AKKA.NET SUCCINCTLY

BY **ZORAN MAKSIMOVIC**

SUCCINCTLY E-BOOK SERIES

Syncfusion®

Akka.NET Succinctly

By
Zoran Maksimovic

Foreword by Daniel Jebaraj

Technical Reviewer: James McCaffrey

Copy Editor: Courtney Wright

Acquisitions Coordinator: Hillary Bowling, online marketing manager, Syncfusion, Inc.

Proofreader: Jacqueline Bieringer, content producer, Syncfusion, Inc.

Table of Contents

The Story Behind the Succinctly Series of Books 9

About the Author 11

Introduction 12

Purpose 12

Target audience 12

Additional information and resources 12

Source code 12

Akka.NET groups and communities 13

Software requirements 13

Conventions used in the book 13

Source code 13

Resources 14

Akka.NET version 14

Chapter 1 Introduction 15

Short history of actor model and Akka implementation 16

Actor model deconstructed 16

Why use Akka.NET? 17

No manual thread management 17

High(er) level of abstraction 17

Scaling up 18

Scaling out 18

Fault tolerance and fault handling 18

Common framework 18

Where to use Akka.NET? 18

The Reactive Manifesto 19
Responsive 19
Resilient 20
Elastic 20
Message-driven 20
Conclusion 20
Chapter 2 Akka.NET Components 21
Akka (Core) library 21
Akka.TestKit 22
Akka.Remote 22
Akka.Cluster 22
Akka.Streams 23
Akka.Persistence 23
Chapter 3 Introduction to Actors 25
Actors 25
Actor lifecycle 26
ActorSystem 26
Actor reference 27
Chapter 4 Working with Actors 28
Type of actors 28
UntypedActor 28
ReceiveActor 28
Actor instantiation 29
ReceiveActor construction 30
Receive 31
ReceiveAsync 32

ReceiveAny 33

IActorRef..35

Messages and immutability..37

Dependency injection 37

First Akka.NET application 41

Example using Tell 41

Example using Ask 43

Serialization..46

Conclusion 47

Chapter 5 Actor Lifecycle and States 48

Actor's lifecycle...48

Example 50

Actor termination 52

Stopping an actor..53

Sending the PoisonPill message...53

Sending the Kill message 54

Gracefully stopping the actor 55

Restarting an actor 56

Chapter 6 The Actor's Switchable Behavior 59

Music player example...59

Chapter 7 Actor Hierarchies...63

Creating actor hierarchies 63

Chapter 8 Actor Path and Actor Selection 69

Actor path...69

What is the difference between actor reference and path? 71

Actor selector 71

Chapter 9 Supervision 76

What is supervision? 76

Actor supervision hierarchy 77

User guardian 77

System guardian 78

Root guardian 78

Supervision strategy 78

Chapter 10 Other Components 82

EventBus 82

DeadLetters 84

How can we monitor DeadLetters? 85

Chapter 11 Unit Testing Akka.NET 87

Direct testing 89

UnitTesting 90

Testing an actor’s message replies 91

Integration testing 93

Chapter 12 Akka.NET Routers 96

Round Robin 96

Random routing 99

Shortest MailBox queue 100

Consistent Hashing 101

Chapter 13 Actors in ASP.NET Core 103

Visual Studio project creation 103

Actor definition 105

ActorSystem integration 105

Controller definition 106

Chapter 14 Akka.NET Remoting....111

Remoting overview....111

Transport....112

Remoting demo application....112

Akka.NET common library....114

Akka.NET server....115

Akka.NET client....117

The Story Behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn’t everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors’ tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

About the Author

Zoran Maksimovic is a Solution Architect and Software Developer with almost 20 years of professional experience. He is passionate about programming and web platforms, especially on Microsoft technologies. Among others, his specialties and interests focus on: Microsoft.NET, OOD, TDD, DDD (Domain Driven Development), CQRS/ES, and event streaming. He is also a certified Scrum Master.

He spent most of his professional life working as a consultant on various projects for clients in the financial sector based in Switzerland, Italy, Germany, and France.

In his spare time, he is contributing to his personal blog and active on Twitter as @zoranmax.

Zoran is also the author of the ServiceStack Succinctly and MongoDB 3 Succinctly e-books.

When not coding or spending time with his family, he enjoys playing guitar, baroque music, good food, and Italian wine. He is married and the father of Alexei, Xenia, and Sofia.

For more info visit https://zoran.me.

Introduction

Purpose

The purpose of this book is to make you aware of Akka.NET framework and to let you start working with this technology in the fastest possible way.

My hope is that after reading this e-book, you will have enough knowledge to start coding and using Akka.NET effectively.

Target audience

This book is intended for software developers and readers with a good understanding of object-oriented programming, the Microsoft.NET Framework (all of the examples are written in C#), and the usage of the asynchronous execution in .NET. Knowledge of multithreading is also nice to have.

You should be already familiar with ASP.NET Core, Microsoft Visual Studio, and Microsoft.NET in general, as those concepts are not fully explained here.

Some shortcuts have been taken in this e-book, as the intention is not to go too deep into the details—but rather to show the various options, possibilities, and concepts.

Additional information and resources

A lot of additional information about Akka.NET can be found directly on the Akka.NET website.

If you want to know more about the technologies mentioned in this book, take a look at the following resources:

- C# Language
- Distributed Computing
- Reactive Manifesto

Source code

Akka.NET is an open-source framework written in C#, and at the time of writing it's hosted on GitHub here.

Akka.NET groups and communities

There are several groups on the web where additional information is available. The following list contains a few that you might find useful:

- Official Akka.NET website
- Akka.NET Bootcamp: Provides good code examples.
- Akka (Official Java version): Contains good documentation that also applies to Akka.NET.
- Official Twitter account
- StackOverflow: Contains questions and answers about Akka.NET.

Software requirements

To get the most out of this book and the included examples, you will need to have a version of Microsoft Visual Studio IDE installed on your computer, and at least Microsoft.NET v4.

At the time of writing, the most current and stable edition of Visual Studio available is **Visual Studio 2017**. You can download **Visual Studio Community Edition 2017** for free directly from the Microsoft website. Please pay attention to the licensing notes, as some restrictions might apply, depending on the usage.

In addition, there are quite a few examples that use **Microsoft .NET Core v2**, which is also required. You can install .NET Core here.

All of the examples in this book have been written and tested on **Microsoft Windows 10** using **Microsoft Visual Studio 2017 Community Edition** and **Visual Studio 2017 Professional**.

Conventions used in the book

There are specific formats that you will see throughout this book to illustrate tips and tricks or other important concepts.

Note: This will identify things to note throughout the book.

Tip: This will identify tips and tricks throughout the book.

Source code

Source code is written in a consistent manner. Command Prompt (terminal) code follows the following style:

Code Listing 1: Command prompt code style

```
C:\>command
```

Most of the code examples are written in C#, and the following style is used when working with C#.

Code Listing 2: C# code style

```
[SomeAttribute]
public class Actor
{
    public string Property { get; set; }
}
```

Resources

The code mentioned in this book can be downloaded here.

Akka.NET version

All of the examples and explanations apply to the version of Akka.NET v.1.3.2, which is the latest stable version at the time of writing.

Chapter 1 Introduction

Millions of people are accessing all kinds of Internet services. We are living in an era where there is a need for highly **responsive** and **robust** applications to serve exact user needs. The typical user is affected by a large amount of information, and because of a proliferation of services of all kinds all across the Internet, the user's attention span is very short, and a few seconds of unresponsive systems might mean that an item won't be sold, or a blog article won't be read.

At the same time, the current technology allows us to have interconnected systems, grids of servers, data centers, cloud infrastructure, and multi-core machines. The computing power of a CPU, GPU, RAM, and HDD space costs have dropped substantially. This practically enables the processing of the Internet of Things (IoT), online streams of data, big data, machine learning, and artificial intelligence, among others.

When building applications, we can now effectively concentrate on making systems in a different way compared to the past. Today we can effectively utilize resources to work together and constitute a single application platform. This obviously brings a lot of new opportunities, and with them some challenges—as usually nothing comes for "free."

We are living in a new world full of opportunities given by the technological advance. Let's take a look at a very high level of technology evolution to understand better what is going on:

Table 1: Technology evolution

Past	**Present**
One server	Multiple servers, cloud systems
Few users	Sometimes millions of (concurrent) users
One CPU core	Multiple CPU cores
Very limited RAM	Potentially nearly unlimited and cheap RAM
Very limited HDD space	Potentially unlimited and cheap HDD space
Slow and expensive network	Fast and cheap expandable network
Few megabytes of data	Large data sets, big data management

Everything got **bigger**, **distributed**, and **parallelized**!

While there are several approaches to implementing a multithreaded, concurrent, scalable, and distributed system, the actor model paradigm is recently (re)emerging. As we are going to see, this is not a new approach, but certainly a good one that is worth exploring. This book is fully dedicated to Akka.NET, an actor model framework written exclusively for Microsoft.NET.

Short history of actor model and Akka implementation

Let’s start from the very beginning. If you are very new to the actor model, you might be surprised that it is not a new concept—it originated in 1973 with the computer scientists Carl Hewitt, Peter Bishop, and Richard Steiger. The concept has been captured in a publication called “A Universal Modular Actor Formalism for Artificial Intelligence IJCAI'73.”

This paper proposes a modular actor architecture and definitional method for artificial intelligence that is conceptually based on a single kind of object: *actors* (or, if you will, virtual processors, activation frames, or streams). The actor model on its own is based upon a mathematical model of a concurrent computation.

Fast forward to the current days: The Akka.NET framework is an implementation of an actor model. It was introduced in 2014 by Aaron Stannard and Roger Alsing. Akka.NET itself is a port of the Java/Scala Akka framework, which was introduced earlier on in 2009.

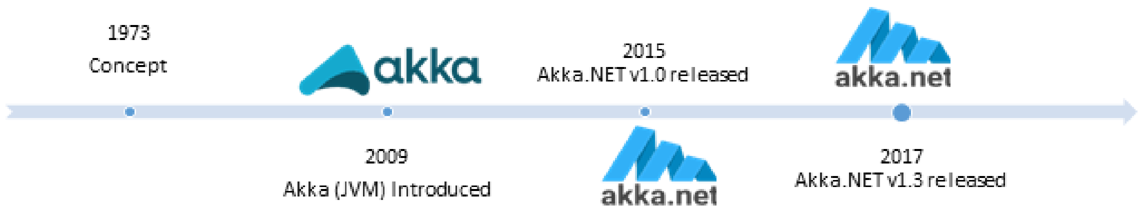


Figure 1: Actor model short history

Akka.NET is a community-driven port, and is not affiliated with the Lightbend company, which made the original Java/Scala version. The first version of Akka.NET was released in 2015. The latest version (1.3) is .NET Core-compatible.

Actor model deconstructed

Programming with the actor model is not only about Akka.NET, which is just one implementation of it.

There are other programming languages, such as Erlang, Elixir, Pony, and Scala, that have been written around the idea of the actor model and, in addition to these, quite a few frameworks are available.

Microsoft alone has recently made quite a contribution, with two implementations of frameworks, Microsoft Orleans and Service Fabric, and it supports actors on Microsoft Azure. On the other hand, an alternative implementation of the actor model from one of the creators of Akka.NET is Proto.Actor, which supports multiple platforms (.NET, Go, JavaScript, and Kotlin).

We can say that the actor model is more like a programming paradigm than a set of tools.

To implement the actor model, there are some fundamental rules to follow:

- Actor contains all the computations.
- Actors can communicate only through messages.

- Actors can change their state or behavior.
- Actors can send messages to other actors.
- Actors can create a finite number of child actors.

Akka.NET fully adheres to these rules.

Why use Akka.NET?

Akka.NET simplifies the building of scalable, concurrent, high-throughput, and low-latency systems. Let’s look at some of the ways Akka.NET makes the life of software developers a bit easier.

No manual thread management

Anyone who’s been involved in writing a multithreading system knows how difficult it can be to write, debug, and test an application. With Akka.NET, we can now safely rely upon the framework to handle the multithreading capabilities and make sure the code is thread safe.

High(er) level of abstraction

Akka.NET offers a higher level of abstraction, which makes everything in the system be considered an *actor*. The framework itself is built upon enabling communication between the actors through message passing.

On the surface, actors look a lot like the objects in object-oriented programming (OOP). If we think about it, OOP is first and foremost about message passing and abstraction. The actor model is all about independent actors maintaining state and passing messages to each other. This sounds very similar, right?

The traditional OOP languages (C#, Java, etc.) weren't designed with concurrency as a first-class use case. While they do support the ability to spawn multiple threads, anyone who's done multithreaded programming with these languages knows how easy it is to introduce race conditions (for example, data desynchronization or corruption). From that perspective, it would be really useful to compare the two paradigms when it comes to thread management and memory access:

Actors are following the ***share-nothing*** *philosophy, while typically the OOP languages are not truly built around parallelism.*

In the actor model, there are no race conditions, which results in simpler code. The share-nothing philosophy of actors also means that the actors don’t have to live on the same machine—we can spawn actors on different nodes and make them collaborate “natively.”

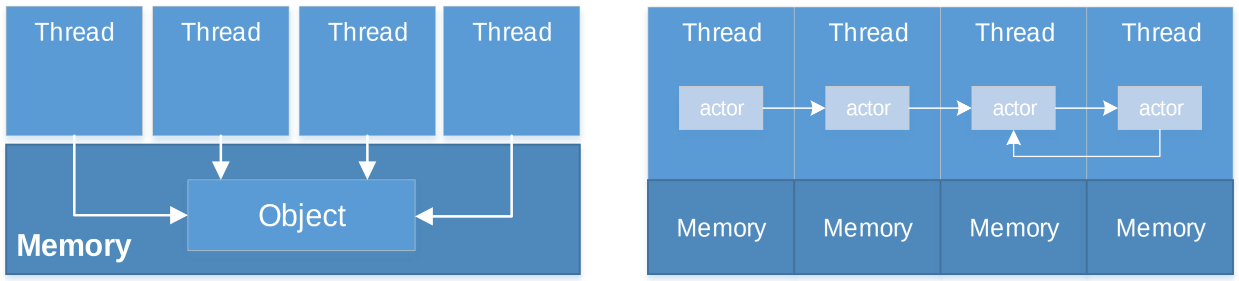


Figure 2: Comparison between the OOP vs actor-model approaches to multithreading

Scaling up

Because the Akka.NET framework is managing concurrency for us, we are able to **scale up** the servers on which the application is running. By adding additional CPUs, the system can have an effective usage of those shared resources without us writing any additional code.

Scaling out

The Akka.NET framework is effectively helping to scale out as well, due to its share-nothing philosophy, as mentioned earlier. By adding additional nodes, once again without us writing any particular implementation, and just through some configuration, the application can effectively use those resources. This is a big win, as we can concentrate on delivering value to our customers rather than writing infrastructure code that enables distributed collaboration.

Fault tolerance and fault handling

Akka.NET offers a way to deal with failures. Unlike the classic synchronous model where each consumer of a component needs to deal with component failures, Akka.NET and the asynchronous model direct failures to dedicated fault handlers. This brings a very controlled way of propagating errors.

Common framework

All of the above-mentioned features make Akka.NET a framework that has a lot of capabilities. Having it all available in one place makes it easier for teams to handle the various complex aspects in a single framework, and makes the learning easier than having to master several technologies to achieve the same result.

Where to use Akka.NET?

Akka.NET and its actor-model framework are practical for use in all kind of scenarios:

- **Transactional application**: Financial, statistical, social media, telecoms, etc.

- **Batch processing**: Actors would allow dividing the workload between them.
- **Services:** REST, SOAP, and general communication services like chats or real-time notifications.

When it comes to the Microsoft.NET framework, we can practically use the framework in every imaginable way:

- As an application backed in a service application (WCF, ASP.NET Web API, etc.).
- As an application front-end in WPF or Windows Forms, used for routing messages to the underlying Akka.NET back-end (or similar).
- In web applications on ASP.NET MVC and Web Forms, used in a similar way as mentioned previously.
- In Windows Services to handle all sorts of messaging.
- In console applications.

The Reactive Manifesto

Akka.NET adheres to the principles defined by the Reactive Manifesto. The Reactive Manifesto proposes a coherent approach to systems architecture in order to build systems that are more robust, more resilient, more flexible, and better positioned to meet modern demands.

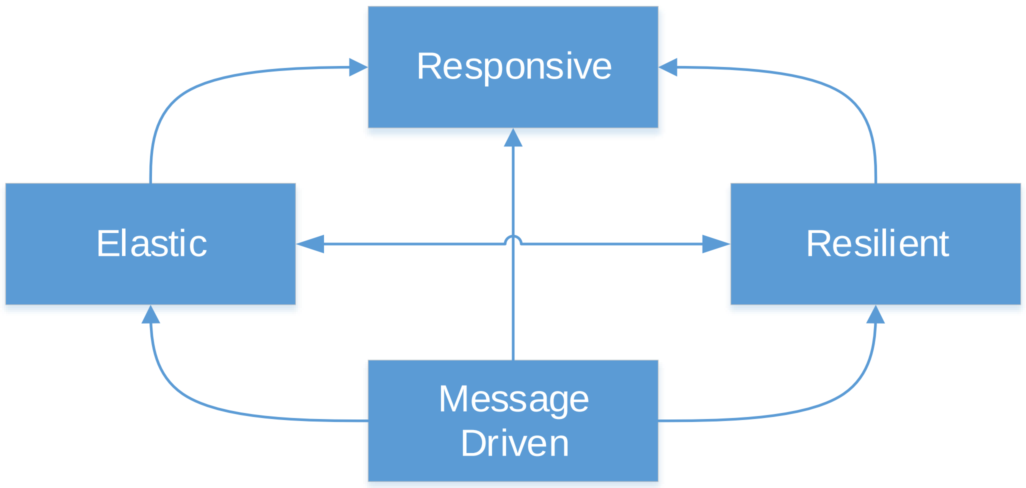


Figure 3: Reactive Manifesto

As you can see in Figure 3, the Reactive Manifesto urges systems to have the following characteristics.

Responsive

The system responds in a timely manner, and it focuses on providing rapid and consistent response times so that the quality of service is constant. As a benefit of such a system, the error-handling is simplified, meets the user expectations, and encourages further interaction.

Resilient

In general, “resilience is the ability to provide required capability in the face of adversity,” as defined by the International Council on Systems Engineering, and in software-engineering terms, this would mean that the system stays responsive in the face of failure. There are several solutions that would help the system to stay resilient—such as replication, containment, isolation, and delegation. These are all in the context of how to handle the failures by not affecting the system as a whole, and how to recover from them successfully.

Elastic

An elastic system is able to adapt to workload changes by provisioning and deprovisioning resources in an autonomic manner, such that at each point in time, the available resources match the current demand as closely as possible. This implies designs that have no contention points or central bottlenecks, resulting in the ability to shard or replicate components and distribute inputs among them.

Message-driven

Reactive systems rely on asynchronous message-passing in order to ensure loose coupling among components, isolation, and location transparency. In turn, the system can be adapted easier (in order to become elastic). Another important aspect is the location transparency of components, which assumes the caller should not worry about the physical location of the target.

Conclusion

This chapter was about the theoretical aspects of the actor model, its history, and various aspects of a reactive system. While entire books can be written on some of these topics, the idea was to give some basic information and context to what this book is about.

Chapter 2 Akka.NET Components

In this chapter, we are going to list the main building blocks of Akka.NET. This includes the libraries (modules) and some key concepts used by the framework.

In order to write any application, we have to rely upon the dependency on the Akka.NET framework. Akka.NET comes as a set of very modular libraries that compose quite a wide range of functionalities. All of the libraries are available through NuGet, as we are going to see later on.

A part of the **`Akka(Core)`**, **`Akka.Remote`**, and **`Akka.TestKit`**, the other components won't be discussed in this book; however, it's still good to know about them.

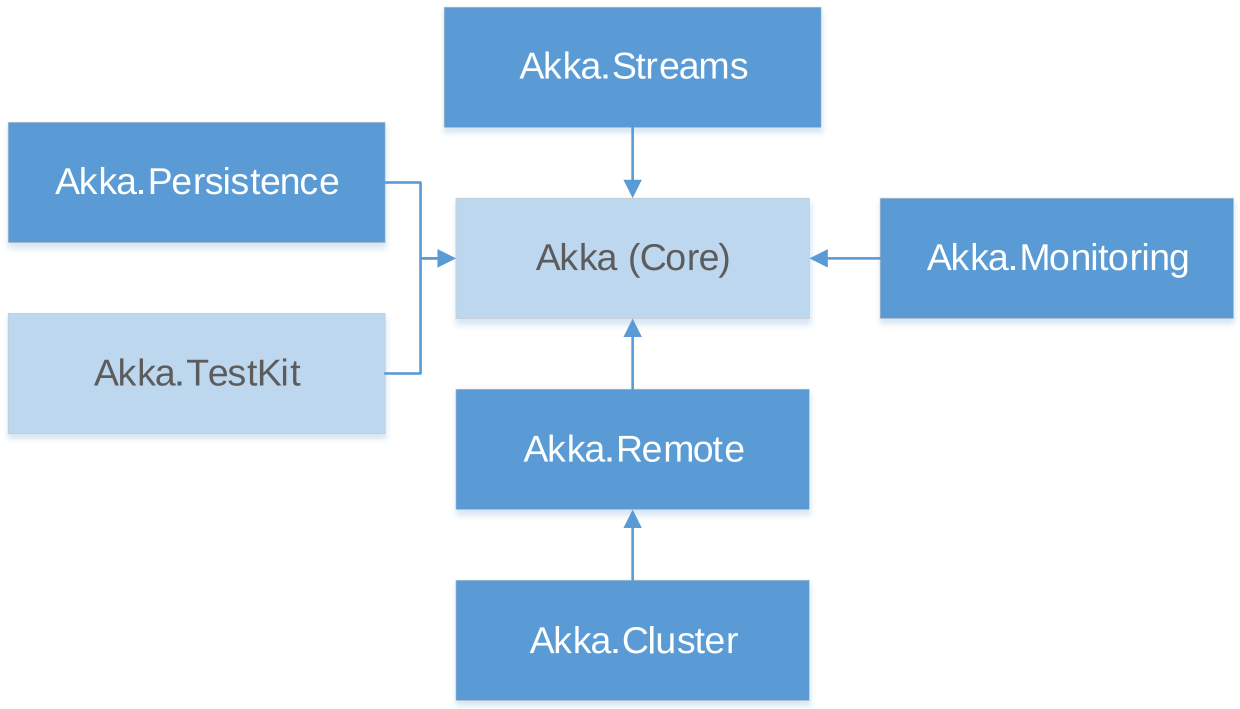


Figure 4: Akka.NET components

Akka (Core) library

As you can see in Figure 4, the Akka.NET core library is the base library upon which all the other components depend.

It contains the definition of an actor, the default object serialization mechanism, routing rules, Human-Optimized Config Object Notation (HOCON) configuration rules, the message dispatching mechanism, and much more. In order to use Akka.NET, we have to have a dependency on this library.

The following command can be run in the Visual Studio Package Manager to reference the Akka base library in Visual Studio:

Code Listing 3: Installing Akka.NET via Package Manager in Visual Studio

```
PM> Install-Package Akka
```

Akka.TestKit

`Akka.TestKit` is a base library with building blocks that allow the effective testing of an actor system. **`Akka.TestKit`** defines the core libraries, and every unit-testing framework will implement the plumbing needed to translate the unit-testing engine's specific needs.

We will discuss unit testing in Chapter 11.

Akka.Remote

`Akka.Remote` brings the capability to build an **`ActorSystem`** across multiple processes over a computer network. Akka.NET supports communication between actors deployed remotely (living on different servers), with the great advantage that the programming model hides this complexity. Communication from the programming view is not different from the local versus remote, as the code would look exactly the same. Usually the **`Akka.Remote`** package is not used in isolation, but as a building block for the clustering capability.

Remoting enables the following functionalities:

- Referencing (individual) actors or actor systems on a remote host.
- Messaging between the two actor systems (local and remote). This involves managing low-level aspects of network (re)connections.

To reference the library in Visual Studio, the following command can be run in the Visual Studio Package Manager:

Code Listing 4: Installing Akka.Remote via Package Manager in Visual Studio

```
PM> Install-Package Akka.Remote
```

The installation of **`Akka.Remote`** will automatically include a reference to the Akka (core) library and **`DotNetty`** (an event-driven asynchronous network application framework), which is used as the transport mechanism.

Akka.Cluster

While **`Akka.Remoting`** solves the problem of addressing and communicating with components on remote systems, clustering gives the ability to organize a number of **`ActorSystems`** to behave as a "single unit," which enables the *scalability* and *high availability* of the system itself.

Clustering provides additional services on top of remoting, such as:

- Handling the remote systems so that they can communicate with each other in a reliable way.
- Handling the change in the server's setup: new cluster memberships, removal (failure) of servers, etc.
- Detecting disconnected systems that are temporarily unreachable.
- Distributing the computation between members (scaling out).

To reference the library in Visual Studio, the following command can be run in the Visual Studio Package Manager:

Code Listing 5: Installing Akka.Cluster package via Package Manager in Visual Studio

```
PM> Install-Package Akka.Cluster
```

The installation of **`Akka.Cluster`** will automatically include a reference to the **`Akka`** (core) library, **`DotNetty`**, and **`Akka.Remote`**.

Akka.Streams

Sometimes actors are not the most suitable tool when it comes to processing a stream (unlimited) of data. In this sense, **`Akka.Streams`** builds on top of actors and provides a higher level of abstraction. **`Akka.Streams`** is also an implementation of the Reactive Streams standard.

Streams implement the following:

- Handling streams of events or large datasets, keeping the proper usage of performance and resources.
- Flexible pipelines in order to reuse functionalities.
- Enabling consumption of Reactive Streams compliant interfaces of other, third-type libraries.

To reference the library in the Visual Studio solution, run the following in the Visual Studio Package Manager:

Code Listing 6: Installing Akka.Streams via Package Manager in Visual Studio

```
PM> Install-Package Akka.Streams
```

The installation of **`Akka.Streams`** will automatically include a reference to the **`Akka`** (core) library and **`Reactive.Streams`** library.

Akka.Persistence

Persistence provides a means to enable actors to persist their current state. There are several libraries that implement persisting data to various systems, such as Microsoft SQL Server, MySql, and Redis.

Persistence resolves the following:

- How to restore the state of an entity/actor when system restarts or crashes.
- Implementation of a system following a CQRS/Event Sourcing pattern.
- Reliable delivery of messages.

To reference the library in Visual Studio, run the following in the Visual Studio Package Manager:

Code Listing 7: Installing Akka.Persistence via Package Manager in Visual Studio

```
PM> Install-Package Akka.Persistence
```

Additionally, there are a number of other packages that have a specific implementation for a given database system:

Code Listing 8: Installing Akka.Persistence packages via Package Manager in Visual Studio

```
PM> Install-Package Akka.Persistence.SqlServer
PM> Install-Package Akka.Persistence.PosgreeSql
PM> Install-Package Akka.Persistence.SqlLite
PM> Install-Package Akka.Persistence.MySql
```

Chapter 3 Introduction to Actors

Actors

We have mentioned several times the existence of an actor, the most basic feature and one of the main building blocks of Akka.NET. An actor encapsulates an object's *behavior* and *state,* and communicates with other actors by exchanging messages. Every instance of an actor has its own **`MailBox`** where the messages will be enqueued and successively processed by the actor, one by one, respecting the order of messages.

By using actors, we have simple and high-level abstractions for concurrency and parallelism. This is due to the fact that every time an actor processes a message, this might happen on another thread.

Tip: To better visualize the idea of an actor, let's think of an actor as a real person. This person can distribute (delegate) the work to other people, or simply fail at processing certain messages.

Actors are built and organized around a *hierarchical* structure in order to split the tasks into smaller and more manageable pieces. If we think about it, this is very similar to object-oriented programming where we have classes and functions in order to split or organize some larger logic into a smaller subset of building blocks. This also implies that actors are able to *create* and *supervise* subactors and control their lifetimes: creation and termination. This also implies that every actor has *only one* supervisor.

