1.7 Akka actors

So far we've discussed the actor programming model from a conceptual perspective and why you would want to use it. Let's see how Akka implements the actor model and get closer to where the rubber meets the road. We'll look at how everything connects together—which Akka components do what. In the next section, we'll start with the details of actor creation.

1.7.1 ActorSystem

The first thing we'll look at is how actors are created. Actors can create other actors, but who creates the first one? See figure 1.20.

The chat application's first actor is the `Supervisor` actor. All the actors shown in figure 1.20 are part of the same application. How do we make actors part of one bigger whole, one bigger picture? The answer that Akka provides for this is the `ActorSystem`. The first thing that every Akka application does is create an `ActorSystem`. The actor system can create so called top-level actors, and it's a common pattern to

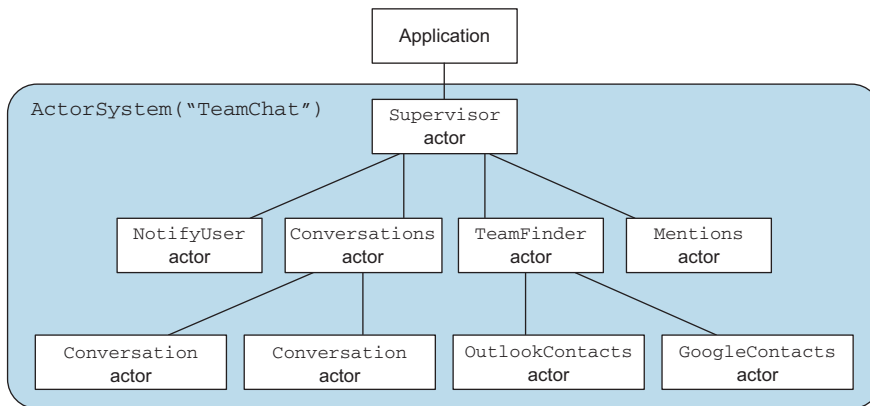


Figure 1.20 TeamChatActorSystem

create only one top-level actor for all actors in the application—in our case, the Supervisor actor that monitors everything.

We’ve touched on the fact that we’ll need support capabilities for actors, like remoting and a journal for durability. The ActorSystem is also the nexus for these support capabilities. Most capabilities are provided as *Akka extensions*, modules that can be configured specifically for the ActorSystem in question. A simple example of a support capability is the scheduler, which can send messages to actors periodically.

An ActorSystem returns an address to the created top-level actor instead of the actor itself. This address is called an ActorRef. The ActorRef can be used to send messages to the actor. This makes sense when you think about the fact that the actor could be on another server.

Sometimes you’d like to look up an actor in the actor system. This is where ActorPaths come in. You could compare the hierarchy of actors to a URL path structure. Every actor has a name. This name needs to be unique per level in the hierarchy: two sibling actors can’t have the same name (if you don’t provide a name, Akka generates one for you, but it’s a good idea to name all your actors). All actor references can be located directly by an actor path, absolute or relative.

1.7.2 ActorRef, mailbox, and actor

Messages are sent to the actor’s ActorRef. Every actor has a mailbox—it’s a lot like a queue. Messages sent to the ActorRef will be temporarily stored in the mailbox to be processed later, one at a time, in the order they arrived. Figure 1.21 shows the relationship between the ActorRef, the mailbox, and the actor.

How the actor actually processes the messages is described in the next section.

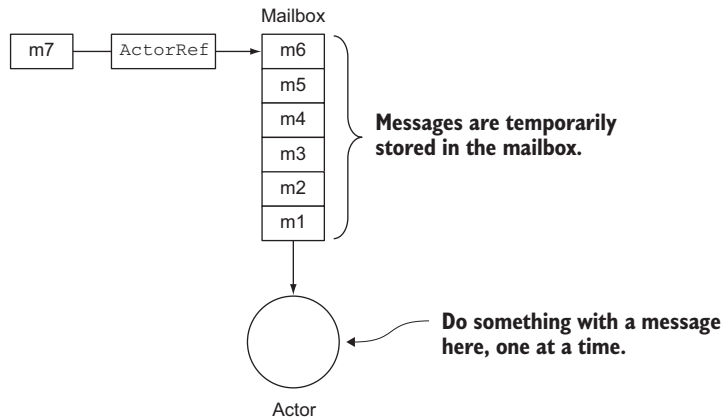


Figure 1.21 ActorRef, mailbox, actor

1.7.3 Dispatchers

Actors are invoked at some point by a dispatcher. The dispatcher pushes the messages in the mailbox through the actors, so to speak. This is shown in figure 1.22.

The type of dispatcher determines which threading model is used to push the messages through. Many actors can get messages pushed through on several threads, as shown in figure 1.23.

Figure 1.23 shows that messages m1 through m6 are going to be pushed through by the dispatcher on threads 1 and 2, and x4 through x9 on threads 3 and 4. This figure shouldn't make you think that you can or should control exactly which message will be

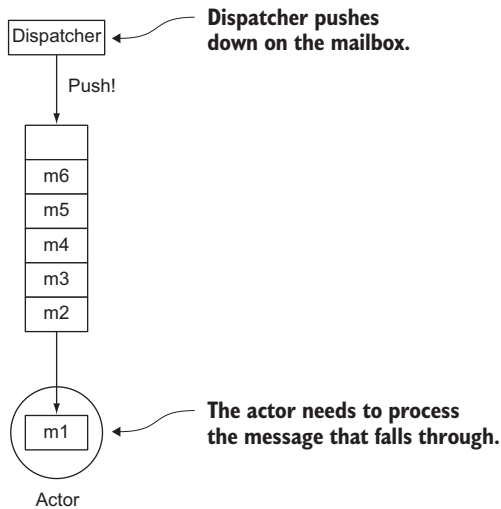


Figure 1.22 Dispatcher pushes messages through mailbox

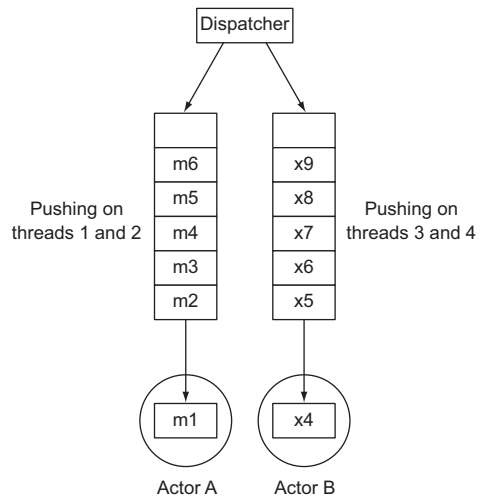


Figure 1.23 Dispatcher pushing messages through many actors

pushed through on which thread. What's important here is that you can configure the threading model to quite some extent. All kinds of dispatchers can be configured in some way, and you can allocate a dispatcher to an actor, a specific group of actors, or to all actors in the system.

So when you send a message to an actor, all you're really doing is leaving a message behind in its mailbox. Eventually a dispatcher will push it through the actor. The actor, in turn, can leave a message behind for the next actor, which will be pushed through at some point.

Actors are lightweight because they run on top of dispatchers; the actors aren't necessarily directly proportional to the number of threads. Akka actors take a lot less space than threads: around 2.7 million actors can fit in 1 GB of memory. That's a big difference compared to 4096 threads for 1 GB of memory, which means that you can create different types of actors more freely than you would when using threads directly.

There are different types of dispatchers to choose from that can be tuned to specific needs. Being able to configure and tune the dispatchers and the mailboxes that are used throughout the application gives a lot of flexibility when performance tuning. We give a couple of simple tips on performance tuning in chapter 15.

CALLBACK HELL A lot of frameworks out there provide asynchronous programming through callbacks. If you've used any of these, chances are high you've been to a place that is called *Callback Hell*, where every callback calls another callback, which calls another callback, and so on.

Compare this to how the dispatcher chops up the messages in mailboxes and pushes them through on a given thread. Actors don't need to provide a callback in a callback, all the way down to some sulfur pit, which is good news. Actors simply drop off messages in mailboxes and let the dispatcher sort out the rest.

1.7.4 Actors and the network

How do Akka actors communicate with each other across the network? `ActorRefs` are essentially addresses to actors, so all you need to change is how the addresses are linked to actors. If the toolkit takes care of the fact that an address can be local or remote, you can scale the solution just by configuring how the addresses are resolved.

Akka provides a remoting module (which we'll discuss in chapter 6) that enables the transparency you seek. Akka passes messages for a remote actor on to a remote machine where the actor resides, and passes the results back across the network.

The only thing that has to change is how the reference to remote actors is looked up, which can be achieved solely through configuration, as you'll see later. The code stays exactly the same, which means that you can often transition from scaling up to scaling out without having to change a single line of code.

The flexibility of resolving an address is heavily used in Akka, as we'll show throughout this book. Remote actors, clustering, and even the test toolkit use this flexibility.

1.8 Summary

Let's recap what you've learned in this chapter. Scaling is traditionally hard to get right. Both inflexibility and complexity quickly get out of control when scaling is required. Akka actors take advantage of key design decisions that provide more flexibility to scale.

Actors are a programming model for scaling up and out, where everything revolves around sending and receiving messages. Although it's not a silver bullet for every problem, being able to work with one programming model reduces some of the complexity of scaling.

Akka is centered on actors. What makes Akka unique is how effortlessly it provides support and additional tooling for building actor-based applications, so that you can focus on thinking and programming in actors.

At this point you should have an intuition that actors can give you more flexibility at a decent level of complexity, making it far easier to scale. But there's a lot more to be learned and, as always, the devil is in the details.

But first, let's get up and running with actors in the next chapter and build a simple HTTP server and deploy it on a PaaS (platform as a service)!

Up and running



In this chapter

- Fetching a project template
- Building a minimal Akka app for the cloud
- Deploying to Heroku

Our goal here is to show you how quickly you can make an Akka app that not only does something nontrivial, but is built to do it to scale, even in its easiest, early incarnations. We'll clone a project from github.com that contains our example, and then we'll walk through the essentials that you need to know to start building Akka apps. First we'll look at the dependencies that you need for a minimal app, using Lightbend's *Simple Build Tool* (*sbt*) to create a single JAR file that can be used to run the app. We'll build a minimal ticket-selling app, and in its first iteration we'll build a minimal set of REST services. We'll keep it as simple as possible to focus on essential Akka features. Finally we'll show you how easy it is to deploy this app to the cloud and get it working on Heroku, a popular cloud provider. What will be most remarkable is how quickly we get to this point!

One of the most exciting things about Akka is how easy it is to get up and running, and how flexible it is, given its small footprint runtime, as you'll soon see. We'll ignore some of the infrastructure details, and chapter 12 will go into more detail on how to use Akka HTTP, but you'll leave this chapter with enough informa-

tion to build serious REST interfaces of all types. You'll see in the next chapter how we can combine this with *TDD* (*test-driven development*).