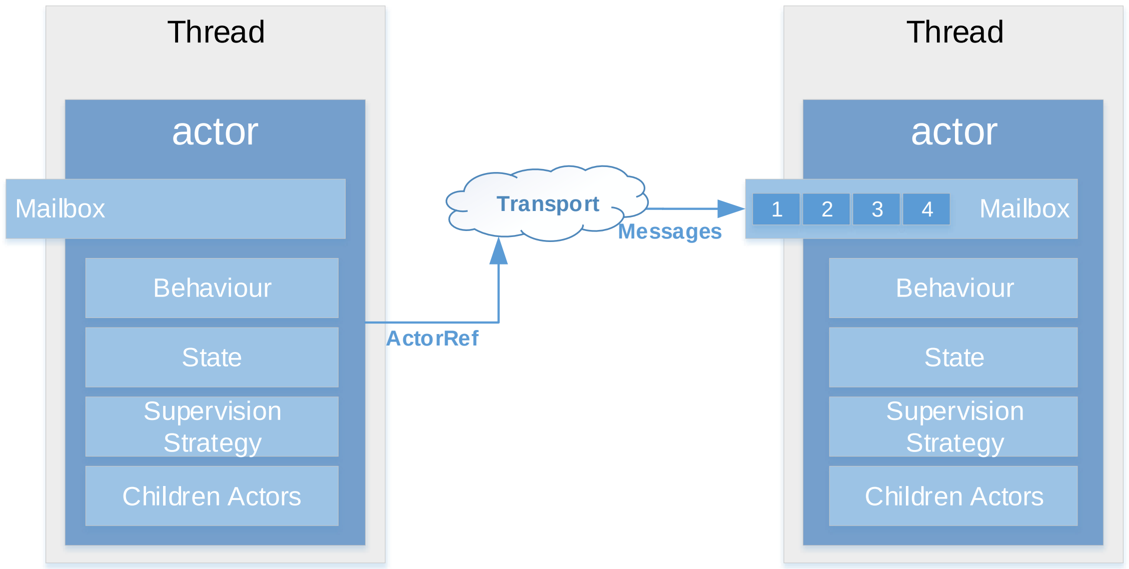


Figure 5: Actor's communication by message passing

Actor lifecycle

Every actor has its own lifecycle, which exists in the moment when the actor is created, is running, and is terminated. Even though creating a new instance of an actor is relatively cheap in terms of memory and processing, a proper handling of the life of an actor should be planned.

ActorSystem

One of the **`ActorSystem`** responsibilities is to manage the resources in order to *run* the actors. An **`ActorSystem`** is a hierarchical group of actors that share common configuration, for example, dispatchers, deployments, thread pooling, remote capabilities, and addresses. An actor cannot exist outside the **`ActorSystem`**; therefore, we can also say that an **`ActorSystem`** is a container or a host of actors.

An **`ActorSystem`** is not cheap to create, it's typically created only once per application, even though multiple systems can run in the same application on top of .NET runtime. Once it's up and running, it will usually be there for the whole lifetime of the application, or at least until it's not needed (terminated). In that case, the different actor systems won't have anything in common and would live in separate memory spaces, threads, etc.

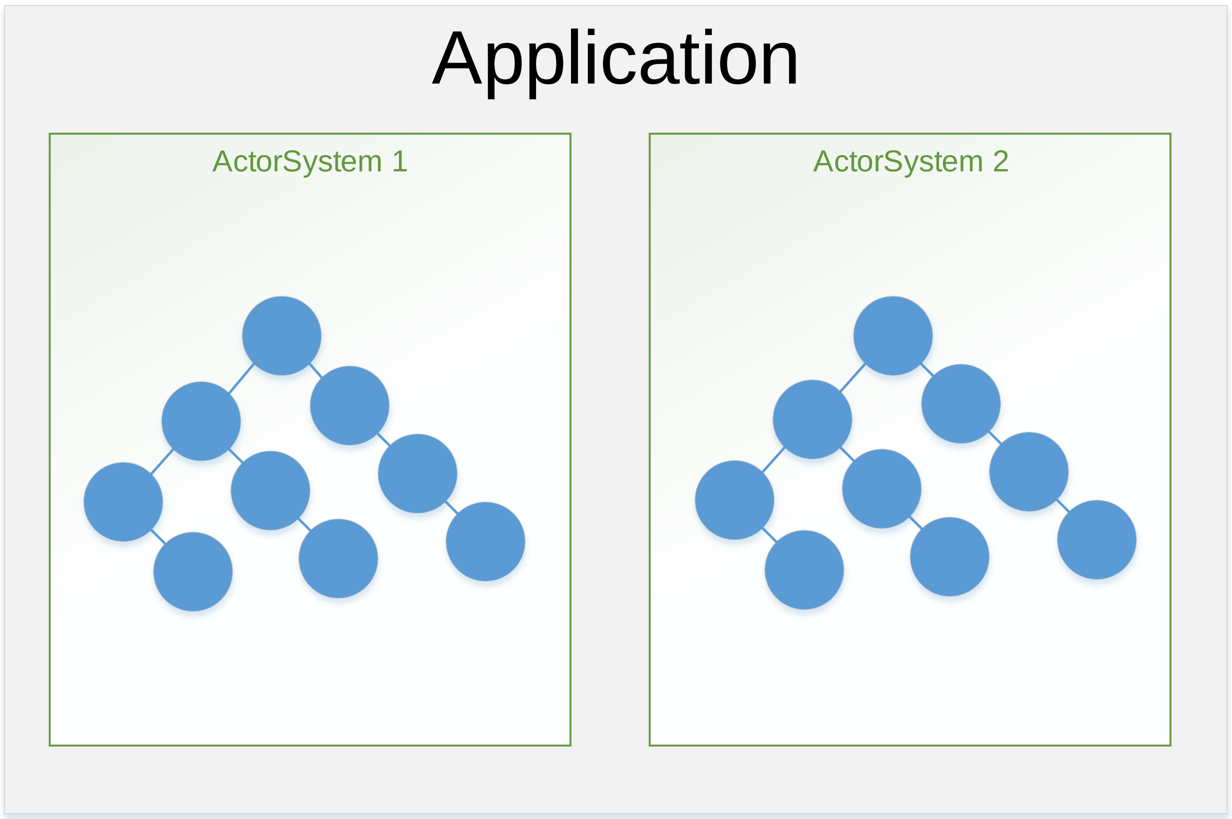


Figure 6: Multiple actor systems within one application

An **`ActorSystem`** handles the following aspects:

- Creates and allocates threads, so that the actors can use the underlying threads.
- Hosts the configuration.

Actor reference

As stated previously, an actor has a state, behavior, mailbox, child actors, and supervision strategy. All of this is encapsulated behind an *actor reference*.

In an actor system, actors are referenced by actor references, which are like pointers to an instance of an actor. The actor reference enables actors to have a reference to another actor (send messages); however, this is unrelated to the actual state (lifecycle) of the actor itself. Removing a reference to an actor doesn't "destroy" that actor, and vice versa.

The actor reference, on the other hand, makes possible the so-called location transparency of actors, which means that an actor can be deployed on a remote system, but from the actor-reference perspective (client calling the actor), nothing changes in the programming model.

Phone calls are a good example to explain the location transparency: We have the phone number (of a person), so sending an SMS or placing a phone call is always possible, where ever the other person is located.

An actor reference can be passed around as a variable. An actor through actor references always has the ability to access the child actors, self, sender, and its parent actor.

Chapter 4 Working with Actors

In the first three chapters, we mentioned some basic building blocks and the theory around actor model programming and Akka.NET. In this chapter, we will finally start implementing the mentioned concepts, starting from the very simple aspects, and end up building a live system.

Type of actors

Akka.NET offers two kinds of actor base types: **`ReceiveActor`** and **`UntypedActor`**. The difference between the two, as we are going to see, is the signature for receiving messages. One is strongly typed, while the other is not. The usage of one or the other would most probably depend on the use case and handling of types of incoming messages.

An actor in Akka.NET is implemented as a class that inherits from one of the aforementioned base types.

UntypedActor

As shown in Code Listing 9, implementing an actor is quite straightforward. In the case of an **`UntypedActor`**, we have to override the **`OnReceive`** method (please note that the **`OnReceive`** method accepts a generic **`object`**). The **`OnReceive`** method will be the entry point for all the messages sent to the actor, and the implementation of this method should handle all kinds of message types that are expected.

Code Listing 9: UntypedActor declaration

```
public class MyUntypedActor : UntypedActor
{
    protected override void OnReceive(object message)
    {
        /* some code goes here */
    }
}
```

ReceiveActor

On the other hand, the **`ReceiveActor`** has a bit of a different structure. There is no **`OnReceive`** method to be overridden anymore, and the mapping to the message handlers is done directly in the actor's constructor. The mapping is done through one of the many **`Receive`** methods to which we have to specify the type and the actual handler.

Code Listing 10: ReceiveActor example

```
public class MyTypedActor : ReceiveActor
```

```
{
    public MyTypedActor()
    {
        /* some code goes here */
    }
}
```

Actor instantiation

Creating a new instance of an **`actor`** can only be done through the **`ActorSystem`**. The **`ActorSystem`** offers two methods: **`ActorOf`** and **`ActorOf<T>`**, which are responsible for the actor's creation.

In order to be identified, an **`ActorSystem`** has to have a name. This is mainly needed in order to ensure the actor's location, as we are going to see later on. In our case, the name of the **`ActorSystem`** is **`my-first-akka`**. The name can be any string.

By using the generic version of the **`ActorOf`** method, we can only specify the name of the actor. The name is optional, and if it's not specified, one will be assigned automatically by the system. Usually the name given is in the form of "**`$a`**", "**`$b`**", etc.

Code Listing 11: Creation of an Actor through ActorSystem by using generic ActorOf<T>

```
static void Main(string[] args)
{
    ActorSystem system = ActorSystem.Create("my-first-akka");

    IActorRef typedActor = system.ActorOf<MyTypedActor>();
    IActorRef untypedActor = system.ActorOf<MyUntypedActor>();
}
```

Note: The two `IActorRef` objects: `typedActor` and `untypedActor`, are actor references, through which we can communicate with the actor.

The alternative to **`ActorOf`**'s generic version is to use the nongeneric version, which accepts different parameters: we have to supply a parameter of type **`Props`**. **`Props`** is a configuration class to specify **`options`** for the creation of actors. By using **`Props`**, we can define a **`SupervisionStrategy`**, create the actor by its constructor attributes, or specify a **`Factory`** for it. Let's see an example of using **`Props`** when creating our previously defined actors.

Code Listing 12: Instantiating actors using props

```
static void Main(string[] args)
{
    ActorSystem system = ActorSystem.Create("my-first-akka");

    Props typedActorProps = Akka.Actor.Props.Create<MyTypedActor>();
    Props untypedActorProps = Akka.Actor.Props.Create<MyUntypedActor>();

    IActorRef typedActor = system.ActorOf(typedActorProps);
    IActorRef untypedActor = system.ActorOf(untypedActorProps);
}
```

There are several ways of using **`Props`**, as we can see in the following code snippet.

Code Listing 13: Various ways of using the Create method

```
Props props1 = Props.Create(typeof(MyTypedActor));
Props props2 = Props.Create(() => new MyTypedActor("arg"));
Props props3 = Props.Create<MyTypedActor>();
Props props4 = Props.Create(typeof(MyTypedActor), "arg");
```

ReceiveActor construction

In this book, we will only be using the **`ReceiveActor`**, for the simplicity and as a matter of preference. In this section, we will be looking at the various methods that the **`ReceiveActor`** offers, and will expand on those.

The configuration of message handlers always happens in the actor’s constructor. So, often you will be writing code similar to the following:

Code Listing 14: Handler message methods are declared in the constructor

```
public class MyTypedActor : ReceiveActor
{
    public MyTypedActor()
    {
        Receive<GreetingMessage>(message =>
GreetingMessageHandler(message));
        Receive<string>(message => GreetingMessageHandler(message));
    }
}

public class GreetingMessage
{
    public GreetingMessage(string greeting)
    {
        Greeting = greeting;
    }

    public string Greeting { get; private set;}
}
```

We use the constructor of an actor to define which message types it accepts (**`GreetingMessage`** and **`string`**, in our example).

The **`Receive`** method is defined in the base (**`ReceiveActor`**) class, and it registers a handler (method) for a particular type of message that the actor receives.

It's worth pointing out that there are several **`Receive`** methods available, such as **`Receive`**, **`Receive<T>`**, **`ReceiveAsync<T>`**, **`ReceiveAsync`**, **`ReceiveAny`**, and **`ReceiveAnyAsync`**. Let's go through some of them.

Receive

The **`Receive`** method, being strongly typed, will ensure that the defined message is handled by a predefined handler (method).

Since the **`Receive<T>`** method in question has several overloads, and may accept the **`Func<T, bool>`**, **`Action<T>`**, or **`Predicate<T>`** objects as arguments, it's possible to write either an inline declaration of the code to be executed, or a pointer to an actual method that implements such an interface.

Code Listing 15: Typed Actor declaration

```
public class MyTypedActor : ReceiveActor
{
    public MyTypedActor()
    {
        Receive<string>(message => Console.WriteLine(message));
    }
}
```

Alternatively, you can use another method to handle the message—Code Listing 15 is equivalent to Code Listing 16.

Code Listing 16: Receiving messages using a handler method

```
public class MyTypedActor : ReceiveActor
{
    public MyTypedActor()
    {
        Receive<string>(message => HandleStringMessage(message));
    }

    private void HandleStringMessage(string message)
    {
        Console.WriteLine(message);
    }
}
```

An important thing to note here is that the messages an actor receives will be processed one-by-one in a synchronous manner. This means no new messages will be processed until the **`HandleStringMessage`** has finished processing the current message.

It is not a good practice (and will actually cause issues) to use the **async**/**await** pattern inside a **Receive** method, as this would break the order of messages—the **Receive** method will exit before the actual processing is finished. This is a very important concept, and to handle this scenario, Akka.NET has implemented the **ReceiveAsync** method.

ReceiveAsync

ReceiveAsync<T> and **ReceiveAsync** enable the **async/await** pattern when defining handlers.

Even though the **ReceiveAsync** handles asynchronous messages, it will process *one message at a time* in order to preserve the ordering of messages—so don't expect parallel processing of messages inside an actor by using the **ReceiveAsync**. Getting new messages from the actor's mailbox in this method will therefore behave exactly as the normal **Receive** method: one-by-one, and in an ordered fashion.

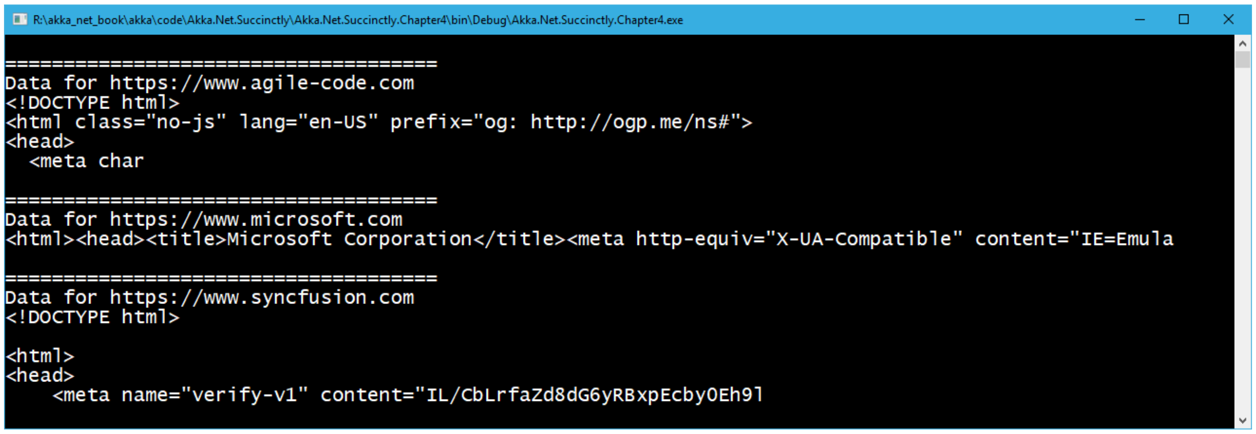
In the following example, we attempt to build an actor (**DownloadHtmlActor**) that downloads data from the Internet in an **asynchronous** fashion by using **DownloadStringTaskAsync**. This method can be awaited, so let's see how we can construct the actor around it.

Code Listing 17: Download HTML via async method

```
public class DownloadHtmlActor : ReceiveActor
{
    public DownloadHtmlActor()
    {
        ReceiveAsync<string>(async url => await GetPageHtmlAsync(url));
    }

    private async Task GetPageHtmlAsync(string url)
    {
        var html = await new
System.Net.WebClient().DownloadStringTaskAsync(url);

        Console.WriteLine("\n====================================");
        Console.WriteLine($"Data for {url}");
        Console.WriteLine(html.Trim().Substring(0, 100));
    }
}

static void Main(string[] args)
{
    ActorSystem system = ActorSystem.Create("html-download-system");

    IActorRef receiveAsyncActor = system.ActorOf<DownloadHtmlActor>("html-
actor ");

    receiveAsyncActor.Tell("https://www.agile-code.com");
    receiveAsyncActor.Tell("https://www.microsoft.com");
    receiveAsyncActor.Tell("https://www.syncfusion.com");
}
```

The first thing to note is that the **ReceiveAsync** signature can now mark the method as an **async** method and **await** it **(async url => await GetPageHtmlAsync(url))**.

When running the code in Code Listing 17, we can clearly see that the order of messages will be maintained.



```
R:\akka_net_book\akka\code\Akka.Net.Succinctly\Akka.Net.Succinctly.Chapter4\bin\Debug\Akka.Net.Succinctly.Chapter4.exe

=====================================
Data for https://www.agile-code.com
<!DOCTYPE html>
<html class="no-js" lang="en-US" prefix="og: http://ogp.me/ns#">
<head>
  <meta char

=====================================
Data for https://www.microsoft.com
<html><head><title>Microsoft Corporation</title><meta http-equiv="X-UA-Compatible" content="IE=Emula

=====================================
Data for https://www.syncfusion.com
<!DOCTYPE html>

<html>
<head>
    <meta name="verify-v1" content="IL/CbLrfaZd8dG6yRBxpEcbyOEh9l
```

Figure 7: Output of invoking the DownloadHtmlActor

ReceiveAny

The **ReceiveAny** and **ReceiveAnyAsync** methods will handle all or any type of message, if the type of a particular message is not already handled by one of the other **Receive** methods. This can be used as a catch-all in case we don't support some messages, in order to have a managed handling of improperly registered messages. So, it is a good practice to put the **ReceiveAny** at the end of the list of **Receive** methods; otherwise, you will get the following exception:

System.InvalidOperationException: `'A handler that catches all messages has been added. No handler can be added after that.'`

Note: ReceiveAny is equivalent to Receive<object>(…).

In Code Listing 18, we can see an example of how to use the **ReceiveAnyAsync** by following the previous example of the HTML page download actor. We can see that now the **GetPageHtmlAsync** method accepts an **object** rather than a **string**.

Code Listing 18: Definition of the DownloadAnyHtmlActor

```
public class DownloadAnyHtmlActor : ReceiveActor
{
    public DownloadAnyHtmlActor()
    {
        ReceiveAnyAsync(async obj => await GetPageHtmlAsync(obj));
    }

    private async Task GetPageHtmlAsync(object obj)
    {
        if (obj is string || obj is Uri)
        {
            var url = obj.ToString();
            var html = await new
System.Net.WebClient().DownloadStringTaskAsync(url);

            Console.WriteLine("\n=====================================");
            Console.WriteLine($"Data for {url}");
            Console.WriteLine(html.Trim().Substring(0, 100));
        }
        else
            throw new ArgumentNullException("Actor doesn't accept this kind
of message");
    }
}
```

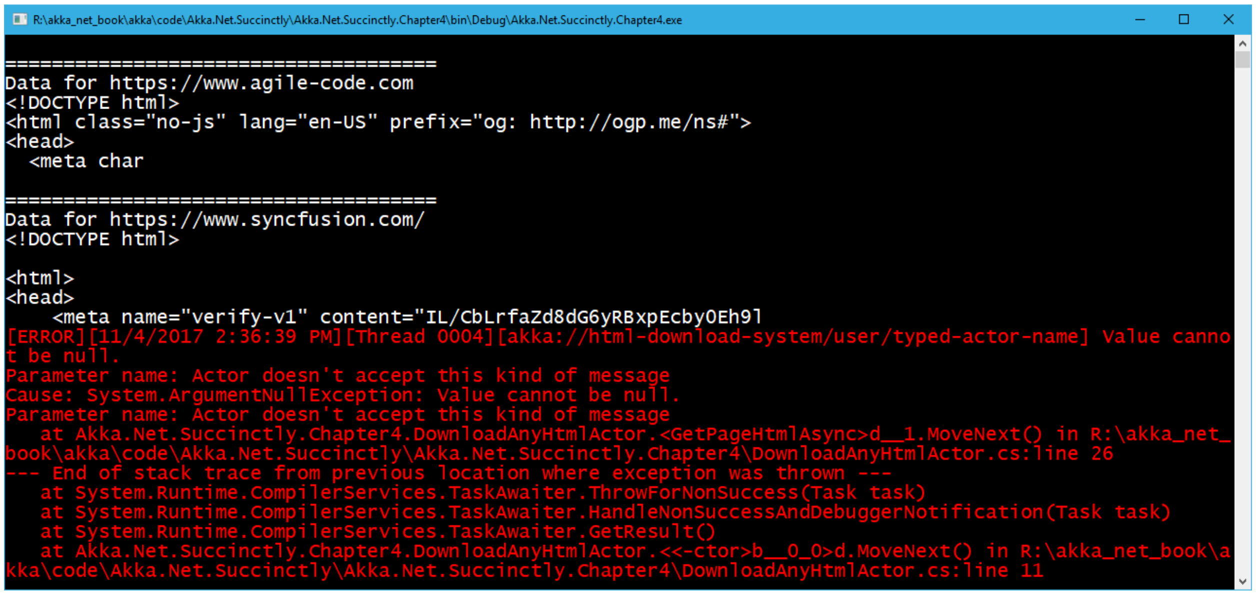
We can also see that the **`GetPageHtmlAsync`** also checks for the type of the object passed. In case of a string, or if a **`System.Uri`** object is passed, the actor would download the page, and otherwise will throw an exception.

Also in the client code, we can clearly see that we pass three types of messages to the actor: a string, a **`System.Uri`**, and a **`GreetingMessage`** (which is not currently being handled).

Code Listing 19: Client code calling DownloadAnyHtmlActor

```
static void Main(string[] args)
{
    ActorSystem system = ActorSystem.Create("html-download-system");

    IActorRef receiveAsyncActor = system.ActorOf<DownloadAnyHtmlActor>();

    receiveAsyncActor.Tell("https://www.agile-code.com");
    receiveAsyncActor.Tell(new Uri("https://www.syncfusion.com"));
    receiveAsyncActor.Tell(new GreetingMessage("hi"));

    Console.Read();
    system.Terminate();
}
```

The output is as we may expect: two successfully downloaded pages, and one exception thrown.



```
R:\akka_net_book\akka\code\Akka.Net.Succinctly\Akka.Net.Succinctly.Chapter4\bin\Debug\Akka.Net.Succinctly.Chapter4.exe
=====================================
Data for https://www.agile-code.com
<!DOCTYPE html>
<html class="no-js" lang="en-US" prefix="og: http://ogp.me/ns#">
<head>
  <meta char
=====================================
Data for https://www.syncfusion.com/
<!DOCTYPE html>

<html>
<head>
    <meta name="verify-v1" content="IL/CbLrfaZd8dG6yRBxpEcby0Eh9l
[ERROR][11/4/2017 2:36:39 PM][Thread 0004][akka://html-download-system/user/typed-actor-name] Value canno
t be null.
Parameter name: Actor doesn't accept this kind of message
Cause: System.ArgumentNullException: Value cannot be null.
Parameter name: Actor doesn't accept this kind of message
   at Akka.Net.Succinctly.Chapter4.DownloadAnyHtmlActor.<GetPageHtmlAsync>d__1.MoveNext() in R:\akka_net_
book\akka\code\Akka.Net.Succinctly\Akka.Net.Succinctly.Chapter4\DownloadAnyHtmlActor.cs:line 26
--- End of stack trace from previous location where exception was thrown ---
   at System.Runtime.CompilerServices.TaskAwaiter.ThrowForNonSuccess(Task task)
   at System.Runtime.CompilerServices.TaskAwaiter.HandleNonSuccessAndDebuggerNotification(Task task)
   at System.Runtime.CompilerServices.TaskAwaiter.GetResult()
   at Akka.Net.Succinctly.Chapter4.DownloadAnyHtmlActor.<<-ctor>b__0_0>d.MoveNext() in R:\akka_net_book\a
kka\code\Akka.Net.Succinctly\Akka.Net.Succinctly.Chapter4\DownloadAnyHtmlActor.cs:line 11
```

Figure 8: Output of DownloadAnyHtmlActor execution

IActorRef

One thing you have probably already noticed is that when we create a new instance of an actor, what is being returned is an implementation of an **`IActorRef`** rather than an instance of the actor itself. We spoke in Chapter 3 about actor references, and this is how they are represented.

`IActorRef` is a *handle* to an actor. Having a reference to an actor guarantees that the actor is alive, or that it existed in the past. Unfortunately, this handle doesn't answer the question of whether or not the actor is still alive.

Among others, it contains three methods that enable communication between actors, being local or remote: **`Ask`**, **`Tell`**, and **`Forward`**. In addition to this, it is possible to see the full **`Path`** of an actor.

Tell

The **`Tell`** method uses the fire-and-forget pattern; using **`Tell`**, one actor simply sends a message to another actor and doesn't wait for any response back, and it returns immediately (it is a nonblocking call).

When we send a message using **`Tell`**, it might seem as if we are doing nothing more than sending a simple plain-old CLR object (POCO). This is not fully the case. When you invoke **`Tell`**, the information about the **`Sender`** of the message tags along. Within a given **`Actor`**, there is an implicit property called **`Context`**. We can use **`Context`** to determine the parent of the **`Actor`**, and **`Context.Parent`**. **`Context.System`** can be used to access the root **`ActorSystem`**, under which the **`Actor`** resides. You can get a reference to the **`Actor`** itself using **`Context.Self`**.

Forward

`Forward` is a special method of **`Tell`**, where the sender information will be carried as part of the message context, which means that even if the message goes through several actors, the original actor that sent the message will be preserved in this context.

Ask

`Ask` is a request/response-based communication pattern: it sends a message to another actor, expecting it to respond with another message, and returns a **`Task`**, asynchronously notifying when the response will come back.

In general, there are performance implications for using **`Ask`**, since under the hood there is quite some work to be done in order to enable the mechanics of mapping the request with the response, etc. So, you should prefer using **`Tell`** for performance, and only use **`Ask`** if you have no other choice.

As an alternative of **`Ask`**, we can always use **`Tell`** or **`Forward`**, and return back the message in a fluent manner, as depicted in Figure 9.

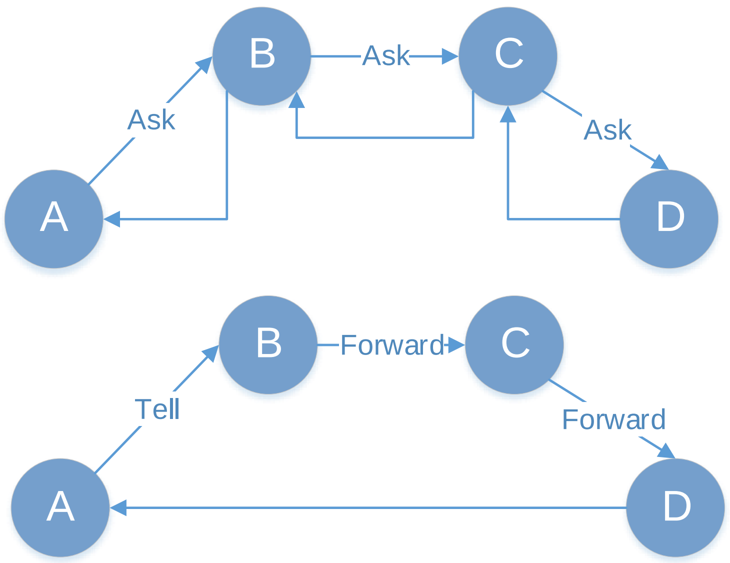


Figure 9: Ask implemented as Tell/Forward

In the first section, we can see that in order to communicate between Actors A and D, we have to block several actors (B and C) in order to get the response back.

In the second part, we can see that by simply passing the message, and maintaining the **`context`**, it is possible to have an asynchronous response by combining **`Tell`** and **`Forward`**.

Messages and immutability

One very important aspect of the data (messages) being passed to actors is that those messages should be *immutable*. Immutable messages are crucial because we can send the same message to many actors concurrently, and if each one of those actors makes a modification to the state of the message, all of those state changes are local to each actor. Obviously, we want to avoid this, so the preferred way of creating messages sent to actors would be to avoid using public **`setters`**, which automatically force us to pass and set properties via constructors.

In this way, we can be completely sure that there will be no side effects if this exact instance of a message is used by multiple actors at the same time. An example of an immutable message is defined in the following code:

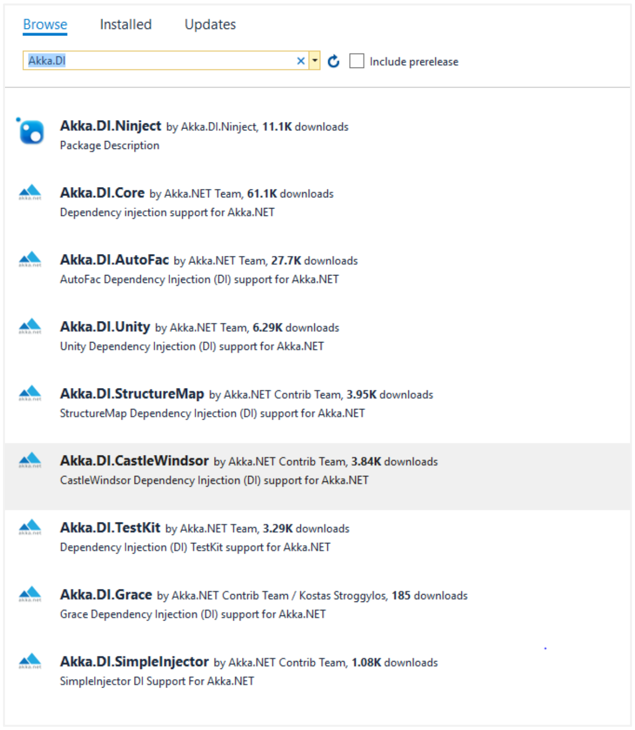
Code Listing 20: Example of an immutable message

```
public class GreetingMessage
{
    public GreetingMessage(string greeting)
    {
        Greeting = greeting;
    }

    public string Greeting { get; private set;}
}
```

Dependency injection

Akka.NET supports dependency injection, which is supported by the **`DependencyResolver`** class that can create **`Props`** using the dependency injection (DI) container.

There are several containers supported out of the box by Akka.NET, and there are libraries ready to be referenced and used. A few examples are Ninject, AutoFac, Microsoft Unity, SimpleInjector, and StructureMap.



Akka.DI.Ninject by Akka.DI.Ninject, **11.1K** downloads
Package Description

Akka.DI.Core by Akka.NET Team, **61.1K** downloads
Dependency injection support for Akka.NET

Akka.DI.AutoFac by Akka.NET Team, **27.7K** downloads
AutoFac Dependency Injection (DI) support for Akka.NET

Akka.DI.Unity by Akka.NET Team, **6.29K** downloads
Unity Dependency Injection (DI) support for Akka.NET

Akka.DI.StructureMap by Akka.NET Contrib Team, **3.95K** downloads
StructureMap Dependency Injection (DI) support for Akka.NET

Akka.DI.CastleWindsor by Akka.NET Contrib Team, **3.84K** downloads
CastleWindsor Dependency Injection (DI) support for Akka.NET

Akka.DI.TestKit by Akka.NET Team, **3.29K** downloads
Dependency Injection (DI) TestKit support for Akka.NET

Akka.DI.Grace by Akka.NET Contrib Team / Kostas Stroggylos, **185** downloads
Grace Dependency Injection (DI) support for Akka.NET

Akka.DI.SimpleInjector by Akka.NET Contrib Team, **1.08K** downloads
SimpleInjector DI Support For Akka.NET

Figure 10: Dependency injection libraries

The steps needed to instantiate an actor via DI are as follows:

1. Instantiate the DI container of preference and register all of the dependencies.
2. Create the actor system (as we always do).
3. Create an instance of the actor by using **`Props`** (as we have seen previously).

In the example we are going to show, we are creating an actor **`MusicActor`** that has **`MusicSongService`** as a dependency. We are keeping things very simple here just to demonstrate the plumbing of the various parts.

First things first. When creating a new Visual Studio project, if we want to use the DI, then we need to reference the library that will enable us to do so. In our example, we are using **`AutoFac`**. Therefore, our project has to install the following NuGet packages:

Code Listing 21: Installing Akka–AutoFac dependencies

```
PM> Install-Package Akka
PM> Install-Package Akka.DI.Core
PM> Install-Package Akka.DI.AutoFac
```

In reality, installing only **`Akka.DI.AutoFac`** would autoinstall the two other libraries and the AutoFac container library itself.
Let’s start with defining the service that our **`MusicActor`** depends upon:

Code Listing 22: MusicSongService definition

```
public interface IMusicSongService
```

```
{
    Song GetSongByName(string songName);
}

public class MusicSongService : IMusicSongService
{
    public Song GetSongByName(string songName)
    {
        return new Song(songName, new byte[0]);
    }
}

public class Song
{
    public Song(string songName, byte[] rowFormat)
    {
        SongName = songName;
        RowFormat = rowFormat;
    }

    public string SongName { get; }
    public byte[] RowFormat { get; }
}
```

Nothing too difficult here: our service **`MusicSongService`** has one method and returns an object of type **`Song`**. To keep the example simple, there is no real implementation of the song retrieval, which isn't really relevant for our example.

The following is the implementation of the actor. In bold, we can see that the **`MusicActor`** requires that an object of type **`IMusicSongService`** gets injected at creation time. Our actor receives a message of type **`string`**, and the **`HandleSongRetrieval`** method will be responsible for using the service to retrieve the song.

Code Listing 23: MusicActor definition

```
public class MusicActor : ReceiveActor
{
    private IMusicSongService SongService { get; }

    public MusicActor(IMusicSongService songService)
    {
        SongService = songService;
        Receive<string>(s => HandleSongRetrieval(s));
    }

    public void HandleSongRetrieval(string songName)
    {
        var song = SongService.GetSongByName(songName);
        /* do something with this song*/
    }
}
```

Now that we have all the moving parts defined, let's see how to inject the **`IMusicSongService`** into the actor upon creation effectively.

Code Listing 24: Client code for DI actor's generation

```
using Akka.Actor;
using Akka.DI;
using Akka.DI.AutoFac;
using Akka.DI.Core;
using Autofac;

static void Main(string[] args)
{
    //1. Create and build the container by registering types.
    var builder = new Autofac.ContainerBuilder();
    builder.RegisterType<MusicSongService>().As<IMusicSongService>();
    builder.RegisterType<MusicActor>().AsSelf();
    var container = builder.Build();

    //2. Create the ActorSystem and Dependency Resolver.
    var system = ActorSystem.Create("MySystem");
    var propsResolver = new AutoFacDependencyResolver(container, system);

    //3. Create an Actor reference.
    IActorRef musicAct = system.ActorOf(system.DI().Props<MusicActor>(),
"MusicActor");

    musicAct.Tell("Bohemian Rhapsody");

    Console.Read();
    system.Terminate();
}
```

We can see that the first part of the program is the creation and configuration of the **`AutoFac`** container system, where we have to register all the dependencies. Don't forget to register the **`MusicActor`**, which is the actor itself. This is no different from what we got used to when working with inversion of control containers. **`ContainerBuilder`** will return the actual container after the **`builder.Build()`** method has been called.

One important thing to notice here is that the **`DependencyResolver`** has to be instantiated and attached to the **`ActorSystem`**. This is done behind the scenes by the **`AutoFacDependencyResolver`**.

`DependencyResolver` will be eventually used by the **`system.DI()`** extension method (for this, we need to include **`Akka.DI.Core`** in the using section). **`system.DI().Props<MusicActor>()`** is equivalent to the **`Props`** we have already seen, with the only difference being that it will use the dependency injection framework to create the actor.

From there on, everything works as with normal actors.

First Akka.NET application

Let's create a console application that will create and instantiate the two actors, and send them a message—the classic "Hello world." Not too exciting, but the simplicity will illustrate the basic principles of how the system works.

In order to do this, let's create a new **`Console Application`** in Visual Studio (it's fine if you want to use the .NET Core or the full .NET, as both are supported by Akka.NET). Don't forget to reference the Akka base library from NuGet by running the following:

Code Listing 25: Install Akka NuGet package

```
PM> Install-Package Akka
```

Example using Tell

The aim of the application is to instantiate an object of type **`GreetingMessage`**, set a greeting, and pass this to the typed and untyped actor. We won't be doing much more than confirming that the message has been received by writing this into the console window.

Code Listing 26: Definition of the GreetingMessage class

```

public class GreetingMessage
{
    public GreetingMessage(string greeting)
    {
        Greeting = greeting;
    }

    public string Greeting { get; }
}
```

Let's create our actors, which will implement a bit more than we have previously seen. Upon receiving a message, we will display some information about the context as well as the message being sent.

Code Listing 27: Definition of the ReceiveActor

```
public class MyTypedActor : ReceiveActor
{
    public MyTypedActor()
    {
        base.Receive<GreetingMessage>(message =>
GreetingMessageHandler(message));

```

```
    }

    private void GreetingMessageHandler(GreetingMessage greeting)
    {
        Console.WriteLine($"Typed Actor named: {Self.Path.Name}");
        Console.WriteLine($"Received a greeting: {greeting.Greeting}");
        Console.WriteLine($"Actor's path: {Self.Path}");
        Console.WriteLine($"Actor is part of the ActorSystem:
{Context.System.Name}");
    }
}
```

The **UntypedActor**’s logic is exactly the same, except that we have to handle the actual message type, as we can get any type of message.

Code Listing 28: UntypedActor example

```
public class MyUntypedActor : UntypedActor
{
    protected override void OnReceive(object message)
    {
        var greeting = message as GreetingMessage;
        if (greeting != null)
        {
            GreetingMessageHandler(greeting);
        }
    }

    private void GreetingMessageHandler(GreetingMessage greeting)
    {
        Console.WriteLine($"Untyped Actor named: {Self.Path.Name}");
        Console.WriteLine($"Received a greeting: {greeting.Greeting}");
        Console.WriteLine($"Actor's path: {Self.Path}");
        Console.WriteLine($"Actor is part of the ActorSystem:
{Context.System.Name}");
    }
}
```

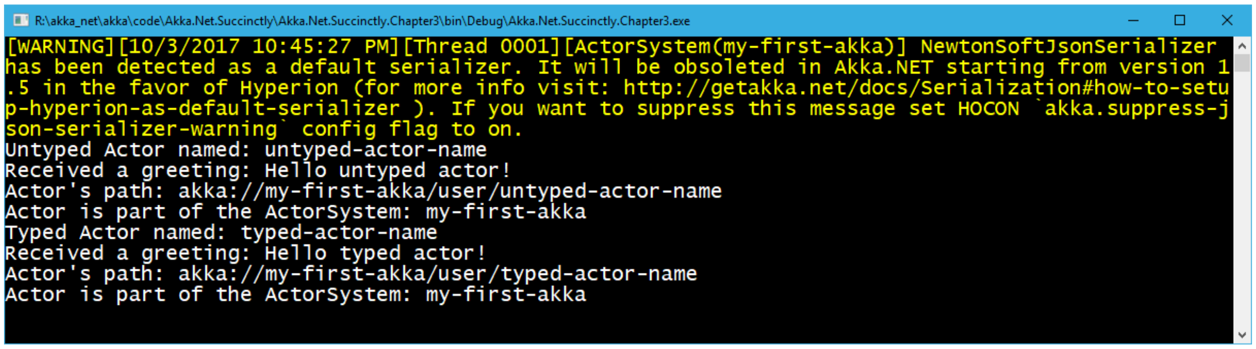
And finally, in Code Listing 29 Listing 29 is the main method that wires everything together by assigning specific names to the actual actors. Please note that we are assigning names, such as **untyped-actor-name** and **typed-actor-name**, just to be able to identify the instance of the actor when the data is displayed in the console windows.

In this method, we can also see the usage of the method **Tell**, where the **GreetingMessage** is being passed as an argument.

Code Listing 29: Main method sending messages to actors

```
static void Main(string[] args)
{
    ActorSystem system = ActorSystem.Create("my-first-akka");

    IActorRef untypedActor = system.ActorOf<MyUntypedActor>("untyped-actor-
name");
    IActorRef typedActor = system.ActorOf<MyTypedActor>("typed-actor-
name");

    untypedActor.Tell(new GreetingMessage("Hello untyped actor!"));
    typedActor.Tell(new GreetingMessage("Hello typed actor!"));

    Console.Read();
}
```

If we run the application, the following output will be shown:



```
[WARNING][10/3/2017 10:45:27 PM][Thread 0001][ActorSystem(my-first-akka)] NewtonSoftJsonSerializer
has been detected as a default serializer. It will be obsoleted in Akka.NET starting from version 1
.5 in the favor of Hyperion (for more info visit: http://getakka.net/docs/Serialization#how-to-setu
p-hyperion-as-default-serializer ). If you want to suppress this message set HOCON `akka.suppress-j
son-serializer-warning` config flag to on.
Untyped Actor named: untyped-actor-name
Received a greeting: Hello untyped actor!
Actor's path: akka://my-first-akka/user/untyped-actor-name
Actor is part of the ActorSystem: my-first-akka
Typed Actor named: typed-actor-name
Received a greeting: Hello typed actor!
Actor's path: akka://my-first-akka/user/typed-actor-name
Actor is part of the ActorSystem: my-first-akka
```

Figure 11: Output of the first application

As we can see, the two actors were able to receive a message, and to display the information about the message received. At the same time, more information about the **Context** has been displayed, too.

Example using Ask

We are going to create a very simple calculator application that will demonstrate the usage of the **Ask** method.

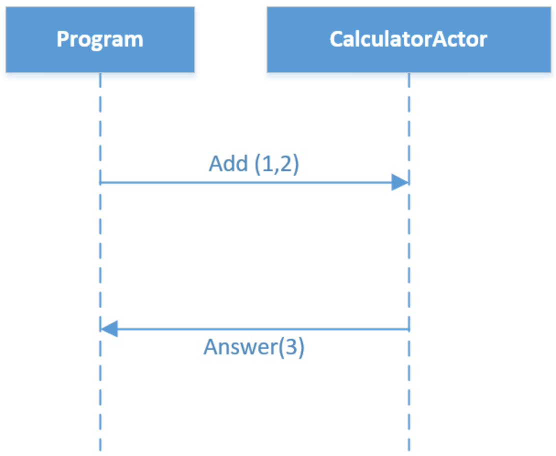


Figure 12: Sequence diagram depicting communication with CalculatorActor

Let's start with the definition of the **`CalculatorActor`** and the two messages that we are going to exchange, **`Add`** and **`Answer`**:

Code Listing 30: CalculatorActor definition

```
public class CalculatorActor : ReceiveActor
{
    public CalculatorActor()
    {
        Receive<Add>(add => Sender.Tell(new Answer(add.Term1 +
add.Term2)));
    }
}

public class Answer
{
    public Answer(double value)
    {
        Value = value;
    }

    public double Value;
}

public class Add
{
    public Add(double term1, double term2)
    {
        Term1 = term1;
        Term2 = term2;
    }

    public double Term1;
    public double Term2;
}
```

We can see that the **`CalculatorActor`** accepts a message of type **`Add`**, and returns an object of type **`Answer`** to the sender. **`Sender`** is a special object that represents the **`producer`** of the original message. The message is returned to the sender via the **`Tell`** method.

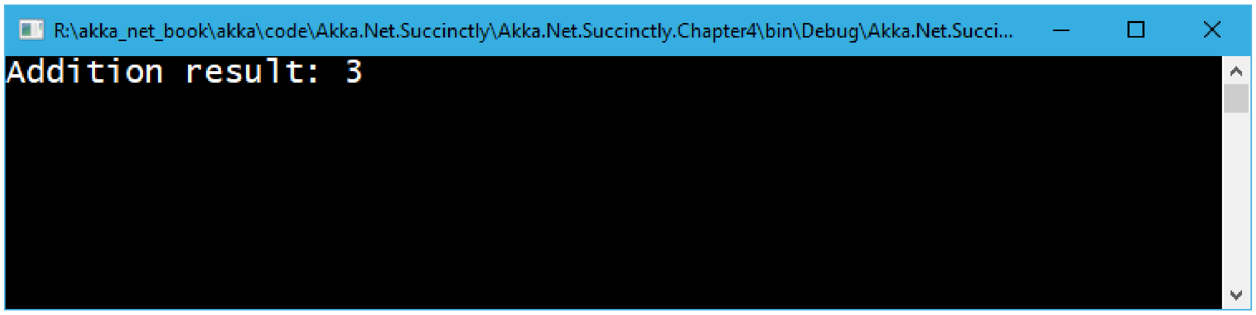
In Code Listing 31, we can see how to send a message to the **`CalculatorActor`**:

Code Listing 31: Sending Messages to the Calculator Actor

```
static void Main(string[] args)
{
    ActorSystem system = ActorSystem.Create("calc-system");

    IActorRef calculator = system.ActorOf<CalculatorActor>("calculator");

    Answer result = calculator.Ask<Answer>(new Add(1,2)).Result;

    Console.WriteLine("Addition result: " + result.Value);
    system.Terminate();
}
```

The main part of this piece of code is the **`calculator.Ask`** method. **`Ask`**, as we have already mentioned, returns a **`Task`** object. In our case, we define the return object as part of the generic signature of the call **`calculator.Ask<Answer>`**. The **`Add`** message is being sent as a parameter to **`Ask`**. In order to return the result, we use **`.Result`**, as we would normally do in .NET. As the **`Main`** method is not marked as **`async`**, we are forced to call **`.Result`**; otherwise, we would use the **`async/await`** pattern.

The output of this call is, without much surprise, as shown in Figure 13.


R:\akka_net_book\akka\code\Akka.Net.Succinctly\Akka.Net.Succinctly.Chapter4\bin\Debug\Akka.Net.Succi...

```
Addition result: 3
```

Figure 13: Calculation result

Serialization

You have probably noticed the warning yellow message at the top of the output. Currently, Akka.NET uses **`NewtonsoftJsonSerializer`** as the default serializer, but the decision has been made to change it in favor of the Hyperion library. Therefore, you may want to install Hyperion from the Visual Studio package manager. Hyperion has some advantages over the **`NewtonsoftJsonSerializer`**.

As the **`url`** provided in the warning didn't work for me, here we can take a look at the basic changes needed to make the warning disappear and, obviously, to set up Hyperion as the default serialization library.

The first thing to do is install the **`Akka.Serialization.Hyperion`** package. This version is still in beta at the time of writing, but is stable enough to be used.

Code Listing 32: Installing Akka.Serialization.Hyperion NuGet package

```
PM> Install-Package Akka.Serialization.Hyperion -pre
```

The second step is to change the **`app.config`** file in order to instruct Akka.NET to use **`Hyperion`** as the default serializer.

For those familiar with .NET configurations, there is nothing fancy about this, apart from the fact that Akka.NET uses the HOCON object model in order to set up properties. For the time being, we won't discuss all of the possible configurations available.

Code Listing 33: HOCON configuration in the app.config

```
<configSections>
  <section name="akka"
type="Akka.Configuration.Hocon.AkkaConfigurationSection, Akka" />
</configSections>

<akka>
  <hocon>
    <![CDATA[
    akka
    {
      actor
      {
          serializers
          {
            hyperion = "Akka.Serialization.HyperionSerializer,
Akka.Serialization.Hyperion"
          }
          serialization-bindings
          {
            "System.Object" = hyperion
          }
      }
    }
    ]]>
  </hocon>
</akka>
```

Conclusion

In this chapter, we have seen how to create actors with or without **`Preps`**. We have also seen the basic building blocks when it comes to the creation and instantiation of an **`ActorSystem`**, created instances of actors, and just scratched the surface of passing a message to an actor.

Even though we didn't get into details about HOCON configuration, we have seen that Akka.NET supports this way of setting up the configuration properties, and that it's quite easy to configure a default serializer.

We've covered quite a lot of concepts in such a small amount of code.

Chapter 5 Actor Lifecycle and States

Actor's lifecycle

An actor has a lifecycle. With the lifecycle, we attend to the actual stages or statuses through which the actor passes: creation, starting, stopping, termination, etc. The actor API offers extension points, so that it is possible to hook into the various stages of the actor's lifecycle, and perform actions.

Let's start with a diagram depicting the various states of an actor's lifecycle:

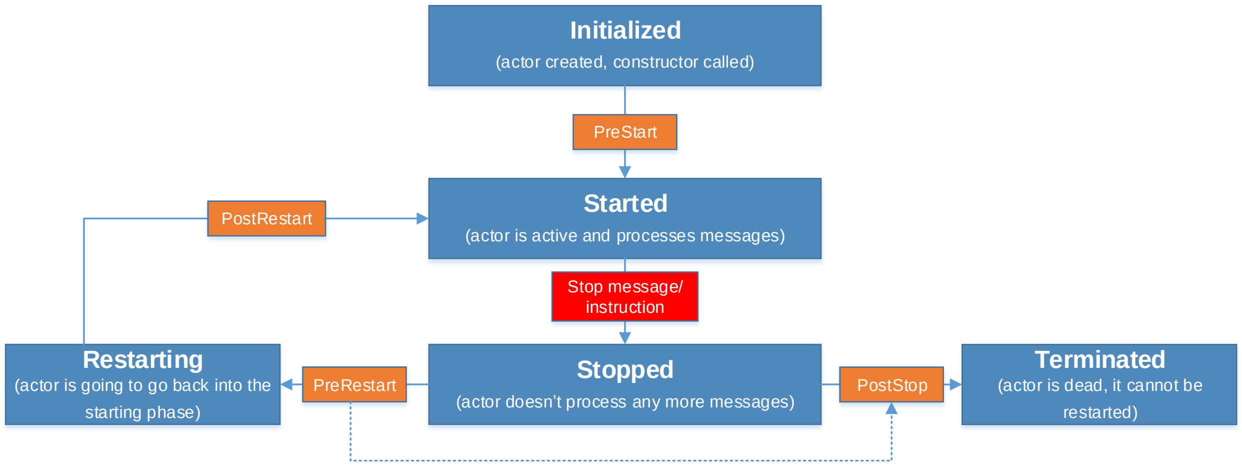

Initialized
(actor created, constructor called)
PreStart
Started
(actor is active and processes messages)
PostRestart
Stop message/ instruction
Restarting
(actor is going to go back into the starting phase)
PreRestart
Stopped
(actor doesn't process any more messages)
PostStop
Terminated
(actor is dead, it cannot be restarted)

Figure 14: Actor LifeCycle

We can see the stages (in blue) and the actual methods executing after that phase (in orange).

Let's go through the various stages, and explain them further.

Table 2: Actor's lifecycle stages

`Initialized`	The actor instance is created and the actor's constructor has been called. At this point no messages are yet received—only after the constructor is executed will the **`PreStart`** be called. **`PreStart`** is a method that enables us to prepare all the necessary work before starting to receive messages.
`Started`	After the **`PreStart`** method has finished executing, the actor is alive and able to receive and process messages from the mailbox, one by one. The actor can be instructed to **`Stop`**, as we are going to see later. The **`stop message`** (or the stop instruction) will lead to the state change, and the actor will go to the next state: **`Stopped`**.

`Stopped`	There are several ways to stop an actor, and two possible outcome scenarios: • Scenario one: The actor **`terminates`**, in which case the **`PostStop`** method will be called. • Scenario two: The actor **`restarts`**, in which case **`PreRestart`** will be called. **`PreRestart`**, as implemented in the base class, will automatically call the **`PostStop`** as the next step. If we want to truly restart the actor, we should not call the base class implementation. In any case, if **`PostStop`** is invoked, there is no way back—the actor is terminated.
`Terminated`	The state in which the actor is not active anymore, and cannot be restarted—in effectƒ, this actor doesn't exist anymore.
`Restarting`	If the parent actor has identified the error, and it doesn't want to terminate the actor, then this actor can be asked to restart. At this point, the actor will be in the restarting state. After this state, the actor is going to become active again, as when originally created. After the **`PreRestart`** has run (as we have seen in the **`Stopped`** phase), just before the actor starts again, we have the ability to implement the **`PostRestart`** method.

How can we use the various Hook methods?

Table 3: Actor's lifecycle methods

`PreStart`	This method is called before the actor starts receiving its first message. Here we can place any kind of custom initialization code. For instance, opening files, database connections, etc.
`PostStop`	This method is called after the actor has been stopped, and is not receiving or processing messages. Here we can place any custom **cleanup** code.
`PreRestart`	This method is called before the actor begins restarting. While **`PreStart`** and **`PostStop`** are not receiving any input parameters, **`PreRestart`** will have available the latest message being processed, and the exception that caused it to restart. One of the reasons for the existence of this method is to save the message being processed when the exception occurs, to be reprocessed later after the actor restarts.

`PostRestart`	**`PostRestart`** gets called after the **`PreRestart`** method, but before the **`PreStart`** method. **`PostRestart`** as an input parameter has the **`exception`** that last occurred, so it allows the code to do something with this **`exception`**: to run some diagnostics or logging, etc.

Example

In the following example, we are going to see how to track the invocation of the actor's constructor, **`PreStart`**, and **`PostStop`** methods.

The idea here is that we have a very trivial **`EmailMessage`** to be sent by using the **`EmailSenderActor`**. **`EmailSenderActor`** will implement the overloads of the Hook methods in order to display in the console what is happening.

Let's define the **`EmailMessage`**, which is the message to be passed to the actor. **`EmailMessage`** is for demonstration purposes only, and doesn't implement anything unusual.

Code Listing 34: Definition of the EmailMessage

```
public class EmailMessage
{
    public EmailMessage(string from, string to, string content)
    {
        From = from;
        To = to;
        Content = content;
    }
    public string From { get; }
    public string To { get; }
    public string Content { get; }
}
```

Let's implement the actual actor, with the overloads, which will be further explained.

Code Listing 35: Definition of the EmailSenderActor

```
public class EmailSenderActor: ReceiveActor
{
    public EmailSenderActor()
    {
        Console.WriteLine("Constructor() -> EmailSenderActor");
        Receive<EmailMessage>(message => HandleEmailMessage(message));
    }

    private void HandleEmailMessage(EmailMessage message)
    {
        Console.WriteLine($"Email sent from {message.From} to 
{message.To}");
    }

    protected override void PreStart()
    {
        Console.WriteLine("PreStart() -> EmailSenderActor");

    }

    protected override void PreRestart(Exception reason, object message)
    {
        Console.WriteLine("PreRestart() -> EmailSenderActor");
        /* base.PreRestart(reason, message); */
    }

    protected override void PostRestart(Exception reason)
    {
        Console.WriteLine("PostRestart() -> EmailSenderActor");
        base.PostRestart(reason);
    }

    protected override void PostStop()
    {
        Console.WriteLine("PostStop() -> EmailSenderActor");
    }
}
```

As you can see, we have simply implemented the overridden methods, which we have discussed previously, and the only thing they do is display a message in the console to show that those have been executed.

An interesting point to note is the **`PreRestart`** implementation. If we call **`base.PreRestart`**, this method will automatically stop all the child actors and call **`PostStop`**. Code Listing 36 shows the actual implementation of the base method we have overridden.

Code Listing 36: PreRestart base class implementation

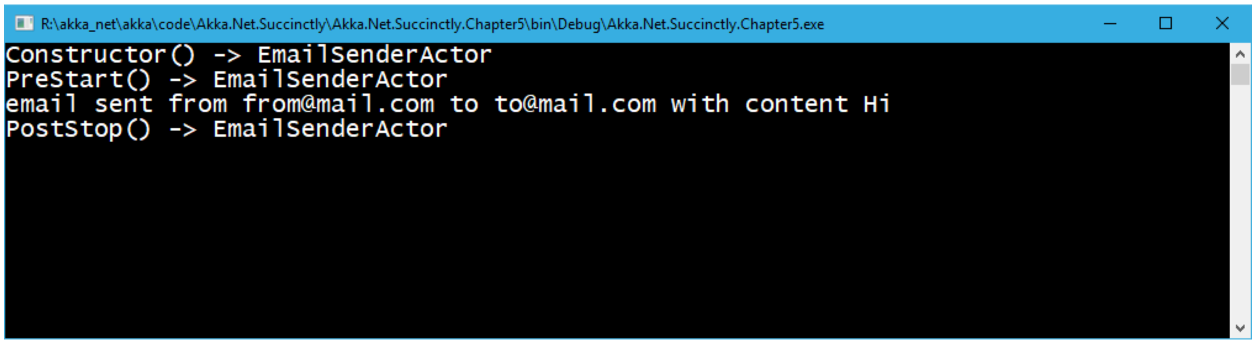
```
protected virtual void PreRestart(Exception reason, object message)
{
    ActorBase
    .Context
    .GetChildren()
    .ToList<IActorRef>()
    .ForEach((Action<IActorRef>)(c =>
        {
            ActorBase.Context.Unwatch(c);
            ActorBase.Context.Stop(c);
        }));
    this.PostStop();
}
```

Now we can create the code that sends the message and eventually stops the **`ActorSystem`**. This is something we have not yet mentioned: the **`ActorSystem.Terminate`** method will actually terminate the actor system, which will obviously terminate all of the actors instantiated. In our case, this will also cause our actor to terminate.

Code Listing 37: Sending messages to the EmailSenderActor

```
static void Main(string[] args)
{
    ActorSystem system = ActorSystem.Create("my-first-akka");

    IActorRef emailSender =
system.ActorOf<EmailSenderActor>("emailSender");

    EmailMessage emailMessage = new EmailMessage("from@mail.com",
"to@mail.com", "Hi");
    emailSender.Tell(emailMessage);

    Thread.Sleep(1000);

    system.Terminate();

    Console.Read();
}
```

The output of this is as follows:


R:\akka_net\akka\code\Akka.Net.Succinctly\Akka.Net.Succinctly.Chapter5\bin\Debug\Akka.Net.Succinctly.Chapter5.exe

```
Constructor() -> EmailSenderActor
PreStart() -> EmailSenderActor
email sent from from@mail.com to to@mail.com with content Hi
PostStop() -> EmailSenderActor
```

Figure 15: Showing the Actor's LifeCycle

Actor termination

In this section, we are going to explain how to terminate an actor: how to programmatically make sure that an actor will be terminated.