2.1 Clone, build, and test interface

To make things easier, we've published the source code for the app on github.com, along with all the code for this book. The first thing you have to do is clone the repo to a directory of your choice.

Listing 2.1 Clone the example project

```
git clone https://github.com/RayRoestenburg/akka-in-action.git
```

Clone Git repo with complete,
working example code

This will create a directory named *akka-in-action* that contains the directory *chapter-up-and-running*, which contains the example project for this chapter. We expect that you're already familiar with Git and GitHub, among other tools. We'll use sbt, Git, the Heroku toolbelt, and *httpie* (an easy to use command-line HTTP client) in this chapter.

NOTE Please note that Akka 2.4 requires Java 8. If you've installed an earlier version of sbt, please make sure to remove it and upgrade to version 0.13.7 or higher. It's also useful to use sbt-extras from Paul Phillips (<https://github.com/paulp/sbt-extras>), which automatically figures out which version of sbt and Scala to use.

Let's look at the structure of the project. sbt follows a project structure similar to Maven. The major difference is that sbt allows for the use of Scala in the build files, and it has an interpreter. This makes it considerably more powerful. For more information on sbt, see the Manning Publications title *SBT in Action* (by Joshua Suereth and Matthew Farwell; www.manning.com/suereth2/). Inside the chapter-up-and-running directory, all the code for the server can be found in `src/main/scala`; configuration files and other resources in `src/main/resources`; and the tests in `src/test/scala`. The project should build right out of the box. Run the following command inside the chapter-up-and-running directory and keep your fingers crossed:

```
sbt assembly
```

Compiles and packages the code into a single JAR

You should see sbt booting up, getting all needed dependencies, running all the tests, and finally building one fat JAR into `target/scala-2.11/goticks-assembly-1.0.jar`. You could run the server by simply running the commands in the following listing.

Listing 2.2 Running the JAR

```
java -jar target/scala-2.11/goticks-assembly-1.0.jar  
RestApi bound to /0:0:0:0:0:0:0:5000
```

Runs the app like
any other Java code

Output to the console: HTTP server
is started and listens on port 5000

Now that we've verified that the project builds correctly, it's time to talk about what it does. In the next section, we'll start with the build file and then look at the resources, and the actual code for the services.

2.1.1 *Build with sbt*

Let's first look at the build file. We're using the simple sbt DSL (domain-specific language) for build files in this chapter because it gives us all we need right now. As we go forward in the book, we'll be back to add more dependencies, but you can see that for your future projects you'll be able to get going quickly, and without the aid of a template, or by cutting and pasting large build files from other projects. If you haven't worked with the sbt settings DSL before, it's important to note that you need to put an empty line between setting lines in the file (which is not required in full configuration mode, in which case you can write Scala code as usual). The build file is located directly under the chapter-up-and-running directory in a file called build.sbt.

Listing 2.3 The sbt build file

```
enablePlugins(JavaServerAppPackaging)
name := "goticks"
version := "1.0"
organization := "com.goticks"

libraryDependencies += {
  val akkaVersion = "2.4.9"
  Seq(
    "com.typesafe.akka" %% "akka-actor" % akkaVersion,
    "com.typesafe.akka" %% "akka-http-core" % akkaVersion,
    "com.typesafe.akka" %% "akka-http-experimental" % akkaVersion,
    "com.typesafe.akka" %% "akka-http-spray-json-experimental" %
      akkaVersion,
    "io.spray" %% "spray-json" % "1.3.1",
    "com.typesafe.akka" %% "akka-slf4j" % akkaVersion,
    "ch.qos.logback" % "logback-classic" % "1.1.3",
    "com.typesafe.akka" %% "akka-testkit" % akkaVersion % "test",
    "org.scalatest" %% "scalatest" % "2.2.0" % "test"
  )
}
```

In case you were wondering where the libraries are downloaded from, sbt uses a set of predefined repositories, including a Lightbend repository that hosts the Akka libraries that we use here. For those with experience in Maven, this looks decidedly more compact. Like Maven, once we have the repository and dependency mapped, we can easily get newer versions by just changing a single value.

Every dependency points to a Maven artifact in the format *organization % module % version* (the %% is for automatically using the right Scala version of the library). The most important dependency here is the *akka-actor* module. Now that we have our

build file set up, we can compile the code, run the tests, and build the JAR file. Run the following command in the `chapter-up-and-running` directory.

Listing 2.4 Running tests

```
sbt clean compile test
```



Delete target; then compile and run tests

If any dependencies still need to be downloaded, sbt will do that automatically. Now that we have the build file in place, let's take a closer look at what we're trying to achieve with this example in the next section.

2.1.2 Fast-forward to the GoTicks.com REST server

Our ticket-selling service will allow customers to buy tickets to all sorts of events, concerts, sports games, and the like. Let's say we're part of a startup called GoTicks.com, and in this first iteration we've been assigned to build the backend REST server for the first version of the service. Right now we want customers to get a numbered ticket to a show. Once all the tickets are sold for an event, the server should respond with a 404 (Not Found) HTTP status code. The first thing we'll implement in the REST API will have to be the addition of a new event (since all other services will require the presence of an event in the system). A new event only contains the name of the event—say "RHCP" for the Red Hot Chili Peppers—and the total number of tickets we can sell for the given venue.

The requirements for the `RestApi` are shown in table 2.1.

Table 2.1 REST API

Description	HTTP method	URL	Request body	Status code	Response example
Create an event	POST	/events/RHCP	{ "tickets" : 250 }	201 Created	{ "name": "RHCP", "tickets": 250 }
Get all events	GET	/events	N/A	200 OK	[{ event : "RHCP", tickets : 249 }, { event : "Radiohead", tickets : 130 }]
Buy tickets	POST	/events/RHCP/tickets	{ "tickets" : 2 }	201 Created	{ "event" : "RHCP", "entries" : [{ "id" : 1 }, { "id" : 2 }] }
Cancel an event	DELETE	/events/RHCP	N/A	200 OK	{ event : "RHCP", tickets : 249 }

Let's build the app and run it inside sbt. Go to the chapter-up-and-running directory and execute the following command.

Listing 2.5 Starting up the app locally with sbt

```
sbt run
```

← Tells the build tool to compile and run our app

```
[info] Running com.goticks.Main
INFO [Slf4jLogger]: Slf4jLogger started
RestApi bound to /0:0:0:0:0:0:0:5000
```

As are most build tools, sbt is similar to make: if the code needs to be compiled, it will be; then packaged, and so on. Unlike a lot of build tools, sbt can also deploy and run the app locally. If you get an error, make sure that you're not already running the server in another console, or that some other process isn't already using port 5000. Let's see if everything works by using `httpie`,¹ a human-readable HTTP command-line tool that makes it simple to send HTTP requests. It has support for JSON and handles the required housekeeping in headers, among other things. First let's see if we can create an event with a number of tickets.

Listing 2.6 Creating an event from the command line

```
http POST localhost:5000/events/RHCP tickets:=10
```

← **httpie command simply sends POST request to our running server, with one parameter**

```
HTTP/1.1 201 Created
Connection: keep-alive
Content-Length: 76
Content-Type: text/plain; charset=UTF-8
Date: Mon, 20 Apr 2015 12:13:35 GMT
Proxy-Connection: keep-alive
Server: GoTicks.com REST API
```

← **Response from the server (201 Created indicates success)**

```
{
  "name": "RHCP",
  "tickets": 10
}
```

The parameter is transformed into a JSON body. Notice the parameter uses `:=` instead of `=`. This means that the parameter is a non-string field. The format of our command is translated into `{ "tickets" : 10 }`. The whole following block is the complete HTTP response dumped by `httpie` to the console. The event is now created. Let's create another one:

```
http POST localhost:5000/events/DjMadlib tickets:=15
```

Now let's try out the GET request. Per the REST conventions, a GET whose URL ends with an entity type should return a list of known instances of that entity.

¹ You can get `httpie` here: <https://github.com/jakubroztocil/httpie>

Listing 2.7 Requesting a list of all events

```

http GET localhost:5000/events
...
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 110
Content-Type: application/json; charset=UTF-8
Date: Mon, 20 Apr 2015 12:18:01 GMT
Proxy-Connection: keep-alive
Server: GoTicks.com REST API

{
  "events": [
    {
      "name": "DjMadlib",
      "tickets": 15
    },
    {
      "name": "RHCP",
      "tickets": 10
    }
  ]
}

```

Requests a list of all current Event instances

Completes response from our HTTP server (200 indicates success)

Notice that we see both events, and all the tickets are still available. Now let's see if we can buy two tickets for the RHCP event.

Listing 2.8 Purchasing two tickets to RHCP

```

http POST localhost:5000/events/RHCP/tickets tickets:=2
...
HTTP/1.1 201 Created
Connection: keep-alive
Content-Length: 74
Content-Type: application/json; charset=UTF-8
Date: Mon, 20 Apr 2015 12:19:41 GMT
Proxy-Connection: keep-alive
Server: GoTicks.com REST API

{
  "entries": [
    {
      "id": 1
    },
    {
      "id": 2
    }
  ],
  "event": "RHCP"
}

```

Sends a POST to request 2 tickets

Server response in the console (201 Created indicates the tickets have been created)

The tickets we purchased, as JSON

The presumption here is that there are at least two Tickets left for this Event; otherwise, we'd have gotten a 404.

If you do the GET with path /events again, you should see the following response.

Listing 2.9 GET after two events created

```
HTTP/1.1 200 OK
Content-Length: 91
Content-Type: application/json; charset=UTF-8
Date: Mon, 20 Apr 2015 12:19:42 GMT
Server: GoTicks.com REST API

[
  {
    "event":
      "DjMadlib",
    "nrOfTickets": 15
  },
  {
    "event":
      "RHCP",
    "nrOfTickets": 8
  }
]
```

As expected, there are now only 8 tickets left for RHCP. You should get a 404 after buying all tickets.

Listing 2.10 Results when seats are gone

```
HTTP/1.1 404 Not Found
Content-Length: 83
Content-Type: text/plain
Date: Tue, 16 Apr 2013 12:42:57 GMT
Server: GoTicks.com REST API
```

```
The requested resource could not be found
but may be available again in the future.
```

← Server responds with
404 when we're out of
Tickets for an Event

That concludes all the API calls in the REST API. Clearly, at this point, the application supports the basic Event CRUD cycle, from creation of the actual Event through the sale of all the tickets until they're sold out. This isn't comprehensive; for instance, we're not accounting for events that won't sell out, but whose tickets will need to become unavailable once the actual event has started. Now let's look at the details of how we're going to get to this result in the next section.

2.2 *Explore the actors in the app*

In this section we'll look at how the app is built. You can participate and build the actors yourself, or just follow along from the source code on github.com. As you now

know, actors can perform four operations; create, send/receive, become, and supervise. In this example we'll only touch on the first two operations. First, we'll take a look at the overall structure: how operations will be carried out by the various collaborators (actors) to provide the core functionality—creating events, issuing tickets, and finishing events.

2.2.1 Structure of the app

The app consists of two actor classes in total. The first thing we have to do is create an actor system that will contain all the actors. After that the actors can create each other. Figure 2.1 shows the sequence.

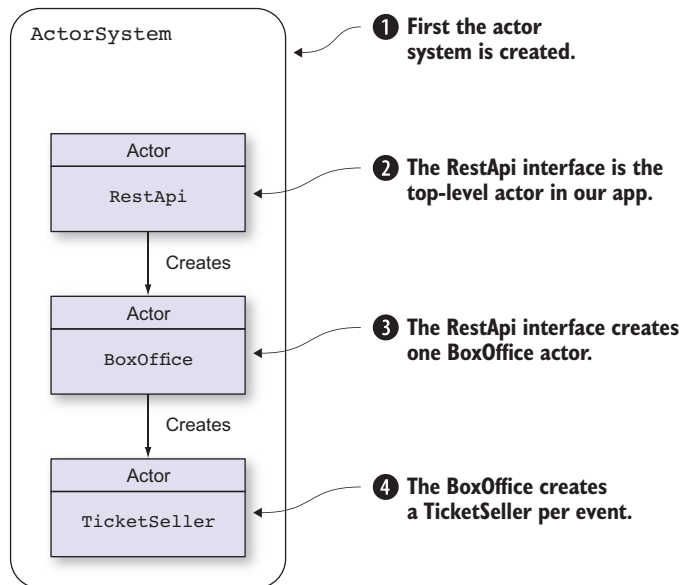


Figure 2.1 Actor creation sequence triggered by REST request

The RestApi contains a number of *routes* to handle the HTTP requests. The routes define how HTTP requests should be handled using a convenient DSL, which is provided by the akka-http module. We'll discuss the routes in section 2.2.4. The RestApi is basically an adapter for HTTP: it takes care of converting from and to JSON, and provides the required HTTP response. We'll show later how we connect this actor to an HTTP server. Even in this simplest example, you can see how the fulfillment of a request spawns a number of collaborators, each with specific responsibilities. The TicketSeller eventually keeps track of the tickets for one particular event and sells the tickets. Figure 2.2 shows how a request for creating an Event flows through the actor system (this was the first service we showed in table 2.1).

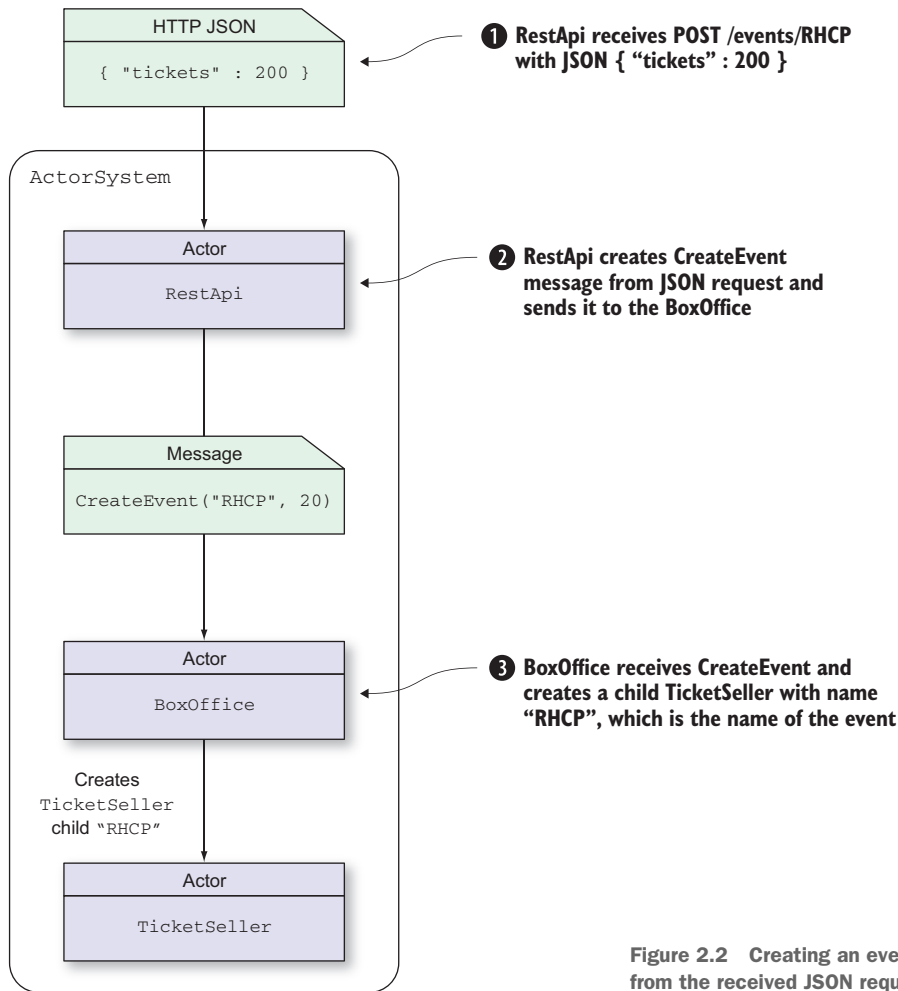


Figure 2.2 Creating an event from the received JSON request

The second service we discussed was the ability for a Customer to purchase a ticket (now that we have an Event). Figure 2.3 shows what should happen when such a ticket purchase request is received (as JSON).

Let's step back and start looking at the code as a whole. First up: the `Main` class, which starts everything up. The `Main` object is a simple Scala app that you can run just like any other Scala app. It's similar to a Java class with a `main` method. Before we get into the complete listing of the `Main` class, let's look at the most important expressions first, starting with the import statements in listing 2.11.

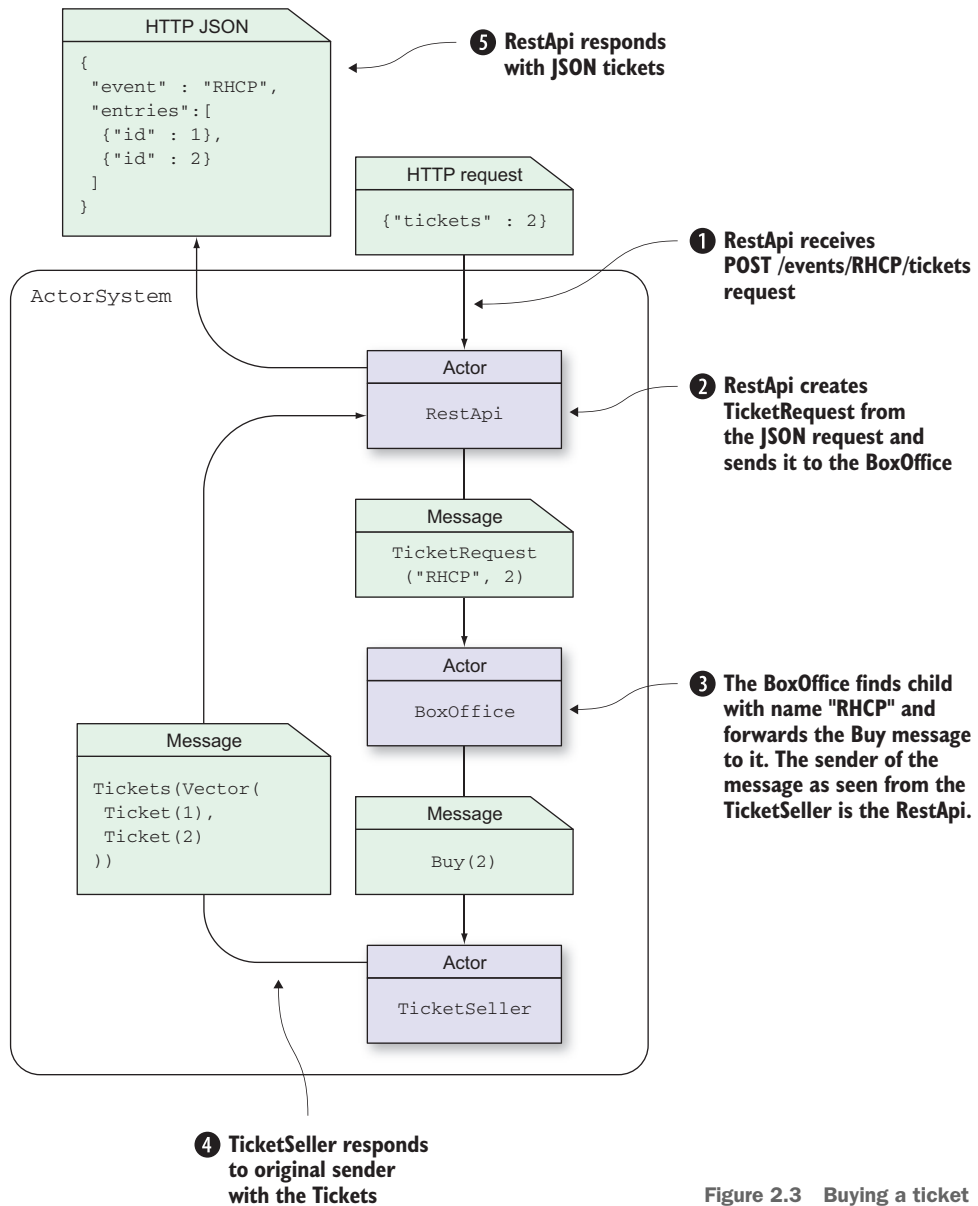


Figure 2.3 Buying a ticket

Listing 2.11 Main class import statements

```
import akka.actor.{ ActorSystem , Actor, Props }
import akka.event.Logging
import akka.util.Timeout
import akka.http.scaladsl.Http
```

Actor-related code is located
in akka.actor package

Logging extension

Asking requires timeout

HTTP-related code is located
in akka.http package


```
import akka.http.scaladsl.Http.ServerBinding
import akka.http.scaladsl.server.Directives._
import akka.stream.ActorMaterializer
import com.typesafe.config.{ Config, ConfigFactory }
```

Imports typesafe configuration library

The `Main` class needs to create an `ActorSystem` first. It then creates the `RestApi`, gets the HTTP extension, and binds the `RestApi` routes to the HTTP extension. How this is done is shown a little later. Akka uses so-called *extensions* for many supporting tools, as you'll see in the rest of this book; `Http` and `Logging` are the first examples of these.

We don't want to hardcode configuration parameters like the host and port that the server should listen to, so we use the *Typesafe Config Library* to configure this (chapter 7 goes into the details of using this configuration library).