There are several ways to achieve this:

- Calling the **`ActorContext.Stop()`** method.
- Sending the **`Kill`** message.
- Sending the **`PoisonPill`** message.

- Terminating the **ActorSystem**, as we have seen in the previous example.
- Using the **GracefulStop**.

Stopping an actor

Stopping an actor can be done by using the **ActorContext.Stop** method. This is available in a few places, such as:

- The **ActorSystem** contains the **Stop** method, to which we can pass the actual actor reference.
- The actor itself has the ability to call the **Stop** method by calling **Context.Stop(Self)**.
- The actor can stop child actors by calling **Context.Stop(childActorReference)**.

One of the most important things to note is that the **Stop** method will only let the actor execute the message currently being executed before shutting down the whole actor instance. This means that no other messages will be executed, including those that are enqueued before the actual **Stop**() method is invoked.

Code Listing 38: Stopping an actor via ActorSystem

```
static void Main(string[] args)
{
    ActorSystem system = ActorSystem.Create("my-first-akka");

    IActorRef emailSender =
system.ActorOf<EmailSenderActor>("emailSender");

    EmailMessage emailMessage = new EmailMessage("from@mail.com",
"to@mail.com", "Hi");

    emailSender.Tell(emailMessage);

    system.Stop(emailSender);

    system.Terminate();
}
```

Sending the PoisonPill message

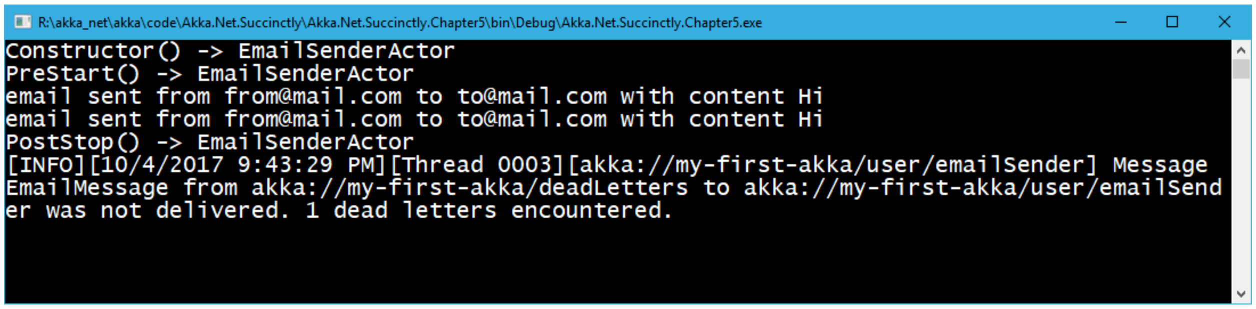
PoisonPill is a special kind of message (system message) that instructs the actor to shut itself down after receiving this kind of message. However, the actor will stop only after executing all of the messages that were part of the mailbox before receiving the **PoisonPill** message. Messages coming after, if any, will throw an error (because, in effect, actor doesn't exist anymore).

In Code Listing 38, we are sending three email messages to be executed; however, as we are going to see, only the first and second ones will be processed, while the third one will not, as it comes after the **PoisonPill** message. The **PoisonPill.Instance** represents an instance of the message.

Code Listing 39: Sending a PoisonPill

```
static void Main(string[] args)
{
    ActorSystem system = ActorSystem.Create("my-first-akka");

    IActorRef emailSender =
system.ActorOf<EmailSenderActor>("emailSender");

    EmailMessage emailMessage = new EmailMessage("from@mail.com",
"to@mail.com", "Hi");

    emailSender.Tell(emailMessage);
    emailSender.Tell(emailMessage);
    emailSender.Tell(PoisonPill.Instance);
    emailSender.Tell(emailMessage);

    Thread.Sleep(1000);

    system.Terminate();

    Console.Read();
}
```

With the following output, we can clearly see that the first two emails are sent, but the third one is not.


R:\akka_net\akka\code\Akka.Net.Succinctly\Akka.Net.Succinctly.Chapter5\bin\Debug\Akka.Net.Succinctly.Chapter5.exe

```
Constructor() -> EmailSenderActor
PreStart() -> EmailSenderActor
email sent from from@mail.com to to@mail.com with content Hi
email sent from from@mail.com to to@mail.com with content Hi
PostStop() -> EmailSenderActor
[INFO][10/4/2017 9:43:29 PM][Thread 0003][akka://my-first-akka/user/emailSender] Message
EmailMessage from akka://my-first-akka/deadLetters to akka://my-first-akka/user/emailSend
er was not delivered. 1 dead letters encountered.
```

Figure 16: PoisonPill effect

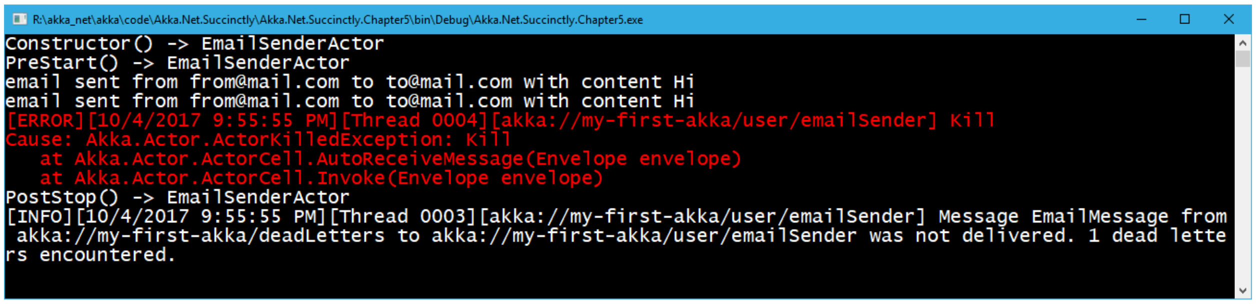
Sending the Kill message

Killing an actor is very similar to sending the **PoisonPill** message, which we have previously seen. The system offers the **Kill.Instance** message to be passed to the actor. The difference in this case is that when the actor encounters this kind of message, an exception will be thrown (**ActorKilledExeption**). This can be pretty useful if we want to show in logs that the actor was terminated. The usage of this seems to be rare, but it's good to know.

Code Listing 40: Sending the Kill.Instance message to an actor

```
static void Main(string[] args)
{
    ActorSystem system = ActorSystem.Create("my-first-akka");

    IActorRef emailSender =
system.ActorOf<EmailSenderActor>("emailSender");

    EmailMessage emailMessage = new EmailMessage("from@mail.com",
"to@mail.com", "Hi");

    emailSender.Tell(emailMessage);
    emailSender.Tell(emailMessage);
    emailSender.Tell(Kill.Instance);
    emailSender.Tell(emailMessage);

    Thread.Sleep(1000);

    system.Terminate();

    Console.Read();
}
```

Executing this code produces the following output—as before, the third message hasn't been delivered!


R:\akka_net\akka\code\Akka.Net.Succinctly\Akka.Net.Succinctly.Chapter5\bin\Debug\Akka.Net.Succinctly.Chapter5.exe

```
Constructor() -> EmailSenderActor
PreStart() -> EmailSenderActor
email sent from from@mail.com to to@mail.com with content Hi
email sent from from@mail.com to to@mail.com with content Hi
[ERROR][10/4/2017 9:55:55 PM][Thread 0004][akka://my-first-akka/user/emailSender] Kill
Cause: Akka.Actor.ActorKilledException: Kill
   at Akka.Actor.ActorCell.AutoReceiveMessage(Envelope envelope)
   at Akka.Actor.ActorCell.Invoke(Envelope envelope)
PostStop() -> EmailSenderActor
[INFO][10/4/2017 9:55:55 PM][Thread 0003][akka://my-first-akka/user/emailSender] Message EmailMessage from
 akka://my-first-akka/deadLetters to akka://my-first-akka/user/emailSender was not delivered. 1 dead lette
rs encountered.
```

Figure 17: Sending the Kill message to an actor result

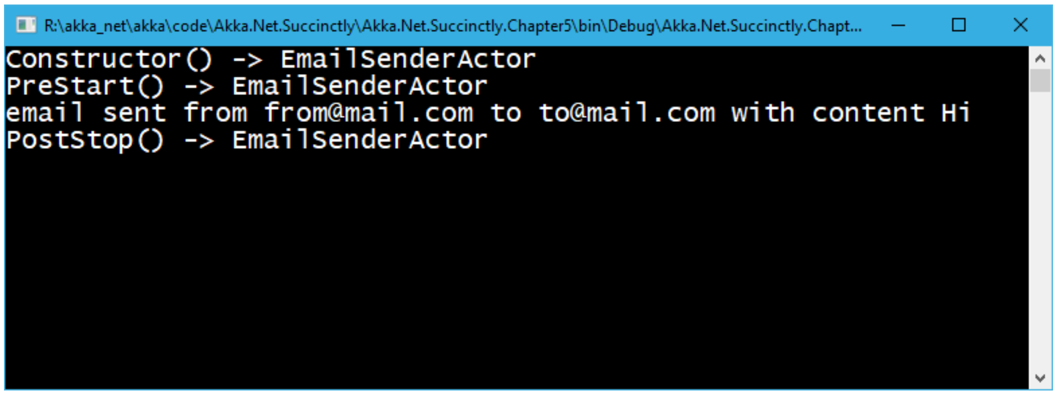
Gracefully stopping the actor

`IActorRef` also offers the **`GracefulStop`** method. By default, this method sends a **`PoisionPill`** and will return to your caller a **`Task<bool>`**, which will complete within the timeout you specify.

Code Listing 41: Gracefully stopping an actor

```
static void Main(string[] args)
{
    ActorSystem system = ActorSystem.Create("my-first-akka");
    IActorRef emailSender =
system.ActorOf<EmailSenderActor>("emailSender");

    EmailMessage emailMessage = new EmailMessage("from@mail.com",
"to@mail.com", "Hi");

    emailSender.Tell(emailMessage);
    var result = emailSender.GracefulStop(TimeSpan.FromSeconds(10));
    Thread.Sleep(1000);
    system.Terminate();
}
```

The result of this method produces the following output:


R:\akka_net\akka\code\Akka.Net.Succinctly\Akka.Net.Succinctly.Chapter5\bin\Debug\Akka.Net.Succinctly.Chapt...

```
Constructor() -> EmailSenderActor
PreStart() -> EmailSenderActor
email sent from from@mail.com to to@mail.com with content Hi
PostStop() -> EmailSenderActor
```

Figure 18: GracefulStop example output

Restarting an actor

Restarting an actor is just a bit more complicated, as it involves the understanding of the supervision strategies. We are going to discuss the actor hierarchy and supervision strategies in the next chapter, but for the time being, let's just say that every time we create a new actor, there is a default supervision strategy being associated to this actor. This means there is a standard behavior, and the system will know how to handle the failure (an **`exception`** being thrown by the actor).

The default **`supervision strategy`** will make it so that every time the actor has a processing error (an exception being thrown), it will ask the actor to **`restart`**, which means that a **new instance** of an actor will be created.

Note: By using** Props.Create<Actor>()**, we are able to supply a Supervision Strategy. If none has been supplied, the default one will be used.

The following example shows what happens when the actor throws an error. In order to demonstrate this, let's slightly change our **EmailSenderActor** to throw an **ArgumentException** in case the email content is not set (null or empty). So, let's go and change the **HandleEmailMessage** method as follows:

Code Listing 42: Email with no content, throws an error

```
private void HandleEmailMessage(EmailMessage message)
{
    if(string.IsNullOrEmpty(message.Content))
    {
        throw new ArgumentException("Cannot handle the empty content");
    }
    Console.WriteLine($"email sent from {message.From} to {message.To}");
}
```

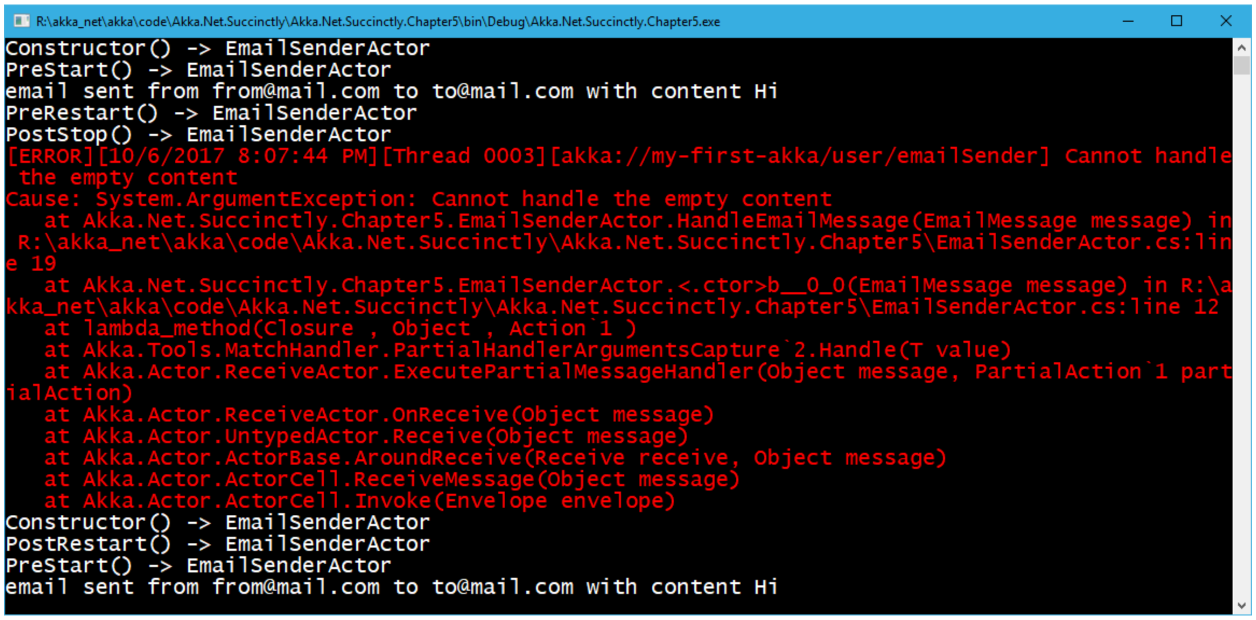
Here's an example that sends a message that will throw an error:

Code Listing 43: Calling the actor

```
static void Main(string[] args)
{
    ActorSystem system = ActorSystem.Create("my-first-akka");

    IActorRef emailSender =
system.ActorOf<EmailSenderActor>("emailSender");

    //send an invalid message (null content).
    EmailMessage invalidEMail = new EmailMessage("from@mail.com",
"to@mail.com", null);
    EmailMessage validEmail = new EmailMessage("from@mail.com",
"to@mail.com", "Hi");

    emailSender.Tell(validEmail);
    emailSender.Tell(invalidEMail);
    emailSender.Tell(validEmail);

    Console.Read();
    system.Terminate();
}
```

What we are doing here is simply sending a *valid* versus an *invalid* (empty content) email. As we have previously mentioned, we already know that the second message (**invalidEmail**) will raise an error and cause the actor to restart.

Let's just check this output, as shown in the following figure.



```
R:\akka_net\akka\code\Akka.Net.Succinctly\Akka.Net.Succinctly.Chapter5\bin\Debug\Akka.Net.Succinctly.Chapter5.exe
Constructor() -> EmailSenderActor
PreStart() -> EmailSenderActor
email sent from from@mail.com to to@mail.com with content Hi
PreRestart() -> EmailSenderActor
PostStop() -> EmailSenderActor
[ERROR][10/6/2017 8:07:44 PM][Thread 0003][akka://my-first-akka/user/emailSender] Cannot handle
 the empty content
Cause: System.ArgumentException: Cannot handle the empty content
   at Akka.Net.Succinctly.Chapter5.EmailSenderActor.HandleEmailMessage(EmailMessage message) in
 R:\akka_net\akka\code\Akka.Net.Succinctly\Akka.Net.Succinctly.Chapter5\EmailSenderActor.cs:lin
e 19
   at Akka.Net.Succinctly.Chapter5.EmailSenderActor.<.ctor>b__0_0(EmailMessage message) in R:\a
kka_net\akka\code\Akka.Net.Succinctly\Akka.Net.Succinctly.Chapter5\EmailSenderActor.cs:line 12
   at lambda_method(Closure , Object , Action`1 )
   at Akka.Tools.MatchHandler.PartialHandlerArgumentsCapture`2.Handle(T value)
   at Akka.Actor.ReceiveActor.ExecutePartialMessageHandler(Object message, PartialAction`1 part
ialAction)
   at Akka.Actor.ReceiveActor.OnReceive(Object message)
   at Akka.Actor.UntypedActor.Receive(Object message)
   at Akka.Actor.ActorBase.AroundReceive(Receive receive, Object message)
   at Akka.Actor.ActorCell.ReceiveMessage(Object message)
   at Akka.Actor.ActorCell.Invoke(Envelope envelope)
Constructor() -> EmailSenderActor
PostRestart() -> EmailSenderActor
PreStart() -> EmailSenderActor
email sent from from@mail.com to to@mail.com with content Hi
```

Figure 19: Restarting an actor

What we can see is that the first message gets sent just fine. The second one is obviously raising an exception (shown in red), but just before that, we can see that the **`PreRestart`** and **`PostStop`** events have been raised. Immediately afterwards, we can see that the actor restarts and the **`constructor`** gets called again, followed by the **`PostRestart`** method. What we have clarified here is that the **`PostRestart`** actually happens after the **`constructor`** is called.

Note: It’s important to keep in mind that when the actor restarts, it doesn’t maintain its state. That means a new instance of the actor gets created, and all of the private variables will simply be lost!

Chapter 6 The Actor's Switchable Behavior

Actors intrinsically have the ability to change their **`behavior`** at runtime! What do we mean by behavior? An actor's behavior is its ability to react differently to the received message, depending on the conditions.

When the actor starts, it has its own default behavior, which means that the function to be executed when the message is received is set up at that point. However, this function might change based on some conditions. For instance, if the client is not authorized to execute something, the actor can switch the behavior and deny any further execution.

Why is behavior interesting? One way of implementing behaviors would be to put conditional statements (**`if`**, **`then`**, **`else`**) all over the place in order to deal with a particular condition (is call authorized). Instead, the actor model solves this by allowing you to switch to a different message processor, so when the next message comes, this will be executed within the new behavior.

There is no limit on how many behaviors an actor may have, and changing of behavior is handled by the function **`Become()`**. After we call **`Become()`**, the previous behavior is forgotten unless we use the **`BecomeStacked()`** function, which creates a stack of behaviors, so that the behavior can be reverted by using **`UnbecomeStacked()`**.

Note: The change in the behavior always applies to the next message to be processed.

The following example will illustrate how this looks in practice.

Music player example

As an example of how the behavior can be used within an actor, we are going to build a little music player. This is a very simple example that hopefully will make it easy to understand the whole behavior concept.

The music player we are building accepts two message types, and has some rules:

- **`PlaySongMessage`**: Instructs the player to **start** playing a song. The rule is that the player cannot play another song if a song is already playing.
- **`StopPlayingMessage`**: Instructs the player to **stop** playing a song. If we stop an already stopped player, nothing happens.

This means that the music player has two behaviors (or states): one in which a song is currently being played, and one in which the player is stopped.

The following is the very simple definition of these two messages:

Code Listing 44: Music player input messages definition

```
public class PlaySongMessage
{
    public PlaySongMessage(string song)
    {
        Song = song;
    }

    public string Song { get; }
}

public class StopPlayingMessage
{
}
```

As we can see, the **`StopPlayingMessage`** class is pretty much a marker to instruct the player to stop playing. There is no additional information, as it's not needed. We know there can be only one song played at a time.

On the other hand, the **`PlaySongMessage`** has an attribute called **`Song`**, which represents the song name.

The actor we are going to create is called **`MusicPlayerActor`**, and will inherit from the **`ReceiveActor`** base class, which we have previously discussed.

Code Listing 45: Definition of the MusicPlayerActor

```
public class MusicPlayerActor : ReceiveActor
{
    protected string CurrentSong;

    public MusicPlayerActor()
    {
        StoppedBehavior();
    }

    private void StoppedBehavior()
    {
        Receive<PlaySongMessage>(m => PlaySong(m.Song));
        Receive<StopPlayingMessage>(m =>
                       Console.WriteLine("Cannot stop, the actor is already
stopped"));
    }

    private void PlayingBehavior()
    {
        Receive<PlaySongMessage>(m =>
                  Console.WriteLine($"Cannot play. Currently playing
'{CurrentSong}'"));
```

```
        Receive<StopPlayingMessage>(m => StopPlaying());
    }

    private void PlaySong(string song)
    {
        CurrentSong = song;
        Console.WriteLine($"Currently playing '{CurrentSong}'");

        Become(PlayingBehavior);
    }

    private void StopPlaying()
    {
        CurrentSong = null;
        Console.WriteLine($"Player is currently stopped.");

        Become(StoppedBehavior);
    }
}
```

After looking at the content of the actor, we can see there are two methods that define a behavior of an actor, as explained in Table 4.

Table 4: Music player behaviors

`PlayingBehavior()`	The behavior used while the song is playing. It handles both messages, and it's set once the song starts playing. It returns an error message (actually just a console message) if a new message to play a song (**`PlaySongMessage`**) is being sent. It responds to the **`StopPlayingMessage`**, as it will make sure that the music player stops.
`StoppedBehavior()`	The behavior used while the player is stopped. It handles both messages, and it's set at the creation of the actor (see the constructor). It returns an error message if we try to stop the already-stopped music player.

We are setting the behavior of the actor by simply creating the two methods that will decide what happens to the messages being handled.

In Code Listing 45, we can see how the **`Become`** method is being used to switch to the new behavior. We use **`Become`** just after we stop the player, or start playing a new song. Very straightforward!

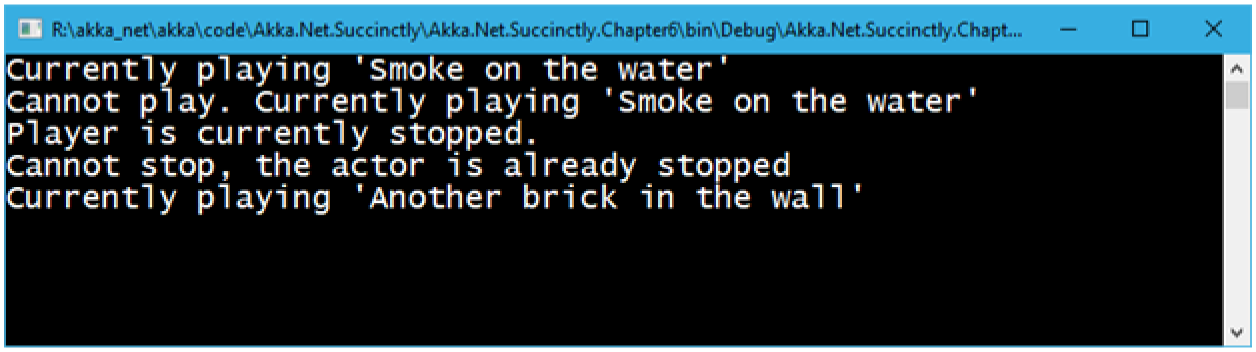
One thing to note is the fact that the music player actor has a state, defined by the **`CurrentSong`** property, which is set every time we play a new song. This shows how an actor can hold state. We mentioned that one of the attributes of an actor is that it has a state, and this is exactly it! It's not any different from a typical usage of a class, as we used when working with C#.

Let's see how to call the actor by starting and playing various songs.

Code Listing 46: Client code for running the music player

```
static void Main(string[] args)
{
    ActorSystem system = ActorSystem.Create("my-first-akka");

    IActorRef musicPlayer =
system.ActorOf<MusicPlayerActor>("musicPlayer");

    musicPlayer.Tell(new PlaySongMessage("Smoke on the water"));
    musicPlayer.Tell(new PlaySongMessage("Another brick in the wall"));
    musicPlayer.Tell(new StopPlayingMessage());
    musicPlayer.Tell(new StopPlayingMessage());
    musicPlayer.Tell(new PlaySongMessage("Another brick in the wall"));

    Console.Read();
    system.Terminate();
}
```

Figure 20 shows the output after running the **`Main`** method.


R:\akka_net\akka\code\Akka.Net.Succinctly\Akka.Net.Succinctly.Chapter6\bin\Debug\Akka.Net.Succinctly.Chapt...

```
Currently playing 'Smoke on the water'
Cannot play. Currently playing 'Smoke on the water'
Player is currently stopped.
Cannot stop, the actor is already stopped
Currently playing 'Another brick in the wall'
```

Figure 20: Result of playing the music

We can see that if we try to play a new song while the previous song is being played, the following message is displayed: **`Cannot play. Currently playing 'Smoke on the water'`**.

In the same way, if we try to stop an already-stopped music player, the following message will be shown: **`Cannot stop, the actor is already stopped`**.

Chapter 7 Actor Hierarchies

We've seen how to create the top-level actors by using the **`ActorSystem`** directly. In this chapter, we are going to see how to create the hierarchy of actors, which means: how an actor can create subactors and form a hierarchy.

Creating actor hierarchies

When creating new actors from existing actors, we don't use the **`ActorSystem`** as the entry point, but instead use the **`ActorContext`** object, which is part of the base actor's implementation.

`ActorContext` (or just **`Context`**, as this is how the actor base library exposes it) exposes contextual information for the actor and the current message. It provides more information, such as: factory methods to create child actors (in the same way the **`ActorSystem`** does), lifecycle monitoring, supervised children, and so on.

That said, let's see the actor's hierarchy in action by creating an example that expands the music player application so that it can support **multiple users** playing a song.

If you recall, in the previous example we only had one instance of the **`MusicPlayerActor`**. This actor was only capable of playing one song at a time. We will expand this example and create a sort of a multi-user music player that will be able to serve requests from multiple users.

What we will do is create another actor, **`MusicPlayerCoordinatorActor`**, which will act as a coordinator between user requests.

It's this actor's responsibility to create new instances (child actors) of **`MusicPlayerActor`** based on the user, so that each user will have its own copy of the **`MusicPlayerActor`**, so that it can play its own songs.

This controller actor, called **`MusicPlayerCoordinatorActor`**, is a regular actor, as we have seen before. However, it won't play any songs—this responsibility will still be within the **`MusicPlayerActor`**. The so-called coordinator actor is actually a pattern, and it enables actors to scale.

In Figure 21, we can see how the **`MusicPlayerCoordinatorActor`** receives the two messages (**`PlaySongMessage`** and **`StopPlayingMessage`**) and coordinates with the subactors.

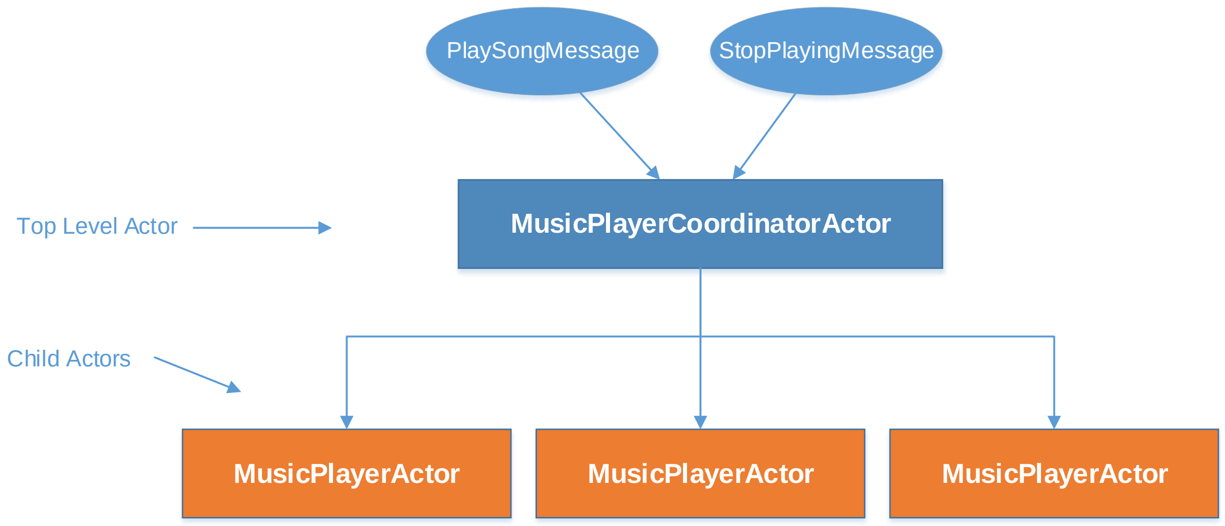


Figure 21: Actors' hierarchy

The input messages have to change slightly to include the **`user`** together with the song. The highlighted changes can be seen in the following code:

Code Listing 47: Input messages that include a user

```
public class PlaySongMessage
{
    public PlaySongMessage(string song, string user)
    {
        Song = song;
        User = user;
    }

    public string Song { get; }
    public string User { get; }
}

public class StopPlayingMessage
{
    public StopPlayingMessage(string user)
    {
        User = user;
    }

    public string User { get; }
}
```

Now we can create the new coordinator actor:

Code Listing 48: MusicPlayerCoordinatorActor definition

```
public class MusicPlayerCoordinatorActor : ReceiveActor
```

```
{
    protected Dictionary<string, IActorRef> MusicPlayerActors;

    public MusicPlayerCoordinatorActor()
    {
        MusicPlayerActors = new Dictionary<string, IActorRef>();

        Receive<PlaySongMessage>(message => PlaySong(message));
        Receive<StopPlayingMessage>(message => StopPlaying(message));
    }

    private void StopPlaying(StopPlayingMessage message)
    {
        var musicPlayerActor = GetMusicPlayerActor(message.User);
        if (musicPlayerActor != null)
        {
            musicPlayerActor.Tell(message);
        }
    }

    private void PlaySong(PlaySongMessage message)
    {
        var musicPlayerActor = EnsureMusicPlayerActorExists(message.User);
        musicPlayerActor.Tell(message);
    }

    private IActorRef EnsureMusicPlayerActorExists(string user)
    {
        IActorRef musicPlayerActorReference = GetMusicPlayerActor(user);

        MusicPlayerActors.TryGetValue(user, out musicPlayerActorReference);

        if (musicPlayerActorReference == null)
        {
            //create a new actor's instance.
            musicPlayerActorReference =
Context.ActorOf<MusicPlayerActor>(user);
            //add the newly created actor in the dictionary.
            MusicPlayerActors.Add(user, musicPlayerActorReference);
        }
        return musicPlayerActorReference;
    }

    private IActorRef GetMusicPlayerActor(string user)
    {
        IActorRef musicPlayerActorReference;
        MusicPlayerActors.TryGetValue(user, out musicPlayerActorReference);
        return musicPlayerActorReference;
    }
}
```

This actor contains a dictionary **`MusicPlayerActors`** object defined as **`Dictionary<string, IActorRef>`**, which contains the instances of the child actors. The string key is the player's name, as we would like to have only one **`MusicPlayerActor`** per user that plays the song.