The following listing shows the essential expressions to start the `ActorSystem` and the HTTP extension, and get the `RestApi` routes to bind to HTTP, which is the responsibility of the `Main` object.

Listing 2.12 Starting the HTTP server

```
object Main extends App
  with RequestTimeout {
    val config = ConfigFactory.load()
    val host = config.getString("http.host")
    val port = config.getInt("http.port")

    implicit val system = ActorSystem()
    implicit val ec = system.dispatcher

    val api = new RestApi(system, requestTimeout(config)).routes
    implicit val materializer = ActorMaterializer()
    val bindingFuture: Future[ServerBinding] =
      Http().bindAndHandle(api, host, port)
  }
```

Gets the host and a port from the configuration

`bindAndHandle` is asynchronous and requires an implicit `ExecutionContext`.

`RestApi` provides the HTTP routes

Starts HTTP server with the `RestAPI` routes

The `Main` object extends `App` like any Scala application.

The `ActorSystem` is *active* immediately after it has been created, starting any thread pools as required.

`Http()` returns the HTTP extension. `bindAndHandle` binds the routes defined in the `RestApi` to the HTTP server. `bindAndHandle` is an asynchronous method that returns a `Future` before it has completed. We'll gloss over the details of this for now and get back to it later (in chapter 5 on futures). The `Main` app doesn't exit immediately, and the `ActorSystem` creates non-daemon threads and keeps running (until it's terminated).

The `RequestTimeout` trait is shown for completeness sake, which makes it possible for the `RestApi` to use the configured request timeout in akka-http.

Listing 2.13 The Main object

```

trait RequestTimeout {
  import scala.concurrent.duration._
  def requestTimeout(config: Config): Timeout = {
    val t = config.getString("spray.can.server.request-timeout")
    val d = Duration(t)
    FiniteDuration(d.length, d.unit)
  }
}

```

Uses the default request timeout of akka-http server configuration

Getting the request timeout is extracted into a `RequestTimeout` trait (which you can skip for now; don't worry if the code isn't immediately clear).

You really don't need to understand all the details of how the HTTP extension is making all of this possible yet because we'll cover them in detail later.

The actors in the app communicate with each other through messages. The messages that an actor can receive or send back on a request are bundled together in the actors' companion object. The `BoxOffice` messages are shown next.

Listing 2.14 BoxOffice messages

```

case class CreateEvent(name: String, tickets: Int)
case class GetEvent(name: String)
case object GetEvents
case class GetTickets(event: String, tickets: Int)
case class CancelEvent(name: String)

case class Event(name: String, tickets: Int)
case class Events(events: Vector[Event])

sealed trait EventResponse
case class EventCreated(event: Event) extends EventResponse
case object EventExists extends EventResponse

```

Message to create an event

Message to get an event

Message to request all events

Message to get tickets for an event

Message to cancel the event

Message describing the event

Message to describe a list of events

Message response to CreateEvent

Message to indicate the event was created

Message to indicate that the event already exists

The `TicketSeller` sends or receives the messages shown next.

Listing 2.15 TicketSeller messages

```

case class Add(tickets: Vector[Ticket])
case class Buy(tickets: Int)
case class Ticket(id: Int)
case class Tickets(event: String,
  entries: Vector[Ticket] = Vector.empty[Ticket])

```

Message to add tickets to the TicketSeller

Message to buy tickets from the TicketSeller

A ticket

A list of tickets for an event

```

case object GetEvent
case object Cancel

```

A message to cancel the event

A message containing the remaining tickets for the event

As is typical of REST apps, we have an interface that revolves around the lifecycles of the core entities: Events and Tickets. All of these messages are immutable (since they are case classes or objects). The Actors have to be designed to get all the information they need, and produce all that is needed if they enlist any collaborators. This lends itself well to REST. In the next sections, we'll look at the Actors in more detail. We'll start from the `TicketSeller` and work our way up.

2.2.2 The actor that handles the sale: `TicketSeller`

The `TicketSeller` is created by the `BoxOffice` and simply keeps a list of tickets. Every time tickets are requested, it takes the number of requested tickets off the list. The following listing shows the code for the `TicketSeller`.

Listing 2.16 `TicketSeller` implementation

```

class TicketSeller(event: String) extends Actor {
  import TicketSeller._

  var tickets = Vector.empty[Ticket]

  def receive = {
    case Add(newTickets) => tickets = tickets ++ newTickets
    case Buy(nrOfTickets) =>
      val entries = tickets.take(nrOfTickets).toVector
      if(entries.size >= nrOfTickets) {
        sender() ! Tickets(event, entries)
        tickets = tickets.drop(nrOfTickets)
      } else sender() ! Tickets(event)
    case GetEvent => sender() ! Some(BoxOffice.Event(event, tickets.size))
    case Cancel =>
      sender() ! Some(BoxOffice.Event(event, tickets.size))
      self ! PoisonPill
  }
}

```

The list of tickets

Returns an event containing the number of tickets left when `GetEvent` is received

Adds the new tickets to the existing list of tickets when `Tickets` message is received

Takes a number of tickets off the list and responds with a `Tickets` message containing the tickets if there are enough tickets available; otherwise, responds with an empty `Tickets` message

The `TicketSeller` keeps track of the available tickets using an immutable list. A mutable list could have been safe as well because it's only available within the actor and therefore never accessed from more than one thread at any given moment.

Still, you should prefer immutable lists. You might forget that it is mutable when you return a part of the list or the entire list to another actor. For instance, look at the `take` method that we use to get the first couple of tickets off the list. On a mutable list (`scala.collection.mutable.ListBuffer`), `take` returns a list of the same type (`ListBuffer`), which is obviously mutable.

In the next section we'll look at the `BoxOffice` actor.

2.2.3 The BoxOffice actor

The BoxOffice needs to create a TicketSeller child for every event and delegate the selling to the TicketSeller responsible for the requested event. The following listing shows how the BoxOffice responds to a CreateEvent message.

Listing 2.17 BoxOffice creates TicketSellers

```
def createTicketSeller(name: String) =
  context.actorOf(TicketSeller.props(name), name)

def receive = {
  case CreateEvent(name, tickets) =>
    def create() = {
      val eventTickets = createTicketSeller(name)
      val newTickets = (1 to tickets).map { ticketId =>
        TicketSeller.Ticket(ticketId)
      }.toVector
      eventTickets ! TicketSeller.Add(newTickets)
      sender() ! EventCreated
    }
    context.child(name).fold(create())(_ => sender() ! EventExists)
```

Creates a TicketSeller using its context, defined in a separate method so it's easy to override during testing

A local method that creates the ticket seller, adds the tickets to the ticket seller, and responds with EventCreated

Creates and responds with EventCreated, or responds with EventExists

The BoxOffice creates a TicketSeller for each event that doesn't exist yet. Notice that it uses its *context* instead of the actor system to create the actor; actors created with the context of another actor are its children and subject to the parent actor's supervision (much more about that in subsequent chapters). The BoxOffice builds up a list of numbered tickets for the event and sends these tickets to the TicketSeller. It also responds to the sender of the CreateEvent message that the Event has been created (the RestApi actor is the sender here). The following listing shows how the BoxOffice responds to the GetTickets message.

Listing 2.18 Getting tickets

```
case GetTickets(event, tickets) =>
  def notFound() = sender() ! TicketSeller.Tickets(event)
  def buy(child: ActorRef) =
    child.forward(TicketSeller.Buy(tickets))
  context.child(event).fold(notFound())(buy)
```

Sends an empty Tickets message if the ticket seller couldn't be found

Executes notFound or buys with the found TicketSeller

Buys from the found TicketSeller

The Buy message is forwarded to a TicketSeller. Forwarding makes it possible for the BoxOffice to send messages as a proxy for the RestApi. The response of the TicketSeller will go directly to the RestApi.

The next message, GetEvents, is more involved and will get you extra credit if you get it the first time. We're going to ask all TicketSellers for the number of tickets

they have left and combine all the results into a list of events. This gets interesting because `ask` is an asynchronous operation, and at the same time we don't want to wait and block the `BoxOffice` from handling other requests.

The following code uses a concept called *futures*, which will be explained further in chapter 5, so if you feel like skipping it now that's fine. If you're up for a tough challenge though, let's look at the code!

Listing 2.19 Getting tickets

```
case GetEvents =>
  import akka.pattern.ask
  import akka.pattern.pipe

  def getEvents = context.children.map { child =>
    self.ask(GetEvent(child.path.name)).mapTo[Option[Event]]
  }
  def convertToEvents(f: Future[Iterable[Option[Event]]]) =
    f.map(_._flatten).map(l=> Events(l.toVector))

  pipe(convertToEvents(Future.sequence(getEvents))) to sender()
```

A local method definition for asking all TicketSellers about the events they sell tickets for

ask returns a Future, a type that will eventually contain a value. getEvents returns Iterable[Future[Option[Event]]]; sequence can turn this into a Future[Iterable[Option[Event]]]. pipe sends the value inside the Future to an actor the moment it's complete, in this case the sender of the GetEvents message, the RestApi.

We're going to ask all TicketSellers. Asking GetEvent returns an Option[Event], so when mapping over all TicketSellers we'll end up with an Iterable[Option[Event]]. This method flattens the Iterable[Option[Event]] into a Iterable[Event], leaving out all the empty Option results. The Iterable is transformed into an Events message.

Right now we'll skim over this example and just look at the concepts. What's happening here is that an `ask` method returns immediately with a *future*. A future is a value that's going to be available at some point in the future (hence the name). Instead of waiting for the response value (the event containing the number of tickets left), we get a future reference (which you could read as "use for future reference"). We never read the value directly, but instead we define what should happen once the value becomes available. We can even combine a list of future values into one list of values and describe what should happen with this list once all of the asynchronous operations complete.

The code finally sends an `Events` message back to the sender once all responses have been handled, by using another pattern, `pipe`, which makes it easier to eventually send the values inside futures to actors.

Don't worry if this isn't immediately clear; we have a whole chapter devoted to this subject. We're just trying to get you curious about this awesome feature—check out chapter 5 on futures if you can't wait to find out how these nonblocking asynchronous operations work.

That concludes the salient details of the `BoxOffice`. We have one actor left in the app, which will be handled in the next section: the `RestApi`.

2.2.4 RestApi

The RestApi uses the Akka HTTP routing DSL, which will be covered in detail in chapter 12. Services interfaces, as they grow, need more sophisticated routing of requests. Since we're really just creating an Event and then selling the Tickets to it, our routing requirements are few at this point. The RestApi defines a couple of classes that it uses to convert from and to JSON, shown next.

Listing 2.20 Event messages used in the RestApi

```
case class EventDescription(tickets: Int) {
  require(tickets > 0)
}

case class TicketRequest(tickets: Int) {
  require(tickets > 0)
}

case class Error(message: String)
```

← Message containing the initial number of tickets for the event

← Message containing the required number of tickets

← Messages containing an error

Let's look at the details of doing simple request routing in the following listings. First, the RestApi needs to handle a POST request to create an Event.

Listing 2.21 Event route definition

```
def eventRoute =
  pathPrefix("events" / Segment) { event =>
    pathEndOrSingleSlash {
      post {
        // POST /events/:event
        entity(as[EventDescription]) { ed =>
          onSuccess(createEvent(event, ed.tickets)) {
            BoxOffice.EventCreated(event) => complete(Created, event)
            case BoxOffice.EventExists =>
              val err = Error(s"$event event exists already.")
              complete(BadRequest, err)
          }
        }
      } ~
      get {
        // GET /events/:event
        onSuccess(getEvent(event)) {
          _.fold(complete(NotFound))(e => complete(OK, e))
        }
      } ~
      delete {
        // DELETE /events/:event
        onSuccess(cancelEvent(event)) {
          _.fold(complete(NotFound))(e => complete(OK, e))
        }
      }
    }
  }
}
```

Completes request with 201 Created when result is successful

Creates event using createEvent method that calls BoxOffice actor

Completes request with 400 BadRequest if event could not be created

The route uses a `BoxOfficeApi` trait, which has methods that wrap the interaction with the `BoxOffice` actor so that the route DSL code stays nice and clean, shown next.

Listing 2.22 `BoxOffice` API to wrap all interactions with the `BoxOffice` actor

```
trait BoxOfficeApi {
  import BoxOffice._

  def createBoxOffice(): ActorRef

  implicit def executionContext: ExecutionContext
  implicit def requestTimeout: Timeout

  lazy val boxOffice = createBoxOffice()

  def createEvent(event: String, nrOfTickets: Int) =
    boxOffice.ask(CreateEvent(event, nrOfTickets))
      .mapTo[EventResponse]

  def getEvents() =
    boxOffice.ask(GetEvents).mapTo[Events]

  def getEvent(event: String) =
    boxOffice.ask(GetEvent(event))
      .mapTo[Option[Event]]

  def cancelEvent(event: String) =
    boxOffice.ask(CancelEvent(event))
      .mapTo[Option[Event]]

  def requestTickets(event: String, tickets: Int) =
    boxOffice.ask(GetTickets(event, tickets))
      .mapTo[TicketSeller.Tickets]
}
```

The `RestApi` implements the `createBoxOffice` method to create a `BoxOffice` child actor. The following code shows a snippet of the DSL that's used to sell the tickets.

Listing 2.23 Ticket route definition

```
def ticketsRoute =
  pathPrefix("events" / Segment / "tickets") { event =>
    post {
      pathEndOrSingleSlash {
        // POST /events/:event/tickets
        entity(as[TicketRequest]) { request =>
          onSuccess(requestTickets(event, request.tickets)) { tickets =>
            if(tickets.entries.isEmpty) complete(NotFound)
            else complete(Created, tickets)
          }
        }
      }
    }
  }
```

Unmarshalls JSON tickets request into TicketRequest case class

Responds with 201 Created, marshalling the tickets to a JSON entity

Responds with 404 Not Found if the tickets aren't available

The messages are automatically converted back to JSON. You can find the details of how this is done in chapter 12. That concludes all the actors in the first iteration of the GoTicks.com application. If you followed along or even tried to build this yourself, congratulations! You’ve just seen how to build your first fully asynchronous Akka actor app with a fully functional REST API. While the app itself is rather trivial, we’ve already made it so that the processing is fully concurrent, and the actual selling of tickets is both scalable (because it’s already concurrent) and fault tolerant (you’ll see much more on that). This example also showed how you can do asynchronous processing within the synchronous request/response paradigm of the HTTP world. We hope that you’ve found that it takes only a few lines of code to build this app. Compare that to a more traditional toolkit or framework and we’re sure that you’re pleasantly surprised to see how little code was needed. For a little cherry on the top, we’ll show you what we need to do to deploy this minimal app to the cloud. We’ll get this app running on Heroku.com in the next section.

2.3 Into the cloud

Heroku.com is a popular cloud provider that has support for Scala applications, and free instances that you can play with. In this section we’ll show you how easy it is to get the GoTicks.com app up and running on Heroku. We expect that you’ve already installed the Heroku toolbelt (see <https://toolbelt.heroku.com/>). If not, please refer to the Heroku website (<https://devcenter.heroku.com/articles/heroku-command>) for how to install it. You’ll also need to sign up for an account on heroku.com. Visit their site—the signup speaks for itself. In the next section, we’ll first create an app on heroku.com. After that we’ll deploy it and run it.

2.3.1 Create the app on Heroku

First, log in to your Heroku account and create a new Heroku app that will host our GoTicks.com app. Execute the following commands in the chapter-up-and-running directory.

Listing 2.24 Create the app on Heroku

```
heroku login
heroku create

Creating damp-bayou-9575... done,
stack is cedar
http://damp-bayou-9575.herokuapp.com/
|
git@heroku.com:damp-bayou-9575.git
```

You should see something like the response shown in the listing.

We need to add a couple of things to our project so Heroku understands how to build our code. First the project/plugins.sbt file.

Listing 2.25 BoxOffice API to wrap all interactions with the BoxOffice actor

Uses
assembly to
create one
big JAR file,
needed for
deployment
to Heroku

```
resolvers += Classpaths.typesafeReleases

addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.13.0")

addSbtPlugin("com.typesafe.sbt" % "sbt-native-
  packager" % "1.0.0")
```

Uses the Typesafe
Releases repository

Uses the packager to
create startup scripts
for running the app
on Heroku

This is a pretty minor intrusion to build one fat JAR and build a native script (in the case of Heroku, this is a Bash shell script; Heroku runs on Ubuntu Linux). We also need a *Procfile* right under the chapter-up-and-running directory, which tells Heroku that our app should be run on a *web dyno*—one of the types of processes Heroku runs on its virtual dyno manifold. The Procfile is shown next.

Listing 2.26 Heroku Procfile

```
web: target/universal/stage/bin/goticks
```

It specifies that Heroku should run the Bash script that the `sbt-native-packager` plugin has built. Let's first test to see if everything runs locally:

```
sbt clean compile stage
```

Cleans target, then builds our
archive but doesn't deploy

```
heroku local
23:30:11 web.1 | started with pid 19504
23:30:12 web.1 | INFO [Slf4jLogger]: Slf4jLogger started
23:30:12 web.1 | REST interface bound to /0:0:0:0:0:0:0:0:5000
23:30:12 web.1 | INFO [HttpListener]: Bound to /0.0.0.0:5000
```

Tells Heroku to
grab archive
and start up
our app locally

Heroku manages to load
the app; we have a PID.

This is all that's required to prepare an application for deployment on Heroku. It lets us go through the whole cycle locally so that we're working at maximum speed while getting our first deploy done. Once we actually deploy to Heroku, you'll see that all subsequent pushes to the cloud instances are accomplished directly through Git by simply pushing the desired version of the source to our remote instance. The deployment of the app to the Heroku cloud instance is described in the next section.

2.3.2 Deploy and run on Heroku

We've just verified that we could locally run the app with `heroku local`. We created a new app on Heroku with `heroku create`. This command also added a git remote with the name *heroku* to the Git configuration. All we have to do now is make sure all changes are committed locally to the Git repository. After that, push the code to Heroku with the following command:

```
git subtree push --prefix chapter-up-and-running heroku master
```

← **Pushes to Heroku to deploy**

```
----> Scala app detected
-----> Installing OpenJDK 1.6...
.... // resolving downloads, downloading dependencies
....
-----> Compiled slug size is 43.1MB
-----> Launching... done,
      v1 http://damp-bayou-9575.herokuapp.com deployed to Heroku
```

← **Just as before, Heroku now builds app, this time on remote instance**

```
To git@heroku.com:damp-bayou-9575.git
* [new branch] master -> master
```

← **Finally, like any other Git push, success: master now on remote**

This assumes that you committed any changes to your master branch and that the project resides in the root of the Git repo. Heroku hooks into the Git push process and identifies the code as a Scala app. It downloads all dependencies on the cloud, compiles the code, and starts the application. Finally, you should see something like the output shown in the listing.

Using the project akka-in-action from GitHub

Normally, you'd use `git push heroku master` to deploy to Heroku. When you're using our project akka-in-action from GitHub, this command won't work, because the application isn't in the root of the Git repo. To make this work, you need to tell Heroku that it should use a subtree, as follows:

```
git subtree push --prefix chapter-up-and-running heroku master
```

For more info see the README.md file within the chapter-up-and-running directory.

This shows the console on creation of the app; note that Heroku figured out that our app is a Scala app, so it installed the OpenJDK, and then compiled and launched the source in the instance. The app is now deployed and started on Heroku. You can now use `httpie` again to test the app on Heroku.

Listing 2.27 Test Heroku instance with `httpie`

```
http POST damp-bayou-9575.herokuapp.com/events/RHCP tickets:=250
http POST damp-bayou-9575.herokuapp.com/events/RHCP/tickets tickets:=4
```

These commands should result in the same responses we saw before (see listing 2.10). Congratulations, you just deployed your first Akka app to Heroku! With that, we conclude this first iteration of the GoTicks.com app. Now that the app is deployed on Heroku, you can call it from anywhere.

2.4 Summary

In this chapter you've seen how little is necessary to build a fully functional REST service out of actors. All interactions were asynchronous. The service performed as expected when we tested it with the `httpie` command-line tool.

We even deployed our app (via [Heroku.com](https://heroku.com)) into the cloud! We hope you got excited about what a quick, out-of-the-box experience Akka offers. The [GoTicks.com](https://goticks.com) app isn't ready for production yet. There's no persistent storage for tickets. We've deployed to Heroku, but web dynos can be replaced at any time, so only storing the tickets in memory won't work in real life. The app is scaled up but has not scaled out yet to multiple nodes.

But we promise to look into those topics in later chapters, where we'll gradually get closer to a real-world system. In the next chapter, we'll look at how to test actor systems.

Test-driven development with actors

In this chapter

- Unit testing actors synchronously
- Unit testing actors asynchronously
- Unit testing actor messaging patterns

It's amusing to think back to when TDD first appeared on the scene—the primary objection was that tests took too long, and thus held up development. Though you rarely hear that today, there's a vast difference in the testing load both between different stacks, and through different phases (such as unit versus integration tests). Everyone has a rapid, fluid experience on the unit side, when testing is confined to a single component. Tests that involve collaborators are where ease and speed generally evaporate rapidly. Actors provide an interesting solution to this problem for the following reasons:

- Actors are a more direct match for tests because they embody behavior (and almost all TDD has at least some BDD—behavior-driven development—in it).
- Too often, regular unit tests test only the interface, or have to test the interface and functionality separately.

- Actors are built on messaging, which has huge advantages for testing, because you can easily simulate behaviors by sending messages.