Instances of an actor are created by the helper method **`EnsureMusicPlayerActorExists`**, whose responsibility it is to check first whether we already have an instance of an actor in a dictionary, and if not, to create an instance by using the **`Context.ActorOf<MusicPlayerActor>(user)`**. The **`Context`** we see here is actually an **`ActorContext`** object, which we have mentioned previously in this chapter.

We can see that every time a new message is received, this will be forwarded to the subactor that corresponds to the user playing the song. We also give a user's name as the name of the actor.

The **`MusicPlayerActor`** has been slightly changed so that it can display information about the current user. Now this actor contains the user name in the messages displayed.

Code Listing 49: MusicPlayerActor definition

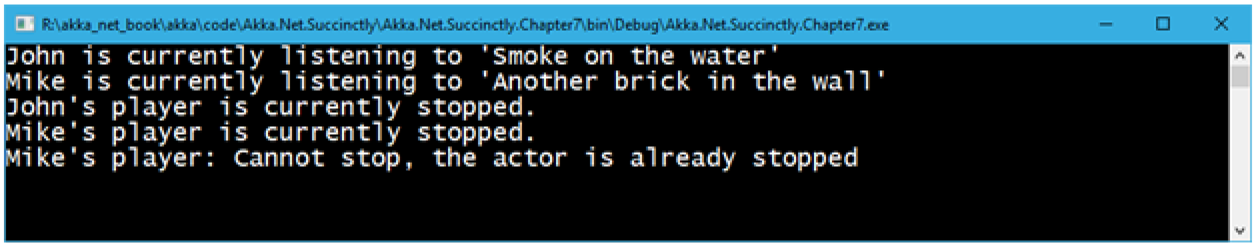
```
public class MusicPlayerActor : ReceiveActor
{
    protected PlaySongMessage CurrentSong;

    public MusicPlayerActor()
    {
        StoppedBehavior();
    }

    private void StoppedBehavior()
    {
        Receive<PlaySongMessage>(m => PlaySong(m));
        Receive<StopPlayingMessage>(m => Console.WriteLine($"{m.User}'s
player: Cannot stop, the actor is already stopped"));
    }

    private void PlayingBehavior()
    {
        Receive<PlaySongMessage>(m =>
Console.WriteLine($"{CurrentSong.User}'s player: Cannot play. Currently
playing '{CurrentSong.Song}'"));

        Receive<StopPlayingMessage>(m => StopPlaying());
    }

    private void PlaySong(PlaySongMessage message)
    {
        CurrentSong = message;

        Console.WriteLine(
                 $"{CurrentSong.User} is currently listening to 
'{CurrentSong.Song}'");

        Become(PlayingBehavior);
    }

    private void StopPlaying()
    {
        Console.WriteLine($"{CurrentSong.User}'s player is currently 
stopped.");
        CurrentSong = null;
        Become(StoppedBehavior);
    }
}
```

Now we can finally define the client code and send a few messages to the coordinator actor. We can see that now the **`User`** is specified in the message with the name of the song.

Code Listing 50: Client sending messages to MusicPlayerCoordinatorActor

```
static void Main(string[] args)
{
    ActorSystem system = ActorSystem.Create("my-first-akka");

    IActorRef dispatcher =
                     system.ActorOf<MusicPlayerCoordinatorActor>("player-
coordinator");

    dispatcher.Tell(new PlaySongMessage("Smoke on the water", "John"));
    dispatcher.Tell(new PlaySongMessage("Another brick in the wall", 
"Mike"));

    dispatcher.Tell(new StopPlayingMessage("John"));
    dispatcher.Tell(new StopPlayingMessage("Mike"));

    dispatcher.Tell(new StopPlayingMessage("Mike"));

    Console.Read();

    system.Terminate();
}
```

We can see that we have a very similar output to when we had a single **`MusicPlayer`**, except that now we can handle an unlimited number of users.


R:\akka_net_book\akka\code\Akka.Net.Succinctly\Akka.Net.Succinctly.Chapter7\bin\Debug\Akka.Net.Succinctly.Chapter7.exe

```
John is currently listening to 'Smoke on the water'
Mike is currently listening to 'Another brick in the wall'
John's player is currently stopped.
Mike's player is currently stopped.
Mike's player: Cannot stop, the actor is already stopped
```

Figure 22: Result of sending multiple messages to the controller

Chapter 8 Actor Path and Actor Selection

In this chapter we are going to discuss the actor path and actor selection. We'll see how the two are different, and their relationship with the actor reference, which we have mentioned already.

Actor path

Every time a new actor is created, a unique path is automatically associated to it. When we create an actor, we can set a name so that the actor is easily recognizable; otherwise, the framework will automatically add a random name. This is necessary for the actor to have its own unique path. This is very similar to the actual HTTP URL.

As an example of the music player seen in the previous chapter, let's simply add some information in the **`MusicPlayerActor`** to display some information about the internals. The parts in bold are new.

Code Listing 51: Change to the MusicPlayerActor to display path information

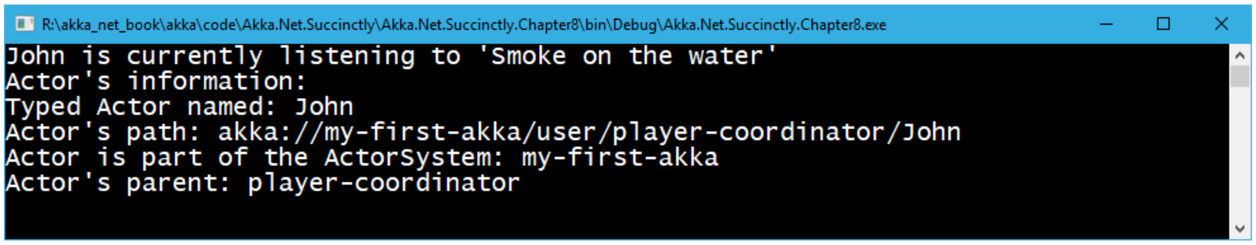
```
private void PlaySong(PlaySongMessage message)
{
    CurrentSong = message;
    Console.WriteLine($"{CurrentSong.User} is currently listening to
'{CurrentSong.Song}'");

    DisplayInformation();
    Become(PlayingBehavior);
}

private void DisplayInformation()
{
    Console.WriteLine("Actor's information:");
    Console.WriteLine($"Typed Actor named: {Self.Path.Name}");
    Console.WriteLine($"Actor's path: {Self.Path}");
    Console.WriteLine($"Actor is part of the ActorSystem:
{Context.System.Name}");
    Console.WriteLine($"Actor's parent: {Context.Self.Path.Parent.Name}");
}
```

The **`DisplayInformation`** method will now be displayed every time a new song is being played.

Code Listing 52: Sending message to the MusicPlayerCoordinatorActor

```
static void Main(string[] args)
{
    ActorSystem system = ActorSystem.Create("my-first-akka");
    var dispatcher = system.ActorOf<MusicPlayerCoordinatorActor>("player-
coordinator");

    dispatcher.Tell(new PlaySongMessage("Smoke on the water", "John"));

    Console.Read();

    system.Terminate();
}
```

The output of this looks as follows:


R:\akka_net_book\akka\code\Akka.Net.Succinctly\Akka.Net.Succinctly.Chapter8\bin\Debug\Akka.Net.Succinctly.Chapter8.exe

```
John is currently listening to 'Smoke on the water'
Actor's information:
Typed Actor named: John
Actor's path: akka://my-first-akka/user/player-coordinator/John
Actor is part of the ActorSystem: my-first-akka
Actor's parent: player-coordinator
```

Figure 23: Output displaying actor's information

We can clearly see that the **`Self.Path`** returns:

`akka://my-first-akka/user/player-coordinator/John`

We can distinguish the various parts in the actor's path itself: protocol, actor system, and the actual path of the actor.

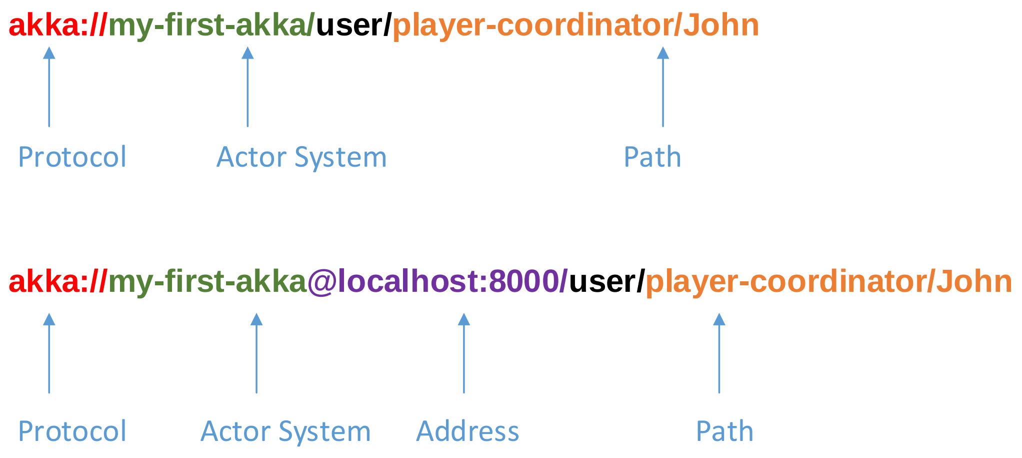

akka://my-first-akka/user/player-coordinator/John
Protocol
Actor System
Path
akka://my-first-akka@localhost:8000/user/player-coordinator/John
Protocol
Actor System
Address
Path

Figure 24: Actors Path

The various parts are explained in the following table.

Table 5: Actor's path parts

Protocol	**`akka://`**	Defines the communication protocol Akka.NET uses to communicate between actors. This is especially useful when it comes to the remote deployments of actors. For instance, we can define the use of the **tcp** protocol, such as **akka.tcp** or **akka.udp.**
Actor system	**`my-first-akka`**	The name of the **`ActorSystem`** to which the actor belongs.
Address	**`@localhost:8000`**	In case of remote deployment of actors (actors running on another system), there is a possibility to also add the address location of that system.
Path	**`player-coordinator/John`**	Refers to the path of this actor in the hierarchy.

What is the difference between actor reference and path?

An actor path represents a **name**, which may or may not be inhabited by an actor, and the path itself does not have a lifecycle—it never becomes invalid. That means a path can exist, but an actor instance might not be present (actor is shut down). It is possible to create an actor, terminate it, and then create a new actor with the same actor path. The newly created actor is a new instance of the actor, and therefore not the same (original) actor. This said, the actor reference to the old instance is not valid for the new instance. Note that messages sent to the old actor reference will not be delivered to the new actor's instance, even though they have the same path.

Actor selector

Let's imagine a situation where the two actors have to communicate to each other, but there is no direct connection between the two—an actor to which we are not holding an actor reference. In all of the examples we have seen so far, we always had an actor reference, which is a direct link to an actor.

In order to enable the communication between the two not directly connected actors, Akka.NET offers the **`ActorSelection`** mechanism. So instead of using the **`IActorRef`** instance to send a message to an actor, we can use the **`ActorSelection`** and directly reference the actor by using the actor's path.

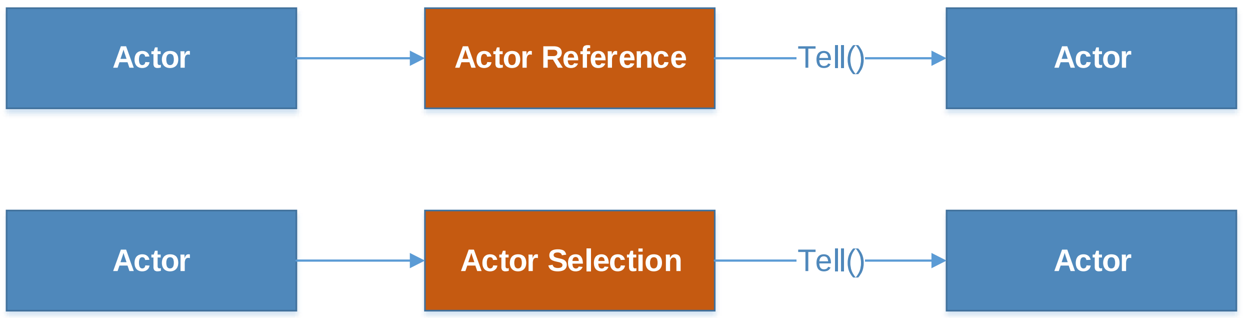


Figure 25: Actor's communication through Actor Reference and Actor Selection

`ActorSelector` is available as part of the **`ActorSystem`** or **`ActorContext`** classes. One possible usage follows.

To keep the previous music player example: let's imagine that we would like to **`Log`** all of the songs that have been played to get some sort of statistics or user preferences for the songs, so that the next time the user logs in, we can actually display similar songs or propose the old songs.

In order to do this, we are going to create a new actor, **`SongPerformanceActor`** (only one instance!), whose responsibility it is to keep track of songs being played, and every **`MusicPlayerActor`** will inform this new actor about the currently played song.

The new actor's communication is going to be as follows:

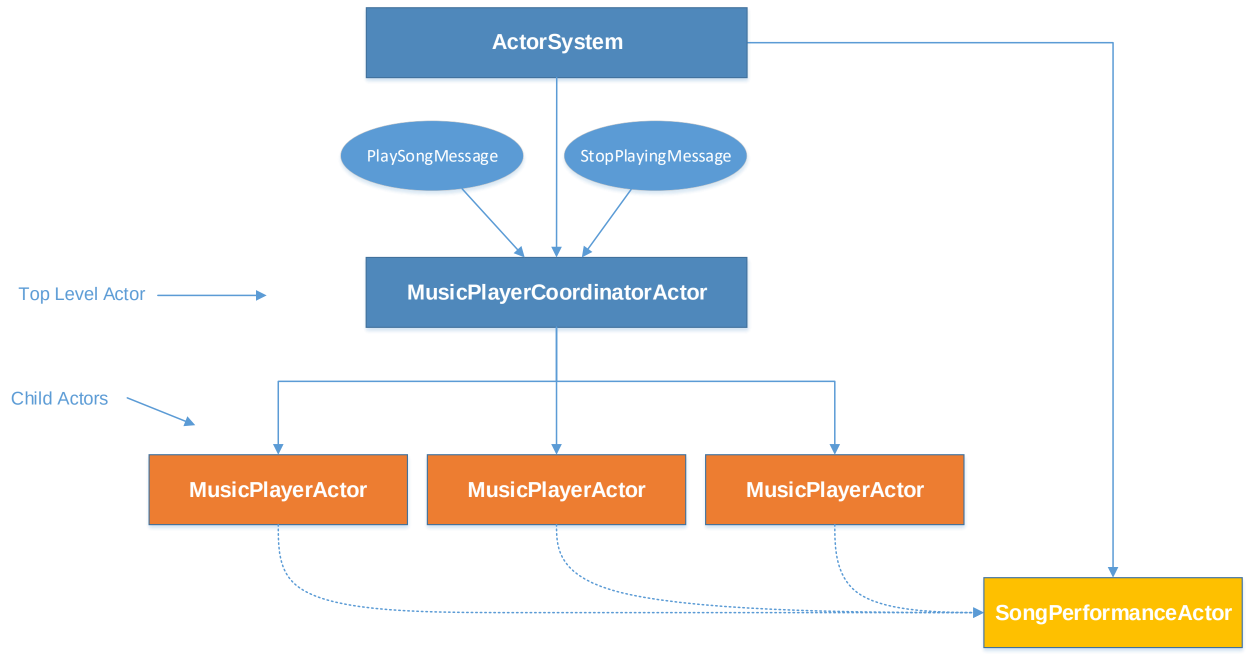


Figure 26: Tracking song statistics

Let's see how this works through an example.

The new **SongPerformanceActor** is quite straightforward: the plan is to have only **one** instance of it, and it will keep the state. The actor has the **SongPerformanceCounter** property, which holds the number of times one song (key) has been played.

Every time a new **PlaySongMessage** is sent to the actor, we will increase the counter by **1**.

The following is the code for the **SongPeformanceActor**:

Code Listing 53: SongPerformanceActor

```
public class SongPerformanceActor : ReceiveActor
{
    protected Dictionary<string, int> SongPeformanceCounter;

    public SongPerformanceActor()
    {
        SongPeformanceCounter = new Dictionary<string, int>();

        Receive<PlaySongMessage>(m => IncreaseSongCounter(m));
    }

    public void IncreaseSongCounter(PlaySongMessage m)
    {
        var counter = 1;
        if (SongPeformanceCounter.ContainsKey(m.Song))
        {
            counter = SongPeformanceCounter[m.Song]++;
        }
        else
        {
            SongPeformanceCounter.Add(m.Song, counter);
        }
        Console.WriteLine($"Song: {m.Song} has been played {counter}
times");
    }
}
```

The whole logic of the actor is placed into the **IncreaseSongCounter** method. First, we check whether the song is already in the dictionary, and if not, we simply add it with the count of **1**.

Alternatively, we retrieve the song item, and increase the value of it by **1**.

As we mentioned, the **MusicPlayerActor** is responsible for sending a message to the **SongPerformanceActor**. For brevity, leaving everything as is in the previous example, the only change to be made is to the **PlaySong** method, as highlighted in bold in the following code snippet:

Code Listing 54: Change made to the MusicPlayerActor in order to track statistics

```
private void PlaySong(PlaySongMessage message)
```

```
{
    CurrentSong = message;

    Console.WriteLine($"{CurrentSong.User} is currently listening to
'{CurrentSong.Song}'");

    var statsActor = Context.ActorSelection("../../statistics");
    statsActor.Tell(message);

    Become(PlayingBehavior);
}
```

We can see that in order to send a message, the **`MusicPlayerActor`** used the **`Context.ActorSelection`** method. The **`ActorSelection`** method accepts the **`Path`** of the actor, which we are supplying. It is possible to supply relative or absolute paths to the method:

- Relative path: **`../../statistics`**
- Absolute path: **`akka://my-first-akka/user/statistics`**

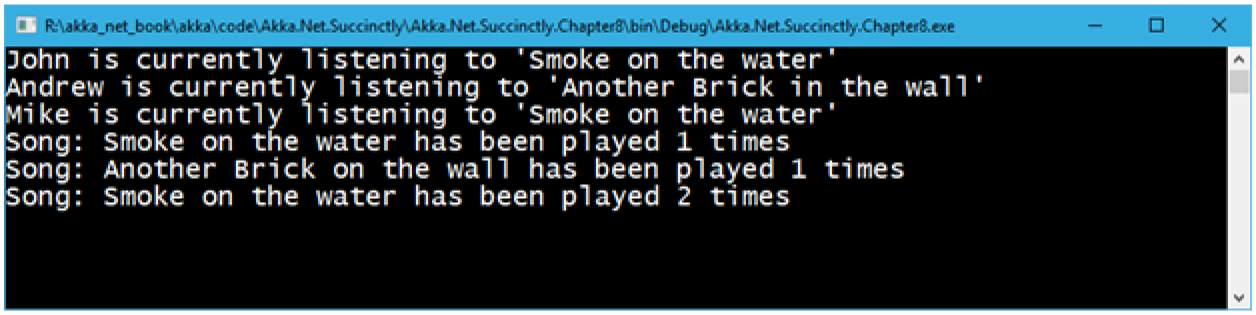
How do we know the path? In the **`Main`** method, we have given the actor name **`statistics`**. As we are directly creating the actor under the **`system.ActorOf`**, this makes it a top-level actor. We can clearly see this, as the actor is under the **`/user`** path directly.

The following code creates this actor and makes it available to be used by other actors.

*Code Listing 55: Main code that creates an instance of the **`SongPerformanceActor`***

```
static void Main(string[] args)
{
    ActorSystem system = ActorSystem.Create("my-first-akka");

    var dispatcher = system.ActorOf<MusicPlayerCoordinatorActor>("player-
coordinator");
    var stats = system.ActorOf<SongPerformanceActor>("statistics");

    dispatcher.Tell(new PlaySongMessage("Smoke on the water", "John"));
    dispatcher.Tell(new PlaySongMessage("Smoke on the water", "Mike"));
    dispatcher.Tell(new PlaySongMessage("Another Brick in the wall",
"Andrew"));
    Console.Read();

    system.Terminate();
}
```

This code produces the following output:


R:\akka_net_book\akka\code\Akka.Net.Succinctly\Akka.Net.Succinctly.Chapter8\bin\Debug\Akka.Net.Succinctly.Chapter8.exe

```
John is currently listening to 'Smoke on the water'
Andrew is currently listening to 'Another Brick in the wall'
Mike is currently listening to 'Smoke on the water'
Song: Smoke on the water has been played 1 times
Song: Another Brick on the wall has been played 1 times
Song: Smoke on the water has been played 2 times
```

Figure 27: ActorSelection example result

We can clearly see that the statistics are tracked correctly. Another thing to note is the order in which the messages are displayed. We can see that the **`SongPeformanceActor`** displays the message, but not fully in sequence. This is obviously due to the fact that this is fully asynchronous and under the hood: since we have three users (John, Mike, and Andrew), three **`MusicPlayerActor`** instances will be created. **`Tell`** is not a blocking call!

Chapter 9 Supervision

In the previous chapters, we have mentioned several times the fact that actors have dependencies and can define a supervision strategy. In this chapter, we are going to learn more about what the supervision is and how it works.

What is supervision?

Supervision describes the dependency relationship between actors. This intrinsically means that an actor can create other actors and *delegate* tasks to them. This ability also comes with the responsibility of handling those subactor's *failures*.

The actor that creates other actors is also known as a *supervisor,* and its children are known as *subordinates*. The failures of subordinates will be propagated to the supervisor actor.

In general, there are only a handful of ways to handle failures propagated back; let's take a look at them:

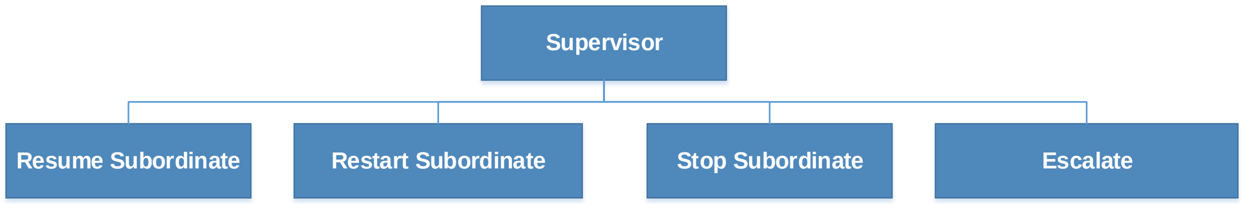


Figure 28: Possible supervision options

In Akka.NET terms, those actions to be taken are also known as **`Directives`**.

The following table briefly summarizes the various options.

Table 6: Possible supervision directives explained

`Resume`	Resuming a subordinate means that the subactor will be restarted, but it **still keeps its internal state**. This means the actor, after resuming, will keep doing its work from the point it stopped. Resuming an actor also means resuming **all** of its own subordinates. This is a chained operation!
`Restart`	This would completely clear out any subordinate's internal state. However, the messages that are in the queue remain intact! This also means all of the subordinates' created actors will be restarted.

`Stop`	Stops the subordinate permanently.
`Escalate`	This means the supervisor actor is not handling the error itself, but the handling of it is escalated to the supervisor's parent actor. The supervisor actor is failing itself!

After looking at these possible options, it becomes very clear that supervision is about forming a **recursive fault-handling structure**. Just remember what we have mentioned previously: the failure-handling is part of the actor model itself!

Actor supervision hierarchy

Recursive fault-handling structure brings us to the next topic: If our supervisor (actor) is not handling the errors, then who is the last supervisor to do so? Fortunately, Akka.NET ships an out-of-the-box the mechanism for this, by supplying the highest level actors that are at the top of the hierarchy. Those actors are always there, and provide the latest level of error propagation.

Let's take a look at the structure.

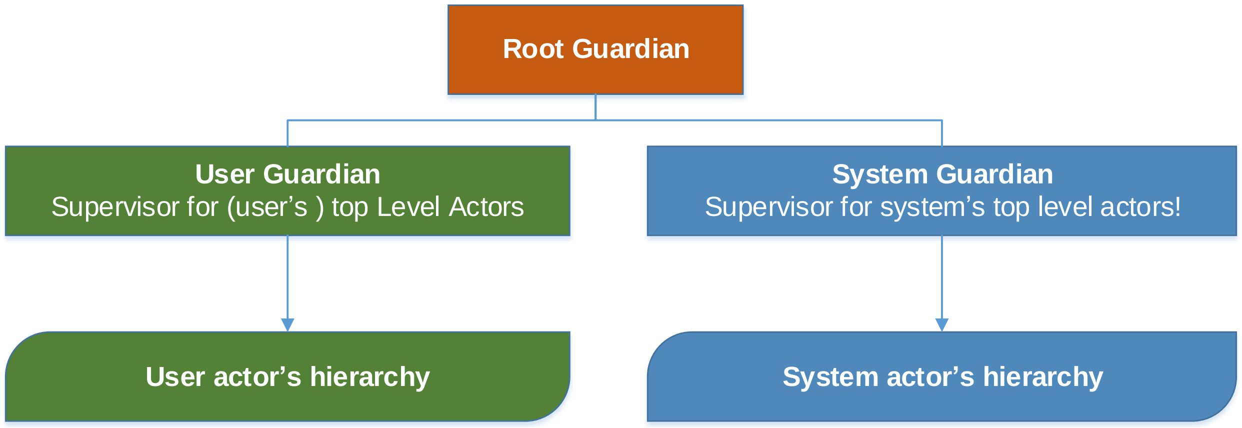


Figure 29: Actors' supervision guardians

User guardian

The user guardian defines the entry point for all of the actors defined by the application developer. Those are all the actors we are actually discussing in this book, as opposed to the system guardian, by which actors are actually managed and created by Akka.NET itself. Therefore, all of the actors created by `actorSystem.ActorOf()`, and their subsequent subactors, fall into this section.

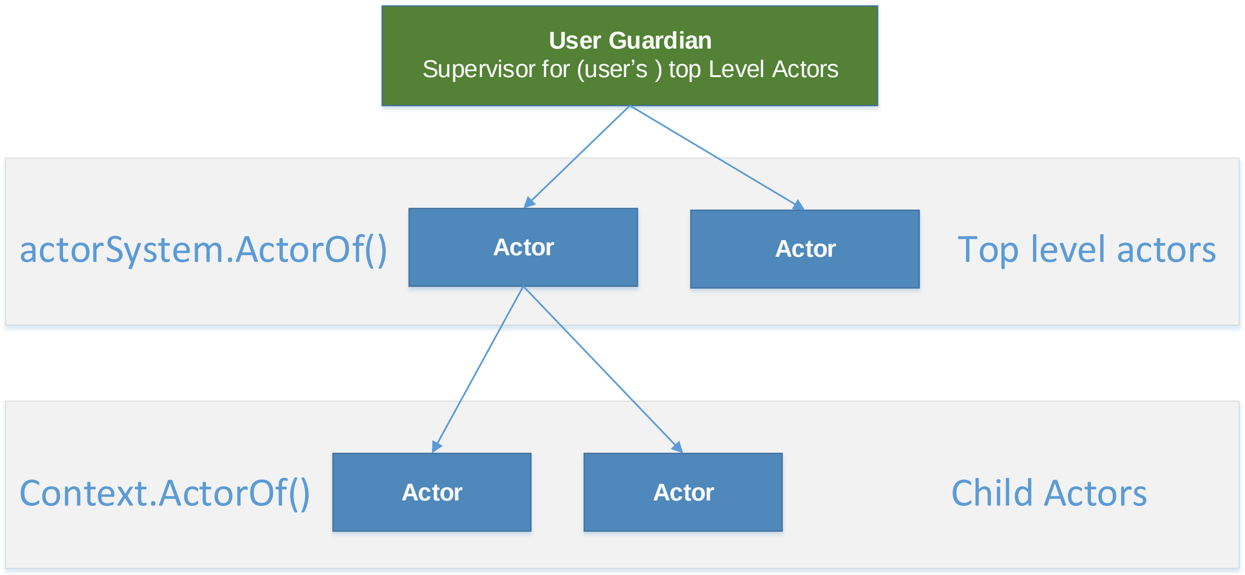


Figure 30: User guardian supervision actors

In the previous chapter we saw that when we create an actor, the path created is under the **`/user`** section. All of the user guardian actors will be placed there.