Before we start testing (and coding), we'll take several of the concepts from the previous chapter and show their expression in code, introducing the Actor API for creating actors, and then sending and receiving messages. We'll cover important details about how actors are actually run and some rules you have to follow to prevent problems. After that, we'll move on to the implementation of some common scenarios, taking a test-driven approach to writing actors, immediately verifying that the code does what we expect. At each step along the way, we'll focus first on the goal that we'll try to achieve with the code (one of the main points of TDD). Next, we'll write a test specification for the Actor, which will start the development of the code (TDD/test-first style). Then we'll write enough code to make the test pass, and repeat. Rules that need to be followed to prevent accidentally sharing state will be discovered as we go, as well as some of the details of how actors work in Akka that have an impact on test development.

3.1 *Testing actors*

First, we'll work on how to test sending and receiving messages, in fire-and-forget style (one-way) followed by request-response style (two-way) interaction. We'll use the *ScalaTest* unit-testing framework that's also used to test Akka itself. *ScalaTest* is an xUnit-style testing framework; if you're not familiar with it and would like to know more about it, please visit www.scalatest.org/ for more information. The *ScalaTest* framework is designed for readability, so it should be easy to read and follow the test without much introduction. On first exposure, testing Actors is more difficult than testing normal objects for a couple of reasons:

- *Timing*—Sending messages is asynchronous, so it's difficult to know when to assert expected values in the unit test.
- *Asynchronicity*—Actors are meant to be run in parallel on several threads. Multi-threaded tests are more difficult than single-threaded tests and require concurrency primitives like locks, latches, and barriers to synchronize results from various actors. This is exactly the kind of thing you want to get further away from. Incorrect usage of just one barrier can block a unit test, which in turn halts the execution of a full test suite.
- *Statelessness*—An actor hides its internal state and doesn't allow access to this state. Access should only be possible through the *ActorRef*. Calling a method on an actor and checking its state, which is something you'd like to be able to do when unit testing, is prevented by design.
- *Collaboration/Integration*—If you wanted to do an integration test of a couple of actors, you'd need to eavesdrop on the actors to assert that the messages have the expected values. It's not immediately clear how this can be done.

Luckily, Akka provides the akka-testkit module. This module contains a number of testing tools that makes testing actors a lot easier. The test kit module makes a couple of different types of tests possible:

- *Single-threaded unit testing*—An actor instance is normally not accessible directly. The test kit provides a `TestActorRef` that allows access to the underlying actor instance. This makes it possible to test the actor instance directly by calling the methods that you’ve defined, or even call the receive function in a single threaded environment, just as you’re used to when testing normal objects.
- *Multithreaded unit testing*—The test kit module provides the `TestKit` and `TestProbe` classes, which make it possible to receive replies from actors, inspect messages, and set timing bounds for particular messages to arrive. `TestKit` has methods to assert expected messages. Actors are run using a normal dispatcher in a multithreaded environment.
- *Multiple JVM testing*—Akka also provides tools for testing multiple JVMs, which comes in handy when you want to test remote actor systems. Multi-JVM testing will be discussed in chapter 6.

`TestKit` has `TestActorRef` extending the `LocalActorRef` class and sets the dispatcher to a `CallingThreadDispatcher` that’s built for testing only. (It invokes the actors on the calling thread instead of on a separate thread.) This provides one of the key junction points for advancing the previously listed solutions.

Depending on your preference, you might use one of the styles more often. The option that’s closest to actually running your code in production is the multithreaded style, testing with the `TestKit` class. We’ll focus more on the multithreaded approach to testing, since this can show problems with the code that won’t be apparent in a single-threaded environment. (You probably won’t be surprised that we also prefer a classical unit testing approach over mocking.)

Before we start, we’ll have to do a little preparation so that we don’t repeat ourselves unnecessarily. Once an actor system is created, it’s started and continues to run until it’s stopped. In all our tests, we need to create actor systems and we have to stop them. To make life easier, let’s build a small trait we can use for all the tests that makes sure that the system under test is automatically stopped when the unit test ends.

Listing 3.1 Stop the system after all tests are done

```
import org.scalatest.{ Suite, BeforeAndAfterAll }
import akka.testkit.TestKit

trait StopSystemAfterAll extends BeforeAndAfterAll {
  this: TestKit with Suite =>
  override protected def afterAll() {
    super.afterAll()
    system.shutdown()
  }
}
```

Extends from the `BeforeAndAfterAll` ScalaTest trait

This trait can only be used if it’s mixed in with a test that uses the `TestKit`.

Shuts down the system provided by the `TestKit` after all tests have executed

We've placed this file in the directory `src/test/scala/aia/testdriven`, because all the test code should be placed within the `src/test/scala` directory, which is the root directory of the all the test code. We'll mixin this trait when we write our tests, so that the system is automatically shut down after all tests are executed. The `TestKit` exposes a `system` value, which can be accessed in the test to create actors and everything else you would like to do with the system.

In the next sections, we'll use the test kit module to test some common scenarios when working with actors, both in a single-threaded and in a multithreaded environment. There are only a few different ways for the actors to interact with each other. We'll explore the different options that are available and test the specific interaction with the test kit module.

3.2 *One-way messages*

Remember, we've left the land of "invoke a function and wait on the response," so the fact that our examples merely send one-way messages with `tell` is deliberate. Given this fire-and-forget style, we don't know when the message arrives at the actor, or even if it arrives, so how do we test this? What we'd like to do is send a message to an actor, and after sending the message, check that the actor has done the work it should've done. An actor that responds to messages should do something with a message and take some kind of action, like send a message to another actor, store some internal state, interact with another object, or interact with I/O. If the actor's behavior is completely invisible from the outside, we can only check if it handled the message without any errors, and we could try to look into the state of the actor with the `TestActorRef`. There are three variations that we'll look at:

- *SilentActor*—An actor's behavior is not directly observable from the outside; it might be an intermediate step that the actor takes to create some internal state. We want to test that the actor at least handled the message and didn't throw any exceptions. We want to be sure that the actor has finished. We want to test the internal state change.
- *SendingActor*—An actor sends a message to another actor (or possibly many actors) after it's done processing the received message. We'll treat the actor as a black box and inspect the message that's sent out in response to the message it received.
- *SideEffectingActor*—An actor receives a message and interacts with a normal object in some kind of way. After we send a message to the actor, we'd like to assert if the object was affected.

We'll write a test for each type of actor in this list that will illustrate the means of verifying results in tests you write.

3.2.1 SilentActor examples

Let's start with the SilentActor. Since it's our first test, let's briefly go through the use of ScalaTest.

Listing 3.2 First test for the silent actor type

```
class SilentActor01Test extends TestKit(ActorSystem("testsystem"))
  with WordSpecLike
  with MustMatchers
  with StopSystemAfterAll {

    "A Silent Actor" must {
      "change state when it receives a message, single threaded" in {
        //Write the test, first fail
        fail("not implemented yet")
      }
      "change state when it receives a message, multi-threaded" in {
        //Write the test, first fail
        fail("not implemented yet")
      }
    }
  }
}
```

Extends from TestKit and provides an actor system for testing

WordSpecLike provides easy-to-read DSL for testing in BDD style

MustMatchers provides easy-to-read assertions

Write tests as textual specifications

Every "in" describes a specific test

Makes sure the system is stopped after all tests

This code is the basic skeleton that we need to start running a silent actor test. We use the `WordSpec` style of testing, which is BDD, since it makes it possible to write the test as a number of textual specifications, which will also be shown when the test is run (the tests are the behavior specification). In the preceding code, we create a specification for the silent actor type with a test that should, as it says, “change internal state when it receives a message.” Right now, it always fails, since it’s not implemented yet—as is expected in *red-green-refactor* style, where you first make sure the test fails (red), then implement the code to make it pass (green), after which you might refactor the code to make it nicer. In the following listing we define an Actor that does nothing, and will always fail the tests.

Listing 3.3 First failing implementation of the silent actor type

```
class SilentActor extends Actor {
  def receive = {
    case msg =>
  }
}
```

Swallows any message; doesn't keep any internal state

To run all the tests at once, run the command `sbt test`. But it's also possible to run only one test. To do this, start `sbt` in the interactive mode and run the `testOnly` command. In the next example, we run the test `aia.testdriven.SilentActor01Test`:

```
sbt
...
> testOnly aia.testdriven.SilentActor01Test
```

Now let's first write the test to send the silent actor a message and check that it changes its internal state. The `SilentActor` actor will have to be written for this test to pass, as well as its *companion* object (an object that has the same name as the actor). The companion object contains the message protocol; that is, all the messages that `SilentActor` supports, which is a nice way of grouping messages that are related to each other, as you'll see later. The following listing is a first pass at this.

Listing 3.4 Single-threaded test internal state

```
"change internal state when it receives a message, single" in {
  import SilentActor._
  val silentActor = TestActorRef[SilentActor]
  silentActor ! SilentMessage("whisper")
  silentActor.underlyingActor.state must (contain("whisper"))
}
```

Creates a `TestActorRef` for single-threaded testing

Imports the messages

Gets the underlying actor and asserts the state

This is the simplest version of the typical TDD scenario: trigger something and check for a state change. Now let's write the `SilentActor` actor. The next listing shows our first version of the actual actor implementation.

Listing 3.5 `SilentActor` implementation

```
object SilentActor {
  case class SilentMessage(data: String)
  case class GetState(receiver: ActorRef)
}

class SilentActor extends Actor {
  import SilentActor._
  var internalState = Vector[String]()

  def receive = {
    case SilentMessage(data) =>
      internalState = internalState :+ data
  }

  def state = internalState
}
```

A companion object that keeps related messages together

The message type that the `SilentActor` can process

State is kept in a vector; every message is added to this vector

State method returns the built-up vector

Since the returned list is immutable, the test can't change the list and cause problems when asserting the expected result. It's completely safe to set/update the `internalState` var, since the Actor is protected from multithreaded access. In general, it's good practice to prefer vars in combination with immutable data structures, instead of vals in combination with mutable data structures. (This prevents accidentally sharing mutable state if you send the internal state in some way to another actor.)

Now let's look at the multithreaded version of this test. As you'll see, we'll have to change the code for the actor a bit as well. Just like in the single-threaded version where we added a state method to make it possible to test the actor, we'll have to add some code to make the multithreaded version testable. The following listing shows how we make this work.

Listing 3.6 Multithreaded test of internal state

```

A companion object that keeps  
related messages together
    "change internal state when it receives a message, multi" in {
      import SilentActor._

      val silentActor = system.actorOf(Props[SilentActor], "s3")
      silentActor ! SilentMessage("whisper1")
      silentActor ! SilentMessage("whisper2")
      silentActor ! GetState(testActor)
      expectMsg(Vector("whisper1", "whisper2"))
    }
  }

```

Test system is used to create an actor →

← **Message is added to the companion to get state**

← **Used to check what message(s) have been sent to the testActor**

The multithreaded test uses the ActorSystem that's part of the TestKit to create a SilentActor actor.

An actor is always created from a Props object. The Props object describes how the actor should be created. The simplest way to create a Props is to create it with the actor type as its type argument, in this case `Props[SilentActor]`. A Props created this way will eventually create the actor using its default constructor.

Since we now can't just access the actor instance when using the multithreaded actor system, we'll have to come up with another way to see state change. For this a `GetState` message is added, which takes an `ActorRef`. The TestKit has a `testActor`, which you can use to receive messages that you expect. The `GetState` method we added is so we can have our `SilentActor` send its internal state there. That way we can call the `expectMsg` method, which expects one message to be sent to the `testActor` and asserts the message; in this case it's a `Vector` with all the data fields in it.

Timeout settings for the expectMsg* methods

The TestKit has several versions of the `expectMsg` and other methods for asserting messages. All of these methods expect a message within a certain amount of time; otherwise, they time out and throw an exception. The timeout has a default value that can be set in the configuration using the `akka.test.single-expect-default` key.

(continued)

A *dilation factor* is used to calculate the actual time that should be used for the timeout (it's normally set to 1, which means the timeout is not diluted). Its purpose is to provide a means of leveling machines that can have vastly different computing capabilities. On a slower machine, we should be prepared to wait a bit longer (it's common for developers to run tests on their fast workstations, and then commit and have slower continuous integration servers fail). Each machine can be configured with the factor needed to achieve a successful test run (check out chapter 7 for more details on configuration). The max timeout can also be set on the method directly, but it's better to just use the configured values, and change the values across tests in the configuration if necessary.

Now all we need is the code for the silent actor that can also process `GetState` messages.

Listing 3.7 SilentActor implementation

```
object SilentActor {
  case class SilentMessage(data: String)
  case class GetState(receiver: ActorRef)
}

class SilentActor extends Actor {
  import SilentActor._
  var internalState = Vector[String]()

  def receive = {
    case SilentMessage(data) =>
      internalState = internalState :+ data
    case GetState(receiver) => receiver ! internalState
  }
}
```

GetState message is added for testing purposes

Internal state is sent to ActorRef in GetState message

The internal state is sent back to the `ActorRef` in the `GetState` message, which in this case will be the `testActor`. Since the internal state is an immutable `Vector`, this is completely safe. This is it for the `SilentActor` types: single- and multithreaded variants. Using these approaches, we can construct tests that are familiar to most programmers: state changes can be inspected and asserted upon by leveraging a few tools in the `TestKit`.

3.2.2 SendingActor example

It's common for an actor to take an `ActorRef` through a `props` method, which it will use at a later stage to send messages to. In this example we'll build a `SendingActor` that sorts lists of events and sends the sorted lists to a receiver actor.

Listing 3.8 Sending actor test

```

"A Sending Actor" must {
  "send a message to another actor when it has finished processing" in {
    import SendingActor._
    val props = SendingActor.props(testActor)
    val sendingActor = system.actorOf(props, "sendingActor")

    val size = 1000
    val maxInclusive = 100000

    def randomEvents() = (0 until size).map{ _ =>
      Event(Random.nextInt(maxInclusive))
    }.toVector

    val unsorted = randomEvents()
    val sortEvents = SortEvents(unsorted)
    sendingActor ! sortEvents

    expectMsgPF() {
      case SortedEvents(events) =>
        events.size must be(size)
        unsorted.sortBy(_.id) must be(events)
    }
  }
}

```

Receiver is passed to props method that creates Props; in the test we pass in a testActor

Randomized unsorted list of events is created

testActor should receive a sorted Vector of Events

A `SortEvents` message is sent to the `SendingActor`. The `SortEvents` message contains events that must be sorted. The `SendingActor` should sort the events and send a `SortedEvents` message to a receiver actor. In the test we pass in the `testActor` instead of a real actor that would process the sorted events, which is easily done, since the receiver is just an `ActorRef`. Since the `SortEvents` message contains a random vector of events, we can't use an `expectMsg(msg)`; we can't formulate an exact match for it. In this case we use `expectMsgPF`, which takes a partial function just like the receive of the actor. Here we match the message that was sent to the `testActor`, which should be a `SortedEvents` message containing a sorted vector of `Events`. If we run the test now, it will fail because we haven't implemented the message protocol in `SendingActor`. Let's do that now.

Listing 3.9 SendingActor implementation

```

object SendingActor {
  def props(receiver: ActorRef) =
    Props(new SendingActor(receiver))
  case class Event(id: Long)
  case class SortEvents(unsorted: Vector[Event])
  case class SortedEvents(sorted: Vector[Event])
}

class SendingActor(receiver: ActorRef) extends Actor {
  import SendingActor._
  def receive = {
    case SortEvents(unsorted) =>
      receiver ! SortedEvents(unsorted.sortBy(_.id))
  }
}

```

receiver is passed through the Props to the constructor of the SendingActor; in the test we pass in a testActor.

The SortedEvent message is sent to the SendingActor.

The SortedEvent message is sent to the receiver after the SendingActor has sorted it.

SortEvents and SortedEvents both use an immutable Vector.

We once again create a companion that contains the message protocol. It also contains a `props` method that creates the `Props` for the actor. In this case the actor needs to be passed the actor reference of the receiver, so another variation of `Props` is used.

Calling `Props(arg)` translates to calling the `Props.apply` method, which takes a by-name creator parameter. By-name parameters are evaluated when they're referenced for the first time, so new `SendingActor(receiver)` is only executed once Akka needs to create it. Creating the `Props` in the companion object has the benefit that you can't refer to an actor's internals, in the case where you would need to create an actor from an actor. Using something internal to the actor from the `Props` could lead to race conditions, or it could cause serialization issues if the `Props` itself were used inside a message that needs to be sent across the network. We'll use this recommended practice for creating props as we go along.

The `SendingActor` sorts the unsorted `Vector` using the `sortBy` method, which creates a sorted copy of the vector, which can be safely shared. The `SortedEvents` is sent along to the receiver. Once again, we take advantage of the immutable property of case classes and of the immutable `Vector` data structure.

Let's look at some variations of the `SendingActor` type. Table 3.1 shows some common variations on the theme.

Table 3.1 `SendingActor` types

Actor	Description
<code>MutatingCopyActor</code>	The actor creates a mutated copy and sends the copy to the next actor, which is the case described in this section.
<code>ForwardingActor</code>	The actor forwards the message it receives; it doesn't change it at all.
<code>TransformingActor</code>	The actor creates a different type of message from the message that it receives.
<code>FilteringActor</code>	The actor forwards some messages it receives and discards others.
<code>SequencingActor</code>	The actor creates many messages based on one message it receives and sends the new messages one after the other to another actor.

The `MutatingCopyActor`, `ForwardingActor`, and `TransformingActor` can all be tested in the same way. We can pass in a `testActor` as the next actor to receive messages and use `expectMsg` or `expectMsgPF` to inspect the messages. The `FilteringActor` is different in that it addresses the question of how we can assert that some messages were *not* passed through. The `SequencingActor` needs a similar approach. How can we assert that we received the correct number of messages? The next test will show you how.

Let's write a test for the `FilteringActor`. The `FilteringActor` that we'll build should filter out duplicate events. It will keep a list of the last messages that it has received, and will check each incoming message against this list to find duplicates. (This is comparable to the typical elements of mocking frameworks that allow you to assert on invocations, counts of invocations, and absence.)

Listing 3.10 FilteringActor test

```

"filter out particular messages" in {
  import FilteringActor._
  val props = FilteringActor.props(testActor, 5)
  val filter = system.actorOf(props, "filter-1")
  filter ! Event(1)
  filter ! Event(2)
  filter ! Event(1)
  filter ! Event(3)
  filter ! Event(1)
  filter ! Event(4)
  filter ! Event(5)
  filter ! Event(5)
  filter ! Event(6)
  val eventIds = receiveWhile() {
    case Event(id) if id <= 5 => id
  }
  eventIds must be(List(1, 2, 3, 4, 5))
  expectMsg(Event(6))
}

```

Sends a couple of events, including duplicates

Receives messages until the case statement doesn't match anymore

Asserts that the duplicates aren't in the result

The test uses a `receiveWhile` method to collect the messages that the `testActor` receives until the case statement doesn't match. In the test, the `Event(6)` doesn't match the pattern in the case statement, which defines that all `Events` with an ID less than or equal to 5 will be matched, popping us out of the while loop. The `receiveWhile` method returns the collected items as they're returned in the partial function as a list. Now let's write the `FilteringActor` that will guarantee this part of the specification.

Listing 3.11 FilteringActor implementation

```

object FilteringActor {
  def props(nextActor: ActorRef, bufferSize: Int) =
    Props(new FilteringActor(nextActor, bufferSize))
  case class Event(id: Long)
}

class FilteringActor(nextActor: ActorRef,
                    bufferSize: Int) extends Actor {
  import FilteringActor._
  var lastMessages = Vector[Event]()
  def receive = {
    case msg: Event =>
      if (!lastMessages.contains(msg)) {
        lastMessages = lastMessages :+ msg
        nextActor ! msg
        if (lastMessages.size > bufferSize) {
          // discard the oldest
          lastMessages = lastMessages.tail
        }
      }
  }
}

```

Max size for the buffer is passed into constructor

Event is sent to next actor if it's not found in the buffer

Oldest event in the buffer is discarded when max buffer size is reached

Vector of last messages is kept

This `FilteringActor` keeps a buffer of the last messages that it received in a `Vector` and adds every received message to that buffer if it doesn't already exist in the list. Only messages that aren't in the buffer are sent to the `nextActor`. The oldest message that was received is discarded when a `max bufferSize` is reached to prevent the `lastMessages` list from growing too large and possibly causing us to run out of space.

The `receiveWhile` method can also be used for testing a `SequencingActor`; you could assert that the sequence of messages that's caused by a particular event is as expected. Two methods for asserting messages that might come in handy when you need to assert a number of messages are `ignoreMsg` and `expectNoMsg`. `ignoreMsg` takes a partial function just like the `expectMsgPF` method, only instead of asserting the message, it ignores any message that matches the pattern. This can come in handy if you're not interested in many messages, but only want to assert that particular messages have been sent to the `testActor`. `expectNoMsg` asserts that no message has been sent to the `testActor` for a certain amount of time, which we could have also used in between the sending of duplicate messages in the `FilteringActor` test. The test in the next listing shows an example of using `expectNoMsg`.