System guardian

System guardian was introduced in order to enable the shutting down of the system in an orderly manner. We don't have to worry about creating the system hierarchy, as the system will take care of it. System guardian actors will be recognized by the path starting with **`/system`**.

Root guardian

Root guardian is the ultimate place where the decision should be made. If the errors are propagated to the root guardian, it will make sure that the user guardian is shut down (or handled), therefore bringing down the whole system.

Supervision strategy

Now that we know which directives are available in order to handle the failure conditions, we have to understand how the parent handles this in terms of code. This is done through **`Supervision Strategies`**.

Whenever we create a new actor, the default supervision strategy is assigned to the actor, which is to **`Resume`** the actor. We obviously can override this behavior.

There are different strategies we can use, such as:

- **`OneForOneStrategy`**: This strategy will apply the directive **only to the failing actor** that has thrown an error. No other actors are affected.
- **`AllForOneStrateg`**y: This strategy applies the directive to **all of its children**. This is particularly useful if the children actors are dependent on each other, so that the failure of one brings the logical failure of others.

The example we are going to build is a continuation of the music player we saw in Chapter 7, where the **`MusicPlayerCoordinatorActor`** creates a new child **`MusicPlayerActor`** for every user that plays a song, so that every user has its own **`MusicPlayer`**.

Let's imagine a situation where the song played is not available, and therefore, the **`MusicPlayerActor`** will be in the faulted state. Just for this example, let's imagine that when the player tries to play the song "Bohemian Rhapsody," the **`SongNotAvailableException`** is thrown. The same happens when "Stairway to Heaven" plays, but this time, **`MusicSystemCorruptedException`** gets thrown. This is obviously hardcoded, and just to demonstrate that the actor is able to throw messages!

The following code contains the changes as described.

Code Listing 56: MusicPlayerActor throws exceptions

```
private void PlaySong(PlaySongMessage message)
{
    CurrentSong = message;

    if (message.Song == "Bohemian Rhapsody")
    {
        throw new SongNotAvailableException("Bohemian Rhapsody is not 
available");
    }

    if(message.Song == "Stairway to Heaven")
    {
        throw new MusicSystemCorruptedException("Song in a corrupt state");
    }

    Console.WriteLine(
              $"{CurrentSong.User} is currently listening to 
'{CurrentSong.Song}'");
}

public class SongNotAvailableException: Exception
{
    public SongNotAvailableException(string message): base(message)
    {

    }
}

public class MusicSystemCorruptedException : Exception
```

```
{
    public MusicSystemCorruptedException(string message): base(message)
    {

    }
}
```

In order to use one strategy or the other, we need to change the **`MusicPlayerCoordinatorActor`** and override the **`SupervisorStrategy`** method as follows:

Code Listing 57: Overriding SupervisionStrategy

```
public class MusicPlayerCoordinatorActor : ReceiveActor
{
…
    protected override SupervisorStrategy SupervisorStrategy()
    {
        return new OneForOneStrategy(e =>
        {
            if (e is SongNotAvailableException)
            {
                return Directive.Resume;
            }
            else if (e is MusicSystemCorruptedException)
            {
                return Directive.Restart;
            }
            else
            {
                return Directive.Stop;
            }
        });
    }
}
```

We can see that we need to return a **`SupervisionStrategy`** (in our case, the supervision strategy only acts on the faulty actor), and we also need to specify the **`Directive`**.

Code Listing 58: Client Code that calls the MusicPlayerCoordinatorActor

```
static void Main(string[] args)
{
    ActorSystem system = ActorSystem.Create("my-first-akka");

    var dispatcher =
system.ActorOf(Props.Create<MusicPlayerCoordinatorActor>());

    dispatcher.Tell(new PlaySongMessage("Bohemian Rhapsody", "John"));
    dispatcher.Tell(new PlaySongMessage("Stairway to Heaven", "Andrew"));

    Console.Read();

    system.Terminate();
}
```

Chapter 10 Other Components

EventBus

By definition, in Akka.NET the actors are communicating directly with each other by sending messages (by using **`Tell`**, **`Ask`**, **`Forward`**, etc.). This is what we can call one-to-one communication; one actor sends a message to another actor. However, sometimes there is a need to send information to more than one actor at a time (one-to-many), where an individual actor sends a message to a group of actors. In effect, this means that Akka.NET offers a publisher-subscriber mechanism out of the box.

The **`ActorSystem`** uses the **`EventStream`** for a number of internal things, including logging, sending dead letters, and cluster events.

An Akka.NET, both **system** and **user**-generated events (messages) can be published by using the **`EventStream`**. **`EventStream`** is the simplest and most common implementation of an **`EventBus`**.

`EventStream` supports several methods, such as:

- **`Publish`**: Publishes a message to the **`EventBus`**.
- **`Subscribe`**: An actor subscribes to the **`EventBus`**.
- **`Unsubscribe`**: An actor unsubscribes from receiving new messages.

The scenario is: One actor publishes a message (of some type) to the **`EventBus`**, while on the other side, one or more actors are subscribed to the events published to the **`EventBus`** of a particular type. Subscribers will automatically receive messages of that particular type as soon as the message is published.

Note: *EventStream* ***only works within one*** *ActorSystem* ***context. This means that the messages won't be available on another node (server).***

Let's demonstrate the usage of the **`EventBus`** with an example.

We are creating two actors: **`BookPublisher`**, responsible for publishing new books, and **`BookSubscriber`**, which will receive a message every time a new event is published on the bus. The implementation of the **`BookPublisher`** is interesting, as it uses the **`Context.System.EventStream.Publish()`** method to publish the message, meaning that the **`EventStream`** is available and accessible within an actor.

`BookSubscriber` is not any different from the actors we already checked. It just handles the receiving of the **`NewBookMessage`** type.

The **`NewBookMessage`** type is a simple class that only contains the name of the book we are publishing.

Code Listing 59: Publisher/Subscriber example

```
public class BookPublisher: ReceiveActor
{
    public BookPublisher()
    {
        Receive<NewBookMessage>(x => Handle(x));
    }

    private void Handle(NewBookMessage x)
    {
        Context.System.EventStream.Publish(x);
    }
}

public class BookSubscriber: ReceiveActor
{
    public BookSubscriber()
    {
        Receive<NewBookMessage>(x => HandleNewBookMessage(x));
    }

    private void HandleNewBookMessage(NewBookMessage book)
    {
        Console.WriteLine(
      $"Book: {book.BookName} got published - message received by
{Self.Path.Name}!");
    }
}

public class NewBookMessage
{
    public NewBookMessage(string name)
    {
        BookName = name;
    }

    public string BookName { get; }
}
```

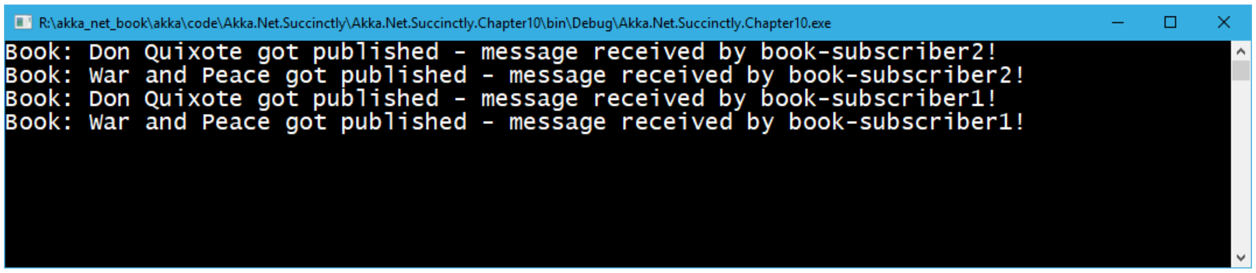
The client code contains some new methods we haven't seen yet. In the **`Main`** method, after creating an **`ActorRef`**, we are subscribing that actor to the event stream, and in specific cases, to receive the **`NewBookMessage`** types only.

We are using the **`Subscribe`** method directly in the **`Main`** function; however, this could be used directly when the actor gets instantiated, by using either the actor's constructor or **`PreStart`** method.

Code Listing 60: Main method enabling publishing and subscribing to messages

```
static void Main(string[] args)
{
    ActorSystem system = ActorSystem.Create("pub-sub-example");

    var publisher = system.ActorOf<BookPublisher>("book-publisher");

    var subscriber1 = system.ActorOf<BookSubscriber>("book-subscriber1");
    var subscriber2 = system.ActorOf<BookSubscriber>("book-subscriber2");

    system.EventStream.Subscribe(subscriber1, typeof(NewBookMessage));
    system.EventStream.Subscribe(subscriber2, typeof(NewBookMessage));

    publisher.Tell(new NewBookMessage("Don Quixote"));
    publisher.Tell(new NewBookMessage("War and Peace"));

    Console.Read();

    system.Terminate();
}
```

In the output, as we might expect, both subscriber actors will get the two published messages.


R:\akka_net_book\akka\code\Akka.Net.Succinctly\Akka.Net.Succinctly.Chapter10\bin\Debug\Akka.Net.Succinctly.Chapter10.exe

```
Book: Don Quixote got published - message received by book-subscriber2!
Book: War and Peace got published - message received by book-subscriber2!
Book: Don Quixote got published - message received by book-subscriber1!
Book: War and Peace got published - message received by book-subscriber1!
```

Figure 31: Publisher-Subscriber result

DeadLetters

There is a possibility that messages are not delivered to an actor. Those messages will be automatically delivered to a special actor called **`DeadLetters`**, which is available at the **`/deadLetters`** path. This rule usually applies to the nontransport lost messages, which means that Akka.NET makes no guarantees for lost messages at the transport layer.

Every time an actor terminates, there is a chance that some messages will be lost. If the actor is not available, and other actors are sending messages to it, those messages will end up in the **`DeadLetters`** mailbox.

How can we monitor DeadLetters?

An actor can subscribe to class **`Akka.Event.DeadLetter`** on the event stream. The subscribed actor will then receive all dead letters published in the (local) system from that point onwards. As dead letters are not propagated over the network, we would need to create an instance of the subscriber on each node.

Here is an example of how dead letters can be monitored.

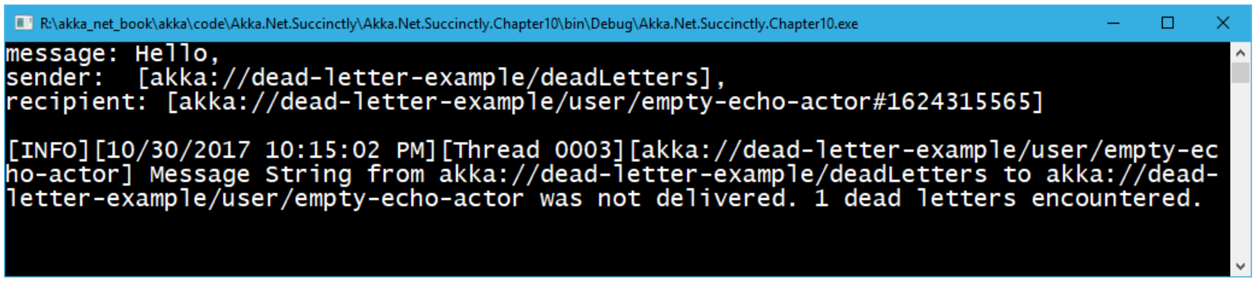
Code Listing 61: Dead letters monitoring

```
public class DeadLetterMonitor : ReceiveActor
{
    public DeadLetterMonitor()
    {
        Receive<DeadLetter>(x => Handle(x));
    }

    private void Handle(DeadLetter deadLetter)
    {
        var msg = $"message: {deadLetter.Message}, \n" +
                   $"sender:  {deadLetter.Sender},  \n" +
                   $"recipient: {deadLetter.Recipient}\n";

        Console.WriteLine(msg);
    }
}

public class EchoActor : ReceiveActor { }

static void Main(string[] args)
{
    ActorSystem system = ActorSystem.Create("dead-letter-example");

    var deadLettersSubscriber = system.ActorOf<DeadLetterMonitor>("dl-
subscriber");
    var echoActor = system.ActorOf<EchoActor>("empty-echo-actor");

    system.EventStream.Subscribe(deadLettersSubscriber,
typeof(DeadLetter));

    echoActor.Tell(PoisonPill.Instance);
    echoActor.Tell("Hello");
    Console.Read();

    system.Terminate();
}
```

There are two actors—one is called **DeadLetterMonitor**, whose responsibility is to subscribe to and handle the **DeadLetter** messages. In order to get the **DeadLetter** messages, the actor has to subscribe to the **EventStream**, and specify the type of messages it would receive (in our case, the **DeadLetter** type). Don't forget that the **DeadLetter** type is part of the **Akka.Event** namespace, so it has to be declared with **using Akka.Event**.

In order to simulate the unsuccessful sending of a message, we are going to create an instance of an **EchoActor**, which we will stop just before sending the first message. In this case, the system is going to try to send a message to a stopped actor, and as we have seen, this is not possible, and therefore the message will end up being captured by the **DeadLetterMonitor**.

When running this code, we can see the following output:



R:\akka_net_book\akka\code\Akka.Net.Succinctly\Akka.Net.Succinctly.Chapter10\bin\Debug\Akka.Net.Succinctly.Chapter10.exe

```
message: Hello,
sender:  [akka://dead-letter-example/deadLetters],
recipient: [akka://dead-letter-example/user/empty-echo-actor#1624315565]

[INFO][10/30/2017 10:15:02 PM][Thread 0003][akka://dead-letter-example/user/empty-ec
ho-actor] Message String from akka://dead-letter-example/deadLetters to akka://dead-
letter-example/user/empty-echo-actor was not delivered. 1 dead letters encountered.
```

Figure 32: Dead letters monitoring result

We can clearly see that the message did get delivered to the **DeadLetterMonitor** actor. We can also see that **akka://dead-letter-example/deadLetters** is the actor sending the message.

Chapter 11 Unit Testing Akka.NET

It's hard to imagine modern application development without the possibility of performing unit tests, or testing in general. Automated tests are not only an important part, but also an essential part of any application development cycle.

As testing actors is a bit different from testing any other piece of object-oriented software that we're used to, Akka.NET comes with a dedicated module, **`Akka.TestKit`**, for supporting tests at different levels.

`Akka.TestKit` is the fundamental library to be used in order to perform tests. In addition to this, there are different libraries built on top of it in order to support several other unit-testing frameworks, such as Xunit, NUnit, and Visual Studio tests.

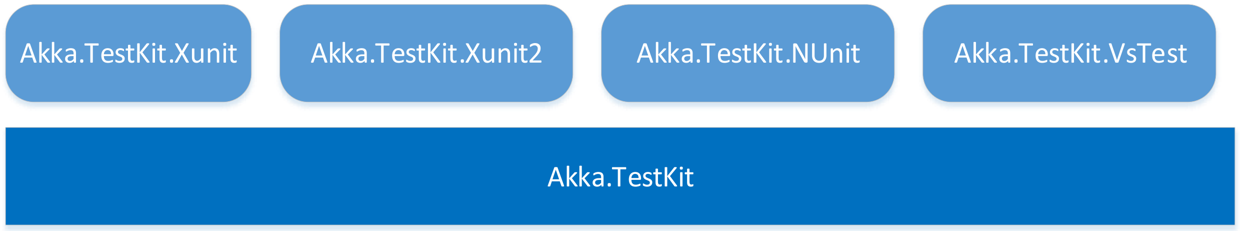


Figure 33: Akka.NET testing libraries

Including these libraries is as easy as installing them in the usual way from the NuGet package manager console.

Code Listing 62

```
PM> Install-Package Akka.TestKit
PM> Install-Package Akka.TestKit.NUnit
```

The **`TestKit`** class is the base for all of the unit tests, and all the unit test classes should inherit from it. **`TestKit`** class is actually implemented by the various libraries supporting the specific testing framework—as the real base class we have the **`Akka.TestKit.TestKitBase`** class, which is implemented in the **`Akka.TestKit`** library.

We would usually never inherit directly from the **`Akka.TestKit.TestKitBase`**, since there would be quite a few things to be implemented, and indeed, every library specific to the testing framework in use (for example, NUnit) would implement the plumbing needed to work with the actor system hiding this complexity from us.

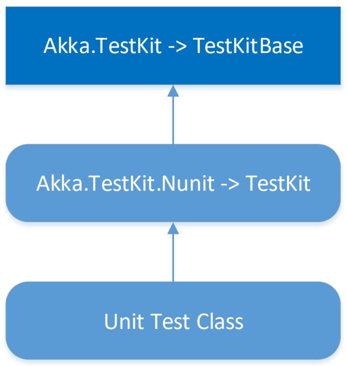


Figure 34: Hierarchy of the TestKit classes

The most basic unit test would look as follows:

Code Listing 62: Test skeleton

```
[TestFixture]
public class CalculatorActorTest: Akka.TestKit.NUnit.TestKit
{
    [Test]
    public void SomeTest()
    {
        /* test goes here */
    }
}
```

If you are not familiar with **`[TestFixture]`** and **`[Test]`**, these are the two attributes used to mark tests with **`NUnit`**, which we are going to use in the examples. You might use your favorite testing framework equally; the concepts are usually very similar.

We can divide the actor unit testing into four categories:

- **`Direct`**: Unit tests are written in the same way as normal classes. However, this method doesn't use any actor system, and therefore is limited in the sense that when performing direct tests, we are completely avoiding the message passing, which is the reason why we use the actor system. On the other hand, this kind of testing allows us to inspect the state of the actor, so it's very useful.
- **`Unit Testing`**: Unit tests are written by using the **`ActorSystem`**, and performed against one actor in particular.
- **`Integration Testing`**: We can use this technique when there is a need to test multiple message passing between actors.
- **`Mixed Mode`**: Unit tests that use the unit testing/integration testing with the help of the direct testing.

Direct testing

Performing direct testing is pretty straightforward, and not any different from what we usually do when testing any other kind of class. There is one thing in particular that we need to be careful about, which is to mark all of the message handlers as **`public`**, so that they're accessible to the unit test code. At first, this might sound like a security issue, but in reality, when actors are used within the actor system, these public methods are never accessed outside the actor itself.

Let's use our **`SongPerformanceActor`** as the actor to be tested. If you recall, this actor was used to increase the count of how many times a song has been played. So, this actor has a state, and this state can be tested. As a reminder, this actor looks like the following. Please note that we changed the **`SongPerformanceCounter`** property to **`public`**, together with the **`IncreaseSongCounter`** method.

Code Listing 63: SongPerformanceActor used for direct testing

```
public class SongPerformanceActor : ReceiveActor
{
    public Dictionary<string, int> SongPeformanceCounter;

    public SongPerformanceActor()
    {
        SongPeformanceCounter = new Dictionary<string, int>();

        Receive<PlaySongMessage>(m => IncreaseSongCounter(m));
    }

    public void IncreaseSongCounter(PlaySongMessage m)
    {
        var counter = 1;
        if (SongPeformanceCounter.ContainsKey(m.Song))
        {
            counter = SongPeformanceCounter[m.Song]++;
        }
        else
        {
            SongPeformanceCounter.Add(m.Song, counter);
        }
    }
}
```

We changed the private/protected methods to **`public`** so that they can be accessed by the unit test.

Our first test to check that the **`SongPeformanceCounter`** doesn't contain any entry upon an actor's creation would look as follows:

Code Listing 64: Actor starts with an empty state

```
[TestFixture]
```

```
public class SongPerformanceActorTest: Akka.TestKit.NUnit.TestKit
{
    [Test]
    public void ShouldStartWithAnEmptyState()
    {
        var actor = new SongPerformanceActor();

        Assert.False(actor.SongPeformanceCounter.Any());
    }
}
```

Nothing really difficult here. We instantiate a new actor, and simply check that there are no entries in the **`SongPerformanceCounter`** property.

The next thing to test would be increasing the count after playing a song. To achieve this, we would simply use the **`IncreaseSongCounter`** method and assert the result once again.

Let's add a new test method and invoke the **`IncreaseSongCounter`** method. We are asserting that the counter was set to **`1`** after "playing" the song once. Needless to say, this test passes.

Code Listing 65: Check whether the statistics are increased by one

```
[Test]
public void ShouldIncreaseSongByOne()
{
    var actor = new SongPerformanceActor();

    var songMessage = new PlaySongMessage("Bohemian Rhapsody", "John");

    actor.IncreaseSongCounter(songMessage);

    Assert.True(actor.SongPeformanceCounter[songMessage.Song] == 1);
}
```

This is very direct, and not at all different from what we already know about unit testing.

UnitTesting

A unit test is about testing the actor using the messaging involved by using the actor system. Now things are getting more complicated, and we need to introduce few concepts first.

When executing tests, the base **`TestKit`** class will create a test actor system in the background, referred to as **`Sys`**. Being an actor system, it will allow us to instantiate new actors, as we have seen in previous chapters, by using **`ActorOf`**. There is an alternative to this method without **`Sys.ActorOf`**, which is to use **`ActorOf<T>`**. Let's see a quick example of the **`ActorOf`** usage.

Code Listing 66: Not compiling ActorOf test

```
[Test]
```

```
public void ShouldIncreaseSongByOne()
{
    //var actor = base.Sys.ActorOf(new
Actor.Props(typeof(SongPerformanceActor)));
    IActorRef actor = ActorOf<SongPerformanceActor>();

    var songMessage = new PlaySongMessage("Bohemian Rhapsody", "John");

    actor.Tell(songMessage);

    Assert.True(actor.SongPeformanceCounter[songMessage.Song] == 1);
}
```

However, we can quickly notice that the **SongPerformanceCounter** class attribute is not any more visible! Unfortunately, this test cannot be performed, so we need to use the **ActorOfAsTestActorRef<T>** method. This is a wrapper method, and it has the **UnderlyingActor** attribute, which represents the underlying actor's instance, so that we can access its internal properties.

Code Listing 67: Passing unit test that uses ActorOfAsTestActorRef

```
[Test]
public void ShouldIncreaseSongByOne()
{
    TestActorRef<SongPerformanceActor> actor =

ActorOfAsTestActorRef<SongPerformanceActor>();

    var songMessage = new PlaySongMessage("Bohemian Rhapsody", "John");

    actor.Tell(songMessage);


Assert.True(actor.UnderlyingActor.SongPeformanceCounter[songMessage.Song]
== 1);
}
```

Testing an actor's message replies

In this section, we will find out how to test a situation where the actor will reply with a message: this is to demonstrate how to test the messages sent back by using the **Sender.Tell(message)** method. In order to do this, we are going to slightly adapt our actor's behavior so that, after increasing the counter, it will send back the message saying that the counter was increased.

The only change is to the **IncreaseSongCounter** method, highlighted in bold. In addition, we are also specifying the **CounterIncreasedMessage** message that will be resent.

Code Listing 68: Addition of the Sender.Tell method

```
public void IncreaseSongCounter(PlaySongMessage m)
{
    var counter = 1;
    if (SongPeformanceCounter.ContainsKey(m.Song))
    {
        counter = SongPeformanceCounter[m.Song]++;
    }
    else
    {
        SongPeformanceCounter.Add(m.Song, counter);
    }
    Sender.Tell(new CounterIncreasedMessage(m.Song, counter));
}

public class CounterIncreasedMessage
{
    public string Song;
    public long Count;

    public CounterIncreased(string song, long count)
    {
        this.Song = song;
        this.Count = count;
    }
}
```

Let's see how can we prove that this message is sent back to the sender.

Akka.TestKit offers a set of assert helper methods such as **`ExpectMsg`**, **`ExpectMsgFrom`**, and **`ExpectNoMsg`**, in order to test that the message is sent back from an actor. In our test example, we can see the **`ExpectMsg`** and the commented-out **`ExpectMsgFrom`** in action.

Code Listing 70: Testing return message with ExpectMsg()

```
[Test]
public void ShouldResendCounterIncreasedMessage()
{
    TestActorRef<SongPerformanceActor> actor =

ActorOfAsTestActorRef<SongPerformanceActor>();

    var songMessage = new PlaySongMessage("Bohemian Rhapsody", "John");

    actor.Tell(songMessage);

    /*
     * specify the actor explicitly
     * var replyMessage = ExpectMsgFrom<CounterIncreasedMessage>(actor);
     */

    var counter =
ExpectMsg<CounterIncreasedMessage>(TimeSpan.FromSeconds(5));

    Assert.That(counter.Song == "Bohemian Rhapsody");
    Assert.That(counter.Count == 1);
}
```

It's worth noticing that the **`ExpectMsg`** returns an instance of the **`CounterIncreasedMessage`** for further inspection and assertion. By default, the method will wait for three seconds, but this can be overridden as needed (**`TimeSpan.FromSeconds(5)`**).

`ExpectNoMsg` will test the fact that no messages are received, which would work fine in our previous example, before we decided to send back the message.

`ExpectMsgFrom` will test that the message comes from a specific actor.

Integration testing

In this section we are going to see how to check the parent-child relationship between two actors. If you recall, with **`MusicPlayerCoordinatorActor`** and **`MusicPlayerActor`**, the first will receive the message and create a new instance of the **`MusicPlayerActor`** for every user that plays music. The new child actor will be given the name of the user itself.

Here is the simplified version of the actors:

Code Listing 69: Simplified version of the MusicPlayerActor for testing

```
public class MusicPlayerActor : ReceiveActor
{
    protected PlaySongMessage CurrentSong;

    public MusicPlayerActor()
    {
        Receive<PlaySongMessage>(m => PlaySong(m));
    }

    private void PlaySong(PlaySongMessage message)
    {
        CurrentSong = message;
        Console.WriteLine(
                $"{CurrentSong.User} is currently listening to
'{CurrentSong.Song}'");
    }
}
```

The corresponding **`MusicPlayerCoordinatorActor`** follows:

Code Listing 70: Simplified version of the MusicPlayerCoordinatorActor for testing

```
public class MusicPlayerCoordinatorActor : ReceiveActor
{
    protected Dictionary<string, IActorRef> MusicPlayerActors;

    public MusicPlayerCoordinatorActor()
    {
        MusicPlayerActors = new Dictionary<string, IActorRef>();

        Receive<PlaySongMessage>(message => PlaySong(message));
    }

    private void PlaySong(PlaySongMessage message)
    {
        var musicPlayerActor = EnsureMusicPlayerActorExists(message.User);
        musicPlayerActor.Tell(message);
    }

    private IActorRef EnsureMusicPlayerActorExists(string user)
    {
        IActorRef musicPlayerActorRef;
        if (MusicPlayerActors.ContainsKey(user))
        {
            musicPlayerActorRef = MusicPlayerActors[user];
        }
        else
        {
            musicPlayerActorRef = Context.ActorOf<MusicPlayerActor>(user);
            MusicPlayerActors.Add(user, musicPlayerActorRef);
        }
        return musicPlayerActorRef;
    }
}
```

One aspect of the testing we might tackle is the creation of the child actor. How can we ensure that the child actor is created after we send a message to the **`MusicPlayerCoordinator`** actor? One way is to use the **`ActorSelection`** as follows:

Code Listing 71: Testing that the child actor is created

```
[Test]
public void ShouldInstantiateANewChildActor()
{
    TestActorRef<MusicPlayerCoordinatorActor> actor =
        ActorOfAsTestActorRef(() => new MusicPlayerCoordinatorActor(),
"Coordinator");

    var songMessage = new PlaySongMessage("Bohemian Rhapsody", "John");

    actor.Tell(songMessage);

    IActorRef child = this.Sys.ActorSelection("/user/Coordinator/John")
        .ResolveOne(TimeSpan.FromSeconds(5))
        .Result;

    Assert.That(child != null);
}
```

Since we know that every actor has a **`Path`** and can have a name specified, we are specifying the coordinator's actor name explicitly, as it is going to help us to define the final path. When creating the **`MusicPlayerCoordinatorActor`**, we are telling the test system to call it **`Coordinator`**.

We also already know that when the new child actor gets created, it will have the name of the user playing the song, so we know which **`Path`** the child actor will have upon instantiation, and we are going to use this information. In our case, this is **`John`**.