Listing 3.12 `FilteringActor` implementation

```
"filter out particular messages using expectNoMsg" in {
  import FilteringActor._
  val props = FilteringActor.props(testActor, 5)
  val filter = system.actorOf(props, "filter-2")
  filter ! Event(1)
  filter ! Event(2)
  expectMsg(Event(1))
  expectMsg(Event(2))
  filter ! Event(1)
  expectNoMsg
  filter ! Event(3)
  expectMsg(Event(3))
  filter ! Event(1)
  expectNoMsg
  filter ! Event(4)
  filter ! Event(5)
  filter ! Event(5)
  expectMsg(Event(4))
  expectMsg(Event(5))
  expectNoMsg()
}
```

Since `expectNoMsg` has to wait for a timeout to be sure that no message was received, this test will run more slowly.

As you've seen, `TestKit` provides a `testActor` that can receive messages, which we can assert with `expectMsg` and other methods. A `TestKit` has only one `testActor`, and since `TestKit` is a class that you need to extend, how would you test an actor that sends messages to more than one actor? The answer is the `TestProbe` class. The `TestProbe` class is much like `TestKit`, only you can use this class without having to

extend from it. Simply create a `TestProbe` with `TestProbe()` and start using it. `TestProbe` will be used often in the tests that we'll write in this book.

3.2.3 SideEffectingActor example

The next listing shows a very simple Greeter actor that prints a greeting according to the message it receives. (It's the actor-based version of a "Hello World" example.)

Listing 3.13 The Greeter actor

```
import akka.actor.{ActorLogging, Actor}

case class Greeting(message: String)

class Greeter extends Actor with ActorLogging {
  def receive = {
    case Greeting(message) => log.info("Hello {}!", message)
  }
}
```

Prints the greeting it receives

The Greeter does just one thing: it receives a message and outputs it to the console. The `SideEffectingActor` allows us to test scenarios such as these: where the effect of the action isn't directly accessible. Though many cases fit this description, this next listing sufficiently illustrates the final means of testing for an expected result.

Listing 3.14 Testing HelloWorld

```
import Greeter01Test._

class Greeter01Test extends TestKit(testSystem)
  with WordSpecLike
  with StopSystemAfterAll {

  "The Greeter" must {
    "say Hello World! when a Greeting(\"World\") is sent to it" in {
      val dispatcherId = CallingThreadDispatcher.Id
      val props = Props[Greeter].withDispatcher(dispatcherId)
      val greeter = system.actorOf(props)
      EventFilter.info(message = "Hello World!",
        occurrences = 1).intercept {
        greeter ! Greeting("World")
      }
    }
  }
}

object Greeter01Test {
  val testSystem = {
    val config = ConfigFactory.parseString(
      """
        akka.loggers = [akka.testkit.TestEventListener]
      """)
    ActorSystem("testsystem", config)
  }
}
```

Single-threaded environment

Uses the testSystem from the Greeter01Test object

Intercepts the log messages that were logged

Creates a system with a configuration that attaches a test event listener

The Greeter is tested by inspecting the log messages that it writes using the Actor-Logging trait. The test kit module provides a `TestEventListener` that you can configure to handle all events that are logged. The `ConfigFactory` can parse a configuration file from a `String`; in this case we only override the event handlers list.

The test is run in a single-threaded environment because we want to check that the log event has been recorded by the `TestEventListener` when the Greeter is sent the “World” Greeting. We use an `EventFilter` object, which can be used to filter log messages. In this case we filter out the expected message, which should only occur once. The filter is applied when the intercept code block is executed, which is when we send the message.

The preceding example of testing a `SideEffectingActor` shows that asserting some interactions can get complex quickly. In many situations, it’s easier to adapt the code a bit so that it’s easier to test. Clearly, if we pass the listeners to the class under test, we don’t have to do any configuration or filtering; we’ll simply get each message our Actor under test produces. The following listing shows an adapted Greeter actor that can be configured to send a message to a *listener* actor whenever a greeting is logged.

Listing 3.15 Simplifying testing of the Greeter Actor with a listener

```
object Greeter02 {
  def props(listener: Option[ActorRef] = None) =
    Props(new Greeter02(listener))
}
class Greeter02(listener: Option[ActorRef])
  extends Actor with ActorLogging {
  def receive = {
    case Greeting(who) =>
      val message = "Hello " + who + "!"
      log.info(message)
      listener.foreach(_ ! message)
  }
}
```

Constructor takes an optional listener; default set to None

Optionally sends to the listener

The `Greeter02` actor is adapted so that it takes an `Option[ActorRef]`, which is by default set to `None` in the `props` method. After it successfully logs a message, it sends a message to the listener if the `Option` is not empty. When the actor is used normally without specifying a listener, it runs as usual. The following listing is the updated test for this `Greeter02` actor.

Listing 3.16 Simpler Greeter Actor test

```
class Greeter02Test extends TestKit(ActorSystem("testsystem"))
  with WordSpecLike
  with StopSystemAfterAll {

  "The Greeter" must {
    "say Hello World! when a Greeting(\"World\") is sent to it" in {
      val props = Greeter02.props(Some(testActor))
    }
  }
```

Sets the listener to the testActor

```

    val greeter = system.actorOf(props, "greeter02-1")
    greeter ! Greeting("World")
    expectMsg("Hello World!")
  }
  "say something else and see what happens" in {
    val props = Greeter02.props(Some(testActor))
    val greeter = system.actorOf(props, "greeter02-2")
    system.eventStream.subscribe(testActor, classOf[UnhandledMessage])
    greeter ! "World"
    expectMsg(UnhandledMessage("World", system.deadLetters, greeter))
  }
}

```

← Asserts the message as usual

As you can see, the test has been greatly simplified. We simply pass in a `Some(testActor)` to the `Greeter02` constructor, and assert the message that's sent to the `testActor` as usual.

In the next section we'll look at two-way messages, and how these can be tested.

3.3 Two-way messages

You've already seen an example of two-way messages in the multithreaded test for the `SendingActor` style actor, where we used a `GetState` message that contained an `ActorRef`. We simply called the `!` operator on this `ActorRef` to respond to the `GetState` request. As shown before, the `tell` method has an implicit sender reference.

In this test we'll use the `ImplicitSender` trait. This trait changes the implicit sender in the test to the actor reference of the test kit. The following listing shows how the trait is mixed in.

Listing 3.17 ImplicitSender

```

class EchoActorTest extends TestKit(ActorSystem("testsystem"))
  with WordSpecLike
  with ImplicitSender
  with StopSystemAfterAll {

```

← Sets the implicit sender to the TestKit its actor reference

Two-way messages are easy to test in a black box fashion: a request should result in a response, which you can simply assert. In the following test, we'll test an `EchoActor`, an actor that echoes any request back in a response.

Listing 3.18 Testing echoes

```

"Reply with the same message it receives without ask" in {
  val echo = system.actorOf(Props[EchoActor], "echo2")
  echo ! "some message"
  expectMsg("some message")
}

```

← Sends a message to the actor


← Asserts the message as usual

We just send the message, and the `EchoActor` will send the response back to the actor reference of the test kit, which was set automatically as the sender by the `ImplicitSender` trait. The `EchoActor` stays exactly the same. It just sends a message back to the sender. The next listing shows this.

Listing 3.19 `EchoActor`

```
class EchoActor extends Actor {  
  def receive = {  
    case msg =>  
      sender() ! msg  
  }  
}
```

Whatever is received is simply
sent back to (implicit) sender



The `EchoActor` reacts exactly the same way whether the `ask` pattern was used or the `tell` method; the preceding is the preferred way to test two-way messages.

Our journey in this section has taken us through actor-testing idioms that are offered by Akka's `TestKit`. They all serve the same goal: making it easy to write unit tests that need access to results that can be asserted on. The `TestKit` provides methods for both single-threaded and multithreaded testing. We can even “cheat” a little and get at the underlying actor instance during testing. Categorizing actors by how they interact with others gives us a template for how to test the actor, which was shown for the `SilentActor`, `SendingActor`, and `SideEffectingActor` types. In most cases the easiest way to test an actor is to pass a `testActor` reference to it, which can be used to assert expectations on the messages that are sent out by the actor under test. The `testActor` can be used to take the place of a sender in a request-response, or it can just act like the next actor that an actor is sending messages to. Finally, you saw that in many cases it makes sense to prepare an actor for testing, especially if the actor is “silent,” in which case it's beneficial to add an optional listener to the actor.

3.4 *Summary*

Test-driven development is more than a quality control mechanism; it's a way of working. Akka was designed to support TDD. Since the bedrock of regular unit testing is to invoke a method and get a response that can be checked for an expected result, we had to look, in this chapter, for ways to adopt a new mindset to go along with our message-based, asynchronous style.

Actors also bring some new powers to the seasoned TDD programmer:

- Actors embody behavior; tests are fundamentally a means of checking behavior.
- Message-based tests are cleaner: only immutable state goes back and forth, precluding the possibility of tests corrupting the state they're testing.
- With an understanding of the core test actors, you can now write unit tests of actors of all kinds.

This chapter was an introduction to Akka's way of testing, and the tools that Akka provides. The real proof of their value lies in the chapters ahead as we use these to achieve the promise of TDD: rapid development of tested, working code.

In the next chapter we'll look at how actor hierarchies are formed and how supervision strategies and lifecycle monitoring can be used to build fault-tolerant systems.

Akka IN ACTION

Roestenburg • Bakker • Williams



Akka makes it relatively easy to build applications in the cloud or on devices with many cores that efficiently use the full capacity of the computing power available. It's a toolkit that provides an actor programming model, a runtime, and required support tools for building scalable applications.

Akka in Action shows you how to build message-oriented systems with Akka. This comprehensive, hands-on tutorial introduces each concept with a working example. You'll start with the big picture of how Akka works, and then quickly build and deploy a fully functional REST service out of actors. You'll explore test-driven development and deploying and scaling fault-tolerant systems. After mastering the basics, you'll discover how to model immutable messages, implement domain models, and apply techniques like event sourcing and CQRS. You'll also find a tutorial on building streaming applications using akka-stream and akka-http. Finally, you'll get practical advice on how to customize and extend your Akka system.

What's Inside

- Getting concurrency right
- Testing and performance tuning
- Clustered and cloud-based applications
- Covers Akka version 2.4

This book assumes that you're comfortable with Java and Scala. No prior experience with Akka required.

A software craftsman and architect, **Raymond Roestenburg** is an Akka committer. **Rob Bakker** specializes in concurrent back-end systems and systems integration. **Rob Williams** has more than 20 years of product development experience.

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

“The most readable and up-to-date treatment of Akka I have seen.”

—Kevin Esler
TimeTrade Systems

“A great way to get started and go beyond the basics with Akka.”

—Andy Hicks
London Scala Users Group

“A user's guide to Akka in the real world!”

—William E. Wheeler
TEKsystems

“A really useful book. Every chapter has working, real-world code that illustrates how to get things done using Akka.”

—Iain Starks
Game Account Network

ISBN-13: 978-1-61729-101-2
ISBN-10: 1-61729-101-3



9 781617 291012



MANNING

\$49.99 / Can \$57.99 [INCLUDING eBook]