In order to figure out whether the child got created, we use the **`ActorSelection`** object, specifying the direct path to the child actor and checking for its existence. The resolution of the actor will take up to five seconds, and if the actor is not created in this timeframe, the test will throw an error and fail.

Chapter 12 Akka.NET Routers

So far, we have seen a single usage of actors: creating one actor at a time, and sending messages to and from actors. However, there are use cases where we would like to scale out the application and be able to have a responsive application, even under a heavy load.

Akka.NET provides special actors, called **`routers`**, that have the ability to group together actors and distribute a single message to several actors by applying different strategies. As a consequence of this, we can build systems that can perform as many operations simultaneously as possible.

In this chapter, we are going to explain how to work with the most common routers:

- Round Robin
- Random routing
- Shortest Mailbox queue
- Consistent Hashing

We can specify the router that an actor will use at the time of the creation of that actor. **`Props`** object contains a method **`WithRouter`** that can be used to specify it.

Round Robin

The Round Robin router is defined by the class **`RoundRobinPool`**, and is part of the namespace **`Akka.Routing`**. The Round Robin router can be configured to contain a number of instances of actors we would like to support in our application. Every time a new message arrives, the router will make sure that the next actor in the list will receive that message. This is particularly useful when we would like to balance the amount of work each actor needs to perform.

In Figure 35, we can see that the sender sends five messages, but we have configured the **`RoundRobinPool`** to contain only four actors. This would mean that Actor 1 will receive Message 1, Actor 2 will receive Message 2, Actor 3 will receive Message 3, and Actor 4 will receive Message 4. The next message (Message 5) will simply start from Actor 1, and so on, in a loop.

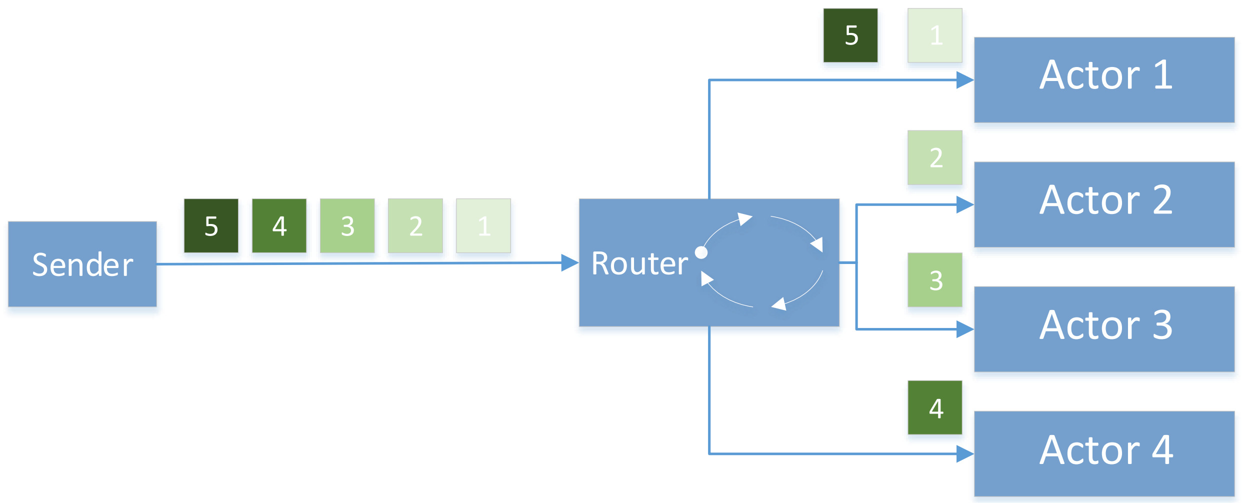


Figure 35: Round Robin router

To demonstrate this with the code, we will use the **`CalculatorActor`** we have seen previously, but modify it to include the debugging info so that we can follow which actor instance received which message.

Code Listing 72: CalculatorActor with logging

```
public class CalculatorActor : ReceiveActor
{
    public CalculatorActor()
    {
        Receive<Add>(add => HandleAddition(add));
    }

    public void HandleAddition(Add add)
    {
       Console.WriteLine($"{Self.Path} received the request:
{add.Term1}+{add.Term2}");
       Sender.Tell(new Answer(add.Term1 + add.Term2)) ;
    }
}
```

Code Listing 75 is the **`Main`** method that contains the code that creates the **`CalculatorActor .WithRouter(new Akka.Routing.RoundRobinPool(4))`** to define the **`RoundRobin`** router, and in addition, sets up four instances of **`CalculatorActor`** to be created and used.

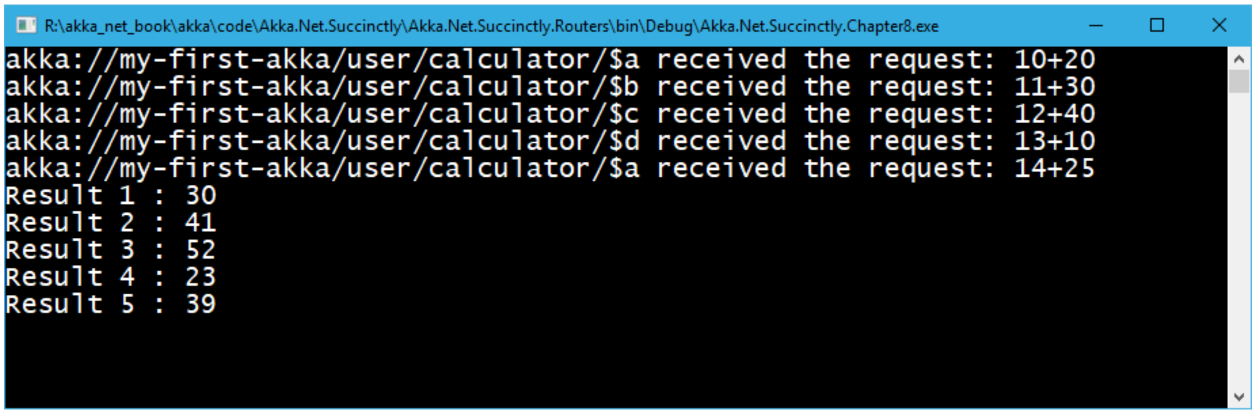
Code Listing 73: Issuing five calls to a round-robin router

```
static void Main(string[] args)
{
    ActorSystem system = ActorSystem.Create("my-first-akka");

    var calculatorProps = Props.Create<CalculatorActor>()
```

```
                            .WithRouter(new
Akka.Routing.RoundRobinPool(4));

    var calculatorRef = system.ActorOf(calculatorProps, "calculator");

    var result1 = calculatorRef.Ask(new Add(10, 20)).Result as Answer;
    var result2 = calculatorRef.Ask(new Add(11, 30)).Result as Answer;
    var result3 = calculatorRef.Ask(new Add(12, 40)).Result as Answer;
    var result4 = calculatorRef.Ask(new Add(13, 10)).Result as Answer;
    var result5 = calculatorRef.Ask(new Add(14, 25)).Result as Answer;

    Console.WriteLine($"Result 1 : {result1.Value}");
    Console.WriteLine($"Result 2 : {result2.Value}");
    Console.WriteLine($"Result 3 : {result3.Value}");
    Console.WriteLine($"Result 4 : {result4.Value}");
    Console.WriteLine($"Result 5 : {result5.Value}");

    Console.Read();

    system.Terminate();
}
```

Running this code returns the following output, where we can clearly see that the fifth message was delivered to the actor named **`$a`**, which was also the first actor in the sequence.

Notice that the names of actors are automatically generated, while the name **`calculator`**, which we set when setting up the router, becomes the parent. **`RoundRobinPool`** doesn't allow you to set up custom names for these actors.


R:\akka_net_book\akka\code\Akka.Net.Succinctly\Akka.Net.Succinctly.Routers\bin\Debug\Akka.Net.Succinctly.Chapter8.exe

```
akka://my-first-akka/user/calculator/$a received the request: 10+20
akka://my-first-akka/user/calculator/$b received the request: 11+30
akka://my-first-akka/user/calculator/$c received the request: 12+40
akka://my-first-akka/user/calculator/$d received the request: 13+10
akka://my-first-akka/user/calculator/$a received the request: 14+25
Result 1 : 30
Result 2 : 41
Result 3 : 52
Result 4 : 23
Result 5 : 39
```

Figure 36: RoundRobinPool–output

Random routing

Random routing probably defines the easiest way of routing, where the messages are sent randomly to the available actors, and therefore without any order. Random routing is defined by the **`RandomPool`** class, which, among other parameters, accepts the number of instances to be used.

As shown in Figure 37, the messages can be delivered to any actor at any given time. It's worth pointing out that one message goes to only one actor.

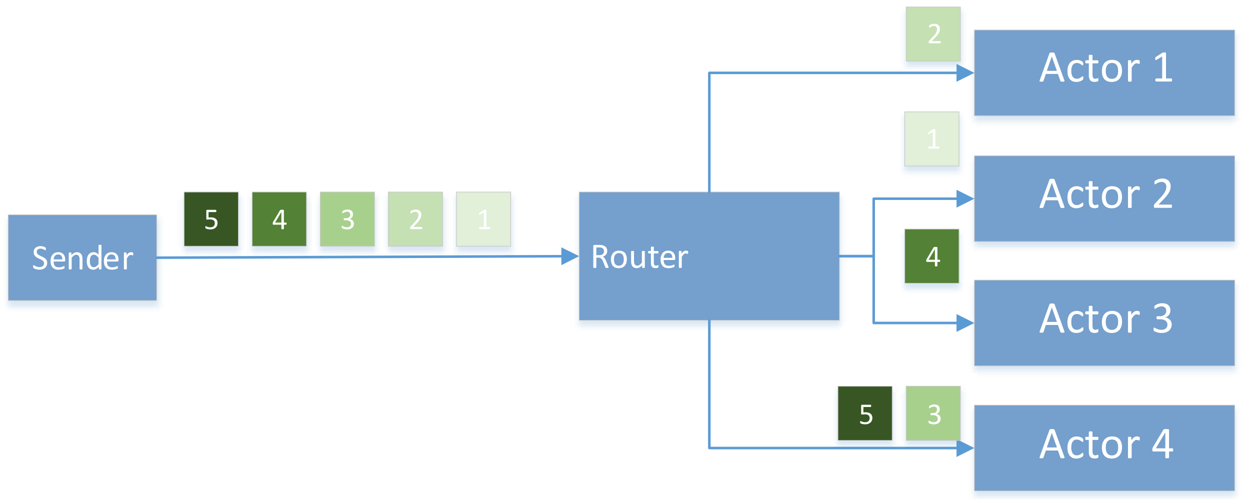


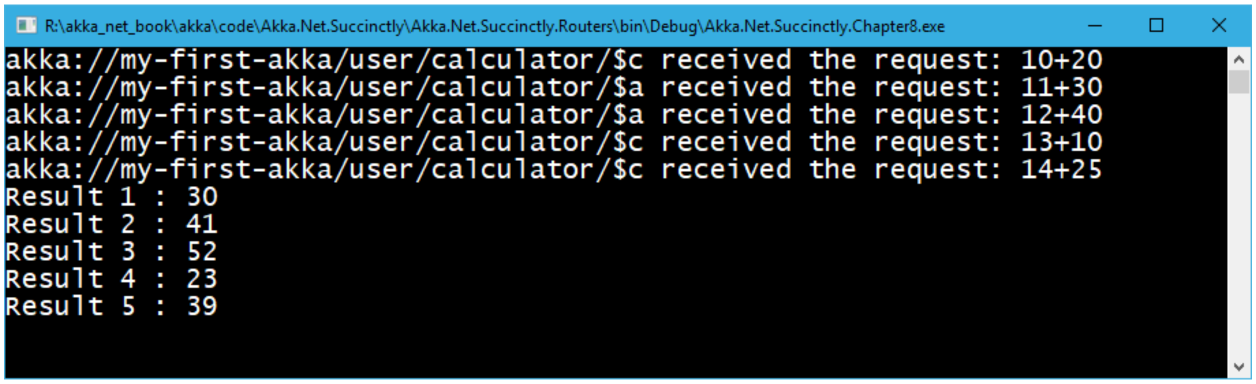
Figure 37: Random Routing

To use the Round Robin example, we will slightly change the previous example so that we use **`RandomPool`** rather than **`RoundRobinPool`**.

Code Listing 74: RandomPool example

```
static void Main(string[] args)
{
    …

    var calculatorProps = Props.Create<CalculatorActor>()
                            .WithRouter(new Akka.Routing.RandomPool(4));

    …
}
```

If we run the application, we can see that in the output, only the actors **`$a`** and **`$c`** have received the messages, in a random order.



```
R:\akka_net_book\akka\code\Akka.Net.Succinctly\Akka.Net.Succinctly.Routers\bin\Debug\Akka.Net.Succinctly.Chapter8.exe
akka://my-first-akka/user/calculator/$c received the request: 10+20
akka://my-first-akka/user/calculator/$a received the request: 11+30
akka://my-first-akka/user/calculator/$a received the request: 12+40
akka://my-first-akka/user/calculator/$c received the request: 13+10
akka://my-first-akka/user/calculator/$c received the request: 14+25
Result 1 : 30
Result 2 : 41
Result 3 : 52
Result 4 : 23
Result 5 : 39
```

Figure 38: Random router output

Shortest MailBox queue

We already know that every actor has its own private **`MailBox`**. Once the message is sent to it, the latest message gets appended to the end of the queue, so it will be picked up once all of the other messages are processed. The time it takes to process that last message depends on the duration of the processing of all other single messages that come before it.

To speed up this process, Akka.NET offers a mechanism so that the message will be sent to the actor whose **`MailBox`** contains fewer items in its queue. This way, the message can be processed as quickly as possible. In Figure 39, we can see that the router will choose Actor 2 to process the next message.

This logic is implemented by **`SmallestMailboxPool`**.

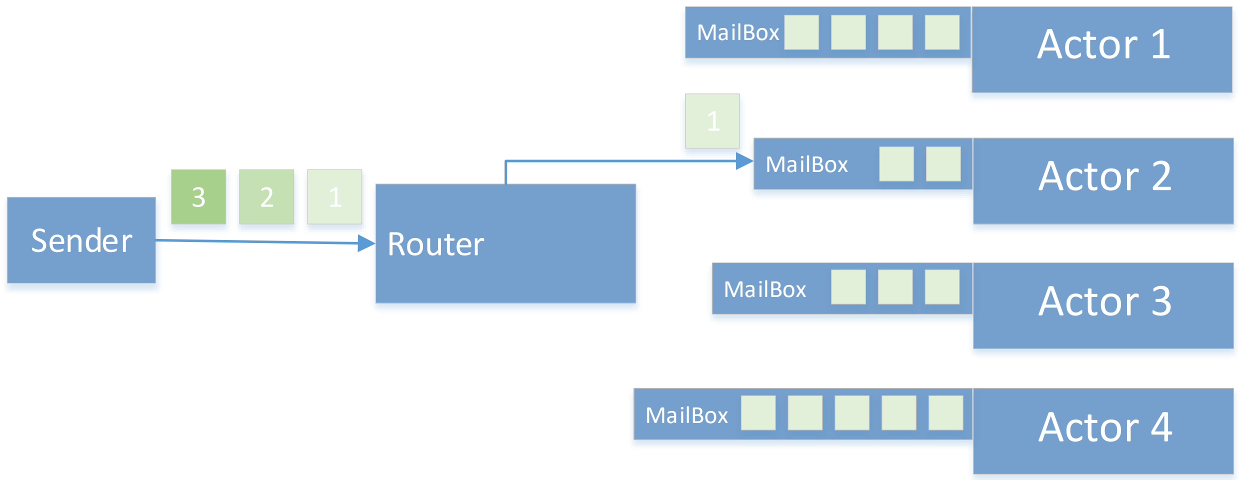


Figure 39: Shortest Mailbox router

We leave the example as it is, and only change the router, as follows:

Code Listing 77: Shortest Mailbox router

```
static void Main(string[] args)
{
    …

    var calculatorProps = Props.Create<CalculatorActor>()
                            .WithRouter(new
Akka.Routing.SmallestMailboxPool(4));

    …
}
```

Consistent Hashing

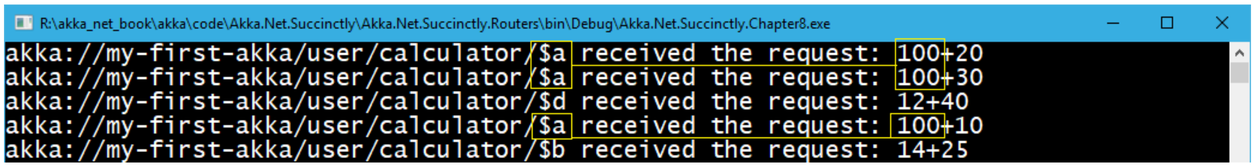
Consistent Hashing is a very interesting router, and a bit different from the previous ones we have seen. Instead of choosing a random router, we want to consistently send the messages with some characteristics, always to the same actor. The decision is based on the hashing of the message, or better, of the chosen property of the message. Consistent Hashing is implemented by the **`ConsistentHashingPool`** class, for which we can override the default hashing mechanism by using the **`WithHashMapping`** method, as follows:

Code Listing 75: Consistent Hashing example

```
using Akka.Routing;

static void Main(string[] args)
{
    ActorSystem system = ActorSystem.Create("my-first-akka");

    var calculatorProps = Props.Create<CalculatorActor>()
                            .WithRouter(new ConsistentHashingPool(4)
                            .WithHashMapping(x =>
                            {
                                if (x is Add)
                                {
                                    return ((Add)x).Term1;
                                }

                                return x;
                            }));

    var calculatorRef = system.ActorOf(calculatorProps, "calculator");

    calculatorRef.Tell(new Add(100, 20));
    calculatorRef.Tell(new Add(100, 30));
    calculatorRef.Tell(new Add(12, 40));
    calculatorRef.Tell(new Add(100, 10));
    calculatorRef.Tell(new Add(14, 25));
    Console.Read();

    system.Terminate();
}

public class CalculatorActor : ReceiveActor
{
    …
    public void HandleAddition(Add add)
    {
        Console.WriteLine($"{Self.Path} received the request:
{add.Term1}+{add.Term2}");

        if (!(Sender is DeadLetterActorRef))
            Sender.Tell(new Answer(add.Term1 + add.Term2));
    }
    …
}
```

In this example, we are choosing the **`Term1`** property to be hashed, so the actor to be used will depend on the hash value of the **`Term1`** property. Because we are using **`Tell`**, we had to slightly change the handling of the message in the **`CalculatorActor`** so that it doesn't return a message when the **`Sender`** is not an actor.

By executing this code, we can see that the Actor for **`Term1 = 100`** is always **`$a`**.



```
R:\akka_net_book\akka\code\Akka.Net.Succinctly\Akka.Net.Succinctly.Routers\bin\Debug\Akka.Net.Succinctly.Chapter8.exe
akka://my-first-akka/user/calculator/$a received the request: 100+20
akka://my-first-akka/user/calculator/$a received the request: 100+30
akka://my-first-akka/user/calculator/$d received the request: 12+40
akka://my-first-akka/user/calculator/$a received the request: 100+10
akka://my-first-akka/user/calculator/$b received the request: 14+25
```

Figure 40: Consistent Hashing output

The risk with this strategy is that one of the routers might take most of the load, depending on the messages—if most of the calculation performed is with number 100, then mostly only one actor will be used, which might not give optimal results.

Chapter 13 Actors in ASP.NET Core

The integration of Akka.NET with an ASP.NET Core application is relatively simple. As you might expect, ASP.NET Core is needed to expose an HTTP endpoint to the actor system, and we then can place all of the business logic inside actors, as we have seen in various examples.

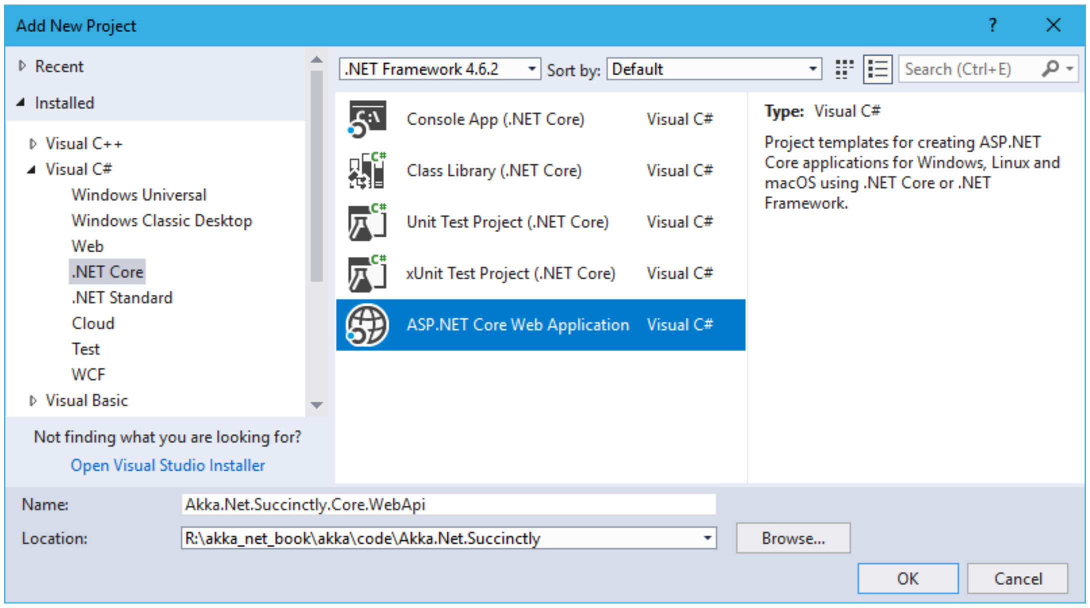
Since ASP.NET Core is a relatively new framework built to run on multiple platforms, it's a perfect fit for our example, as we are going to demonstrate that it's possible to use Akka.NET in a .NET Core application.

Our application won't be too difficult, but good enough to demonstrate the integration between the two frameworks. We are going to reuse one of the actors we already created, the **`CalculatorActor`**, and expose the addition as a RESTful service.

We won't be going into the details of ASP.NET Core, as the expectation is that you already know the basics.

Visual Studio project creation

Let's start by opening Visual Studio 2017 and adding a new project of type: **`.NET Core`** -> **`ASP.NET Core Web Application`**. You may call the project whatever you like, but in my case, this is **`Akka.Net.Succinctly.Core.WebApi`**.


Add New Project
?
Recent
Installed
Visual C++
Visual C#
Windows Universal
Windows Classic Desktop
Web
.NET Core
.NET Standard
Cloud
Test
WCF
Visual Basic
Not finding what you are looking for?
Open Visual Studio Installer
.NET Framework 4.6.2
Sort by:
Default
Search (Ctrl+E)
Console App (.NET Core) Visual C#
Class Library (.NET Core) Visual C#
Unit Test Project (.NET Core) Visual C#
xUnit Test Project (.NET Core) Visual C#
ASP.NET Core Web Application Visual C#
Browse...
OK
Cancel

Type: Visual C#

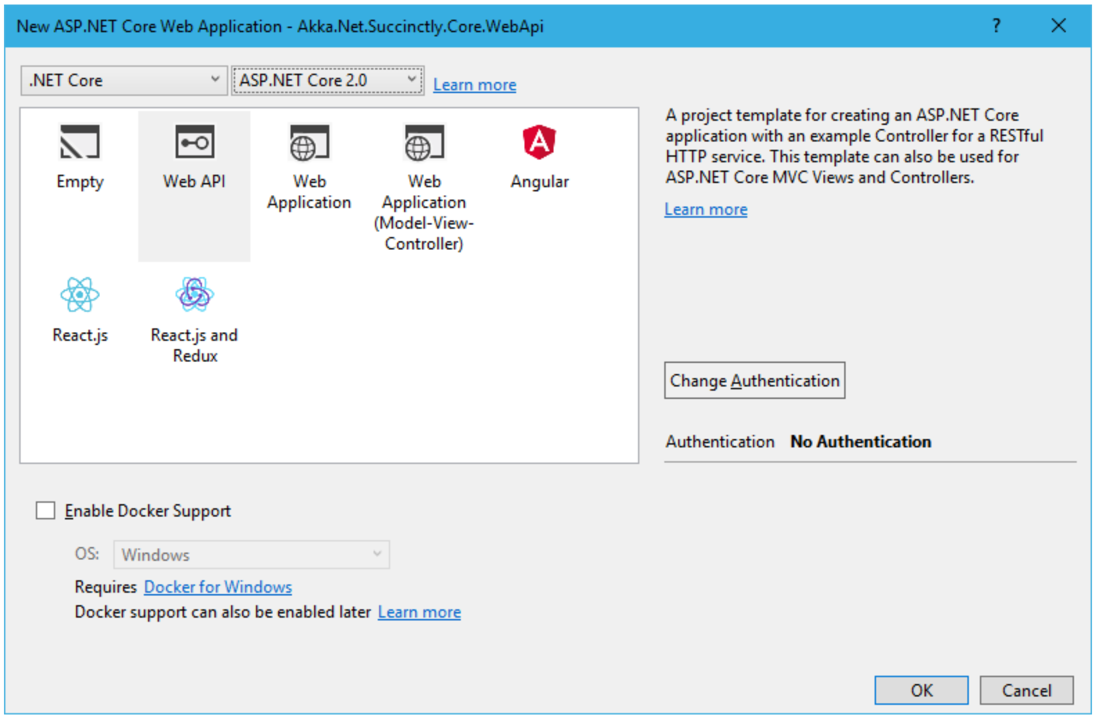
Project templates for creating ASP.NET Core applications for Windows, Linux and macOS using .NET Core or .NET Framework.

Name: Akka.Net.Succinctly.Core.WebApi

Location: R:\akka_net_book\akka\code\Akka.Net.Succinctly

Figure 41: Add a new .NET Core project

The next step is to choose which kind of **`.NET Core`** application we want to create. In our case, this is **`Web API`**. We will also choose **`.NET Core`** and **`ASP.NET Core 2.0`** as the running framework.



A project template for creating an ASP.NET Core application with an example Controller for a RESTful HTTP service. This template can also be used for ASP.NET Core MVC Views and Controllers.

Learn more

Authentication **No Authentication**

Requires Docker for Windows

Docker support can also be enabled later Learn more

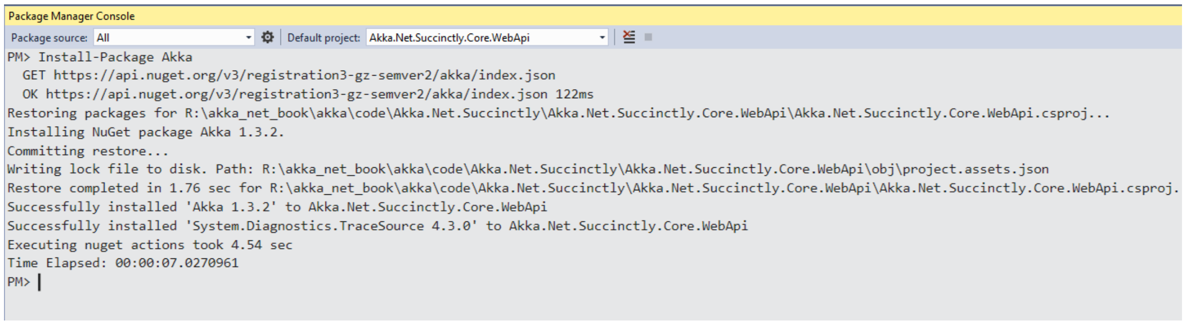
Figure 42: Choosing the right .NET Core project

First, we should reference Akka.NET from NuGet. So, let's run the following command in the Package Manager Console:

Code Listing 76: Installing Akka.NET in ASP.NET Core project

```
PM> Install-Package Akka
```

The Package Manager Console should look similar to the following:



```
PM> Install-Package Akka
  GET https://api.nuget.org/v3/registration3-gz-semver2/akka/index.json
  OK https://api.nuget.org/v3/registration3-gz-semver2/akka/index.json 122ms
Restoring packages for R:\akka_net_book\akka\code\Akka.Net.Succinctly\Akka.Net.Succinctly.Core.WebApi\Akka.Net.Succinctly.Core.WebApi.csproj...
Installing NuGet package Akka 1.3.2.
Committing restore...
Writing lock file to disk. Path: R:\akka_net_book\akka\code\Akka.Net.Succinctly\Akka.Net.Succinctly.Core.WebApi\obj\project.assets.json
Restore completed in 1.76 sec for R:\akka_net_book\akka\code\Akka.Net.Succinctly\Akka.Net.Succinctly.Core.WebApi\Akka.Net.Succinctly.Core.WebApi.csproj.
Successfully installed 'Akka 1.3.2' to Akka.Net.Succinctly.Core.WebApi
Successfully installed 'System.Diagnostics.TraceSource 4.3.0' to Akka.Net.Succinctly.Core.WebApi
Executing nuget actions took 4.54 sec
Time Elapsed: 00:00:07.0270961
PM> 
```

Figure 43: Package Manager: Installing Akka.NET

Now the project should be properly set up and ready to be changed!

Actor definition

Let's remind ourselves of what our actor looks like. The actor receives the **`AddMessage`**, and it sends back to the **`Sender`** the **`AnswerMessage`**, which is the result of the calculation.

Code Listing 80: CalculatorActor definition

```
public class CalculatorActor : ReceiveActor
{
    public CalculatorActor()
    {
        Receive<AddMessage>(add =>
        {
            Sender.Tell(new AnswerMessage(add.Term1 + add.Term2));
        });
    }
}

public class AddMessage
{
    public AddMessage(double term1, double term2)
    {
        Term1 = term1;
        Term2 = term2;
    }

    public double Term1;
    public double Term2;
}

public class AnswerMessage
{
    public AnswerMessage(double value)
    {
        Value = value;
    }

    public double Value;
}
```

ActorSystem integration

First, we need to find a place to instantiate the **`ActorSystem`**. We know that the creation of the **`ActorSystem`** is not cheap in terms of time and resource usage, so we want to do it only once, and reuse it throughout the whole application.

In order to do this, we can use the **`Startup.cs`**, which is an integral part of the ASP.NET Core template, and has already been created for us.

We will register the **ActorSystem** with the **Service** collection. The **Startup** class contains the **ConfigureService** method. After creating an instance of the **ActorSystem**, we will add it to the list of services, but as a singleton (**AddSingleton** method).

The statement **(serviceProvider) => actorSystem** means that every time the instance is requested, the **actorSystem** instance that we already created will be used.

Code Listing 77: Registering the ActorSystem in the service collection

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        var actorSystem = ActorSystem.Create("calculator-actor-system");
        services.AddSingleton(typeof(ActorSystem), (serviceProvider) =>
actorSystem);

        /* the rest of configuration */
    }
}
```

Controller definition

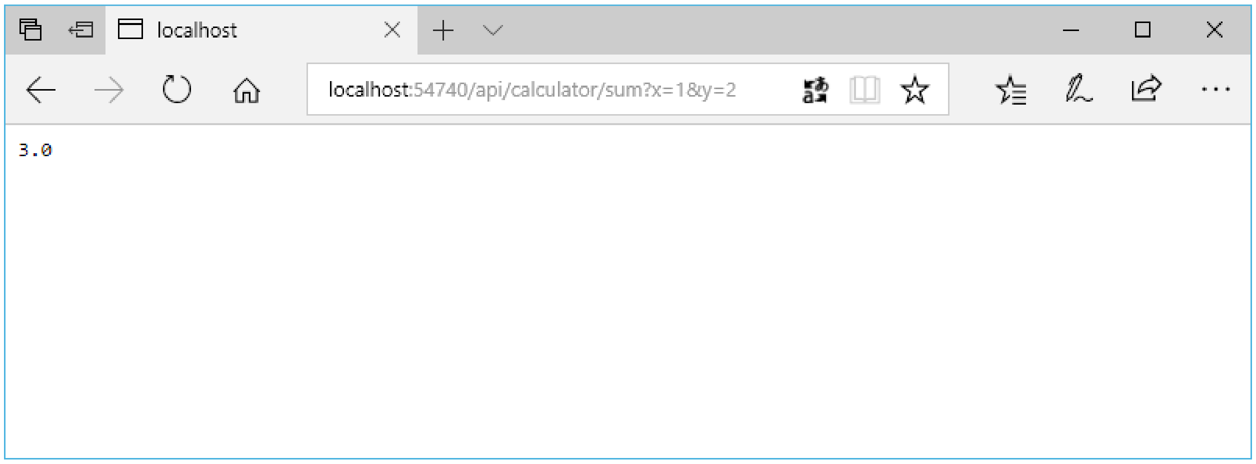
We are going to create a new controller called **CalculatorController**. **CalculatorController** only has one method, called **Sum**, which can be accessed by using the **GET HTTP** method. The **Sum** method has two parameters, **x** and **y**, which correspond to the two numbers we would like to sum up. The content of the **Sum** method is already familiar. We instantiate the **CalculatorActor**, construct the **AddMessage**, and **Ask** the actor to return the result of type **AnswerMessage**.

As we have registered the **ActorSystem** as part of services, we can now inject it to the controller via a constructor.

Code Listing 78: CalculatorController definition

```
[Route("api/[controller]")]
public class CalculatorController : Controller
{
    private ActorSystem _actorSystem;

    public CalculatorController(ActorSystem actorSystem)
    {
        _actorSystem = actorSystem;
    }

    [HttpGet("sum")]
    public async Task<double> Sum(double x, double y)
    {
        var calculatorActorProps = Props.Create<CalculatorActor>();
        var calculatorRef = _actorSystem.ActorOf(calculatorActorProps);

        AddMessage addMessage = new AddMessage(x, y);
        AnswerMessage answer = await
calculatorRef.Ask<AnswerMessage>(addMessage);

        return answer.Value;
    }
}
```

When running the application in the browser (by pressing **`F5`** in Visual Studio), we can now navigate to the **`<host>:<port>/api/calculator/sum?x=1&y=2`** page to see if the method returns the correct answer. As we can see in Figure 44, the correct answer is returned.



```
3.0
```

Figure 44: Invoking the Sum method via browser

One thing to pay attention to is the following: In the previous example, we are creating a new actor instance every time the **`Sum`** method is called, which is not that great, as it will consume memory, and those actors will be simply hanging around.

One workaround would be to try to get the instance of the actor via **`ActorSelection`**, but this also might lead to issues in a highly concurrent environment. A naïve implementation using **`ActorSelection`** would be:

Code Listing 79: Definition of the Sum method

```
[HttpGet("sum")]
public async Task<double> Sum(double x, double y)
{
    IActorRef calculatorRef;
    try
    {
        calculatorRef = await
_actorSystem.ActorSelection("/user/calculator")
.ResolveOne(TimeSpan.FromMilliseconds(100));

    }
    catch (ActorNotFoundException exc)
    {
        var calculatorActorProps = Props.Create<CalculatorActor>();
        calculatorRef =
_actorSystem.ActorOf(calculatorActorProps,"calculator");
    }

    AddMessage addMessage = new AddMessage(x, y);
    AnswerMessage answer = await
calculatorRef.Ask<AnswerMessage>(addMessage);

    return answer.Value;
}
```

In this example, we are trying to get an instance of the actor via **`ActorSelection`** by specifying the path, and if not found, we would create one and carry on. But, as I mentioned, there is a possibility that two concurrent ASP.NET Core threads will attempt to perform the actor selection at the same time, and to create an instance of the new calculator, which then would lead to errors. So, even though the version in Code Listing 83 would work, there is a potential risk that it would fail.

If we want to have only one actor serving all the requests, a better implementation would be the following:

Code Listing 80: CalculatorActor wrapper

```
public interface ICalculatorActorInstance
{
    Task<AnswerMessage> Sum(AddMessage message);
}

public class CalculatorActorInstance : ICalculatorActorInstance
{
    private IActorRef _actor;

    public CalculatorActorInstance(ActorSystem actorSystem)
    {
        _actor = actorSystem.ActorOf(Props.Create<CalculatorActor>(),
"calculator");
    }

    public async Task<AnswerMessage> Sum(AddMessage message)
    {
        return await _actor.Ask<AnswerMessage>(message);
    }
}
```

We will create a wrapper class around the actor itself. The logic stays the same as we had it previously in the controller. This new class will be registered in the ASP.NET Core services list as a singleton item, as follows:

Code Listing 81: Registering the ICalculatorActorInstance as singleton

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        var actorSystem = ActorSystem.Create("calculator-actor-system");
        services.AddSingleton(typeof(ActorSystem), (serviceProvider) =>
actorSystem);
        services.AddSingleton(typeof(ICalculatorActorInstance),

typeof(CalculatorActorInstance));

        /* the rest of configuration */
    }
}
```

Now we need to change the controller in order to inject the actor itself, rather than the **`ActorSystem`**.

The new version of the controller is much simpler, and it contains less code.

Code Listing 82: Injecting ICalculatorActorInstance

```
[Route("api/[controller]")]
public class CalculatorController : Controller
{
    private readonly ICalculatorActorInstance CalculatorActor;

    public CalculatorController(ICalculatorActorInstance calculatorActor)
    {
        CalculatorActor = calculatorActor;
    }

    [HttpGet("sum")]
    public async Task<double> Sum(double x, double y)
    {
        var answer = await CalculatorActor.Sum(new AddMessage(x, y));

        return answer.Value;
    }
}
```

Now we are pretty certain that there will be only one actor serving the requests. This will also allow us to better control the number of actor instances. We can choose to create not one instance of the **`CalculatorActor`** within the **`ICalculatorActorInstance`** implementation, but many, and have a pool of actors to serve the **`Sum`** method.

Chapter 14 Akka.NET Remoting

In the previous chapters, we saw the creation of actors belonging to only one **`ActorSystem`** and the communication between them. Being part of one **`ActorSystem`** automatically means that the application is hosted and running as part of one single process, which is a single application space.

While this is perfect for some kinds of applications, it might not be for others that need to scale out in order to increase the overall processing capabilities. This is where we need **`Akka.NET Remoting`**.

Remoting overview

Akka.NET remoting enables communication between **`ActorSystems`** deployed on different application processes. This might include a different process on a single machine, or a completely different server.

Let’s see an example. As shown in Figure 45, we have two actor systems deployed: one running on the client host, and one on the server host.

The communication between the two will happen via the network, and all of the messages sent across the wire will be serialized and deserialized. In order for the serialization to work, both actor systems should have the common definition of the messages (for instance, by sharing a library with message classes).

In addition to this, the communication between actors on two hosts will happen by using the full **`Path`**, which comprehends the protocol, address, port, and actor’s path. We have discussed the actor path in previous chapters.

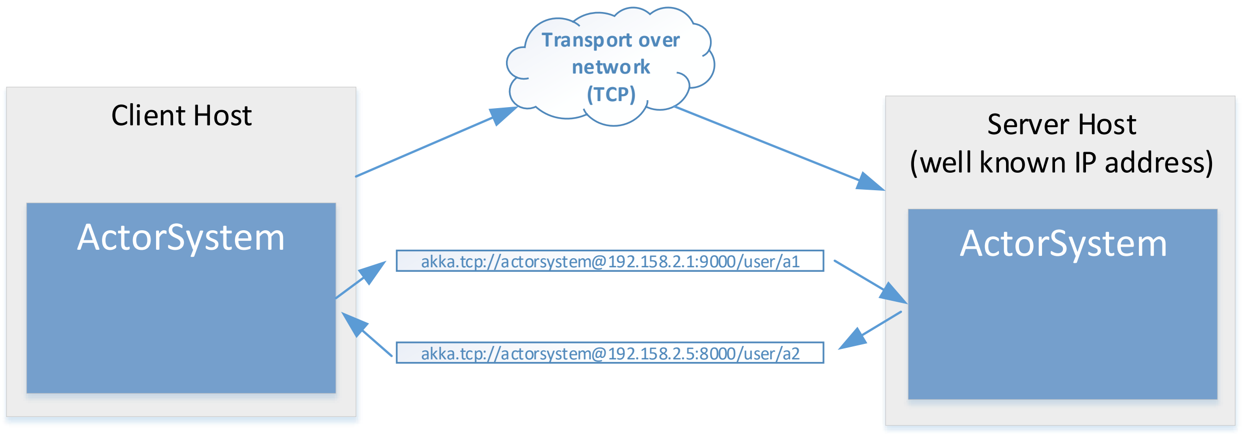

Transport over network (TCP)
Client Host
ActorSystem
Server Host (well known IP address)
ActorSystem
akka.tcp://actorsystem@192.158.2.1:9000/user/a1
akka.tcp://actorsystem@192.158.2.5:8000/user/a2

Figure 45: Akka.NET remoting enables communication

The client host (as shown in Figure 45) would typically be responsible for initiating the communication with the server host. The client host is also responsible for the creation of new actor instances.

With the use of remoting, the client host can choose *where* to instantiate new actors: either on the client (the local machine) or on the server host (remotely). To specify where the actor is going to be created, we use either the HOCON notation in the configuration file, or do that directly in the code.

The great thing about this is that the **`IActorRef`** returned upon creating an actor will not change its behavior, which means that our code will work in the same way as if the actor were running locally. This ability of the **`IActorRef`** to abstract the location of the actor is also known as the location transparency.

Transport

By default, Akka.NET uses the DotNetty library for the TCP transport. The responsibilities of the transport in Akka.NET are as follows:

- **IP addressing and ports**: All Akka.NET endpoints (client and servers) have to be accessible by an IP address and port.
- **Message delivery**: Should enable the delivery of messages from **`ActorSystem`** A to **`ActorSystem`** B.
- **Message framing**: Distinguishes individual messages within a network stream.
- **Disconnect and error reporting**: Tells **`Akka.Remote`** when a disconnect or transport error occurred.
- **Preserving message order (optimal)**: Some transports (like UDP) make no such guarantees, but in general, it's recommended that the underlying transport preserve the write order of messages on the read side.
- **DNS resolution (optional)**: Be able to resolve the DNS.

Remoting demo application

Let’s build an example to demonstrate the remote instantiation of the actors.

As the demo application, we will expand the previously built ASP.NET Core calculator web application by changing it a bit, as follows:

1. We will add a new Console Application project that will be used to host the remote **`ActorSystem`**, and execute the code as requested by the client.
2. We will create a new Common Code library, which is a library that will be shared by the two other applications.

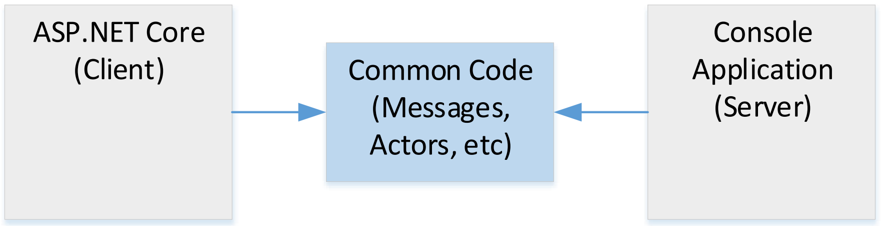
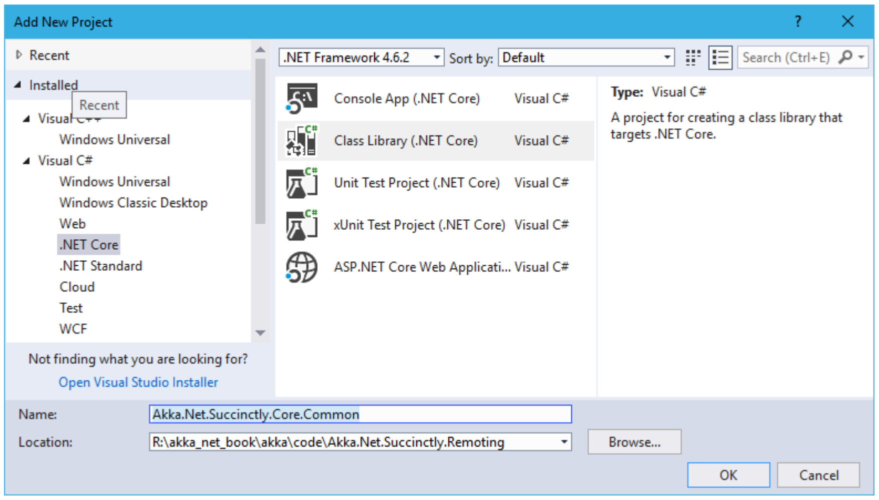

ASP.NET Core (Client)
Common Code (Messages, Actors, etc)
Console Application (Server)

Figure 46: Physical structure of the remoting demo

Let's start by creating the two needed projects, which are to be added to the already existing ASP.NET Core solution.


Add New Project
Recent
Installed
Visual C++
Windows Universal
Visual C#
Windows Universal
Windows Classic Desktop
Web
.NET Core
.NET Standard
Cloud
Test
WCF
Not finding what you are looking for?
Open Visual Studio Installer
.NET Framework 4.6.2
Sort by: Default
Search (Ctrl+E)
Console App (.NET Core) Visual C#
Class Library (.NET Core) Visual C#
Unit Test Project (.NET Core) Visual C#
xUnit Test Project (.NET Core) Visual C#
ASP.NET Core Web Applicati... Visual C#
Browse...
OK
Cancel

Type: Visual C#

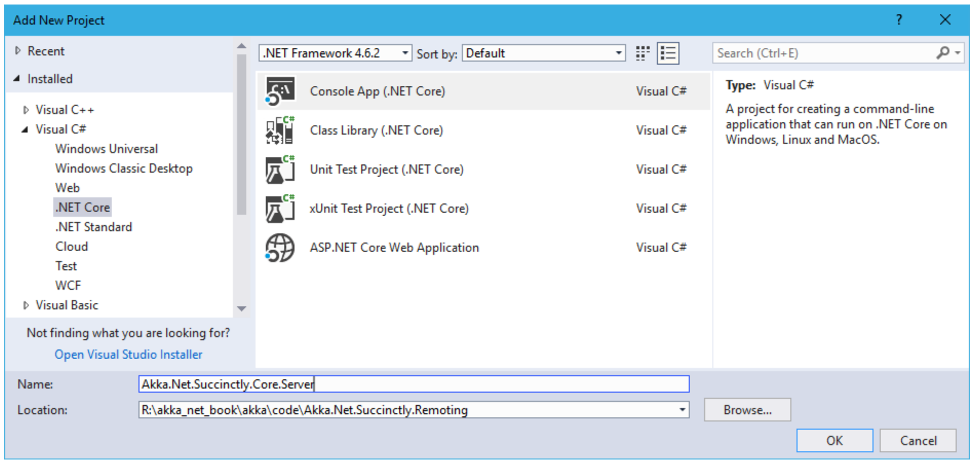
A project for creating a class library that targets .NET Core.

Name: Akka.Net.Succinctly.Core.Common

Location: R:\akka_net_book\akka\code\Akka.Net.Succinctly.Remoting

Figure 47: Adding a common library

As we can see in Figure 47, we are creating a .NET Core Library project called **`Akka.Net.Succinctly.Core.Common`**, and in addition to that, we need to create a .NET Core Console Application project called **`Akka.Net.Succinctly.Core.Server`**.


Add New Project
Recent
Installed
Visual C++
Visual C#
Windows Universal
Windows Classic Desktop
Web
.NET Core
.NET Standard
Cloud
Test
WCF
Visual Basic
Not finding what you are looking for?
Open Visual Studio Installer
.NET Framework 4.6.2
Sort by: Default
Search (Ctrl+E)
Console App (.NET Core) Visual C#
Class Library (.NET Core) Visual C#
Unit Test Project (.NET Core) Visual C#
xUnit Test Project (.NET Core) Visual C#
ASP.NET Core Web Application Visual C#
Browse...
OK
Cancel

Type: Visual C#

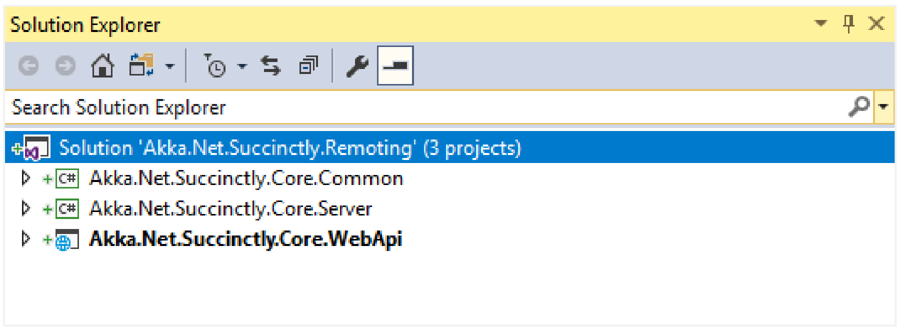
A project for creating a command-line application that can run on .NET Core on Windows, Linux and MacOS.

Name: Akka.Net.Succinctly.Core.Server

Location: R:\akka_net_book\akka\code\Akka.Net.Succinctly.Remoting

Figure 48: Creation of the Server

Now our Visual Studio Solution should contain three projects.


Solution Explorer

Search Solution Explorer

```
Solution 'Akka.Net.Succinctly.Remoting' (3 projects)
  Akka.Net.Succinctly.Core.Common
  Akka.Net.Succinctly.Core.Server
  Akka.Net.Succinctly.Core.WebApi
```

Figure 49: Three projects in Visual Studio

Akka.NET common library

We will start by defining the content of the common library. The common library at this point will contain the actor, messages, and other shared code. In production systems, we would probably like to separate the concerns better and not put everything into one bucket (library), but what we do here is good enough for a simple demo project.

We will start by adding the reference to the Akka library from NuGet.

Code Listing 83: Installing Akka package from NuGet

```
PM> Install-Package Akka
```

We also add the following code, which defines our **`CalculatorActor`**, **`AddMessage`**, and **`AnswerMessage`** classes. In bold, we have added the console logging of the currently processing message, which is important for debugging purposes, in order to see if the server received any message to be processed.

Code Listing 84: CalculatorActor and messages redefined

```
public class AddMessage
{
    public AddMessage(double term1, double term2)
    {
        Term1 = term1;
        Term2 = term2;
    }

    public double Term1;
    public double Term2;
}

public class AnswerMessage
{
    public AnswerMessage(double value)
    {
        Value = value;
    }

    public double Value;
}

public class CalculatorActor : ReceiveActor
{
    public CalculatorActor()
    {
        Receive<AddMessage>(add =>
        {
            Console.WriteLine($"{DateTime.Now}: Sum {add.Term1} +
{add.Term2}");
            Sender.Tell(new AnswerMessage(add.Term1 + add.Term2));
        });
    }
}
```

We will add another class that will be shared across the two solutions (client, server) and is responsible for reading out the Akka.NET configuration from the file. The **`HoconLoader`** class is a very simple class that reads the content of a file and returns the **`Akka.Configuration.Config`** object, which contains the object model for the HOCON string.

Code Listing 85: HOCON configuration file loader class

```
public static class HoconLoader
{
    public static Config FromFile(string path)
    {
        var hoconContent = System.IO.File.ReadAllText(path);
        return ConfigurationFactory.ParseString(hoconContent);
    }
}
```

Akka.NET server

External references

The server is probably the simplest of the three projects, as it only hosts the **`ActorSystem`**.

We will start by adding the references to **`Akka`** and **`Akka.Remote`** from NuGet.

Code Listing 90: Installing NuGet packages required for Akka remoting

```
PM> Install-Package Akka
PM> Install-Package Akka.Remote
```

In addition to this, we reference the already-created **`Akka.Net.Succinctly.Common`** project.

Configure remoting

The next step is to configure Akka to allow remote connections. In order to do this, we will create a new file in the console application and call it **`akka.net.hocon`**, where we place the necessary HOCON configuration for the remoting.

Do not forget to set the properties for this file in Visual Studio as:

- Build action = "Content"
- Copy to Output directory = "Copy always"

Code Listing 86: Content of the akka.net.hocon (server) config file

```
akka {
    actor {
        provider = "Akka.Remote.RemoteActorRefProvider, Akka.Remote"
    }

    remote {
        dot-netty.tcp {
           port = 8888 # bound to a specific port
           hostname = localhost
         }
    }
}
```

Notice that we have to specify the **`actor -> provider`** settings to **`Akka.Remote.RemoteActorRefProvider`**. This provider enables the actors to be deployed and instantiated remotely.

The next section to be configured is **`remote`**. Here we have to define the protocol supported for accessing this server application, the port, and the hostname. **`Dot-netty.tcp`** is the library that supports the remoting channel on the TCP protocol and the creating of sockets, in effect enabling the communication between the two actor systems. If you'd like to know a bit more about **`DotNetty`**, visit the official website.

`Port = 8888` defines the port at which the server will be responding. Being a server, this port has to be well known, and the client calling the server has to be aware of it.

Server's entry point

We are now ready to start the **`ActorSystem`** and inject the configuration previously defined.

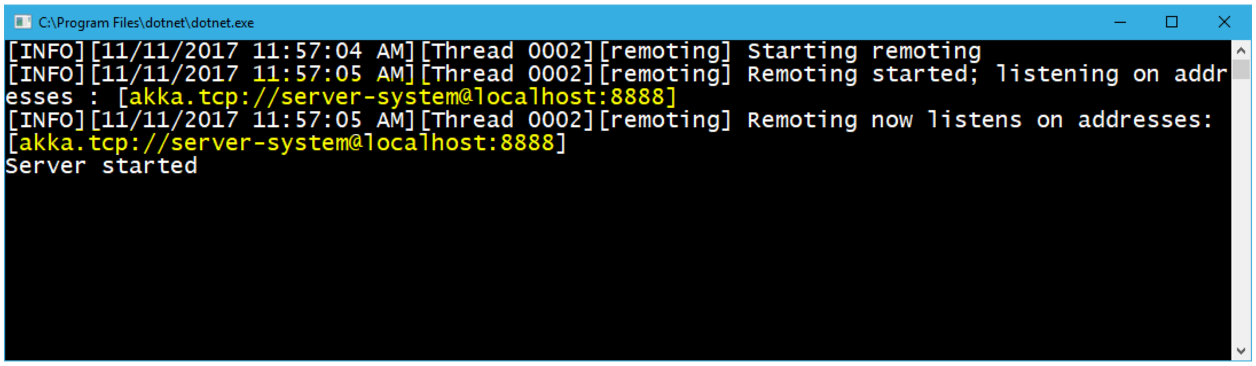
The entry point for the **`Server`** application simply loads the content of the **`akka.net.hocon`** file and injects the configuration settings into the **`ActorSystem`** at the time of the creation (**`Create`** method).

Code Listing 87: Server's main function definition

```
static void Main(string[] args)
{
    var hocon = HoconLoader.FromFile("akka.net.hocon");
    ActorSystem system = ActorSystem.Create("server-system", hocon);

    Console.WriteLine("Server started");

    Console.Read();
    system.Terminate().Wait();
}
```

Now we can run the **Server** (start the server application by pressing **F5**). As we can see in Figure 50, the application now listens to **Port 8888**, just as we have configured it.

Another important thing to notice is that the full path to the actor system is mentioned, which is **akka.tcp://server-system@localhost:8888**. This is very important because the client will need to know the full address as defined by the server.



```
C:\Program Files\dotnet\dotnet.exe
[INFO][11/11/2017 11:57:04 AM][Thread 0002][remoting] Starting remoting
[INFO][11/11/2017 11:57:05 AM][Thread 0002][remoting] Remoting started; listening on addr
esses : [akka.tcp://server-system@localhost:8888]
[INFO][11/11/2017 11:57:05 AM][Thread 0002][remoting] Remoting now listens on addresses:
[akka.tcp://server-system@localhost:8888]
Server started
```

Figure 50: Server is running, allowing remote connections

Akka.NET client

We have already mentioned that the client is our ASP.NET Core application, which we also have to configure to allow remoting.

External references

As we did for the **Server**, we need to reference the **Akka** and **Akka.Remote** libraries from NuGet.

Code Listing 93

```
PM> Install-Package Akka
PM> Install-Package Akka.Remote
```

And in addition to this, reference the already-created **Akka.Net.Succinctly.Common** project.

Configure remoting

The next step is to configure Akka to allow remote connections. In order to do this, we will add a file to the project and call it **`akka.net.hocon`**. As we did for the **`Server`**, this file will contain the configuration of the client **`ActorSystem`**.

In Visual Studio, do not forget to set the properties for this file as:

- Build action = "Content"
- Copy to Output directory = "Copy always"

Code Listing 88: Content of the akka.net.hocon (client) config file

```
akka {
    actor {
        provider = "Akka.Remote.RemoteActorRefProvider, Akka.Remote"
            deployment {
                /calculator {
                    remote = "akka.tcp://server-system@localhost:8888"
                }
            }
        }

    remote {
        dot-netty.tcp {
            port = 0 # bound to a dynamic port assigned by the OS
            hostname = localhost
        }
    }
}
```

The actor provider part looks exactly the same as when we configured the server by setting the **`RemoteActorRefProvider`** as actor provider.

The interesting part is the fact that we have to configure the deployment section. In the deployment section, we can place the configuration per actor (name) and define where remotely we would like to run this actor. So, **`/calculator`** is the name of the actor (no need to specify **`/user`** before it), and we are instructing Akka to instantiate this actor on a remote system defined by the path. We can also specify multiple actors to point to multiple servers.

Code Listing 89: Specifying more than one actor in deployment section

```
 actor {
   provider = "Akka.Remote.RemoteActorRefProvider, Akka.Remote"
       deployment {
           /calculator {
               remote = "akka.tcp://server-system@localhost:8888"
           }
           /actorXXX{
               remote = "akka.tcp://server-system@system2:8888"
```

```
            }
        }
}
```

The next section to be configured is **`remote`**. Again, as we did for the **`Server`**, we are configuring **`dot-netty.tcp`** as the transport protocol, with the difference of the **`Port`**. We don't have to specify the port for the client, since the port will be automatically assigned by Akka and transmitted to **`Server`** so that **`Server`** knows where to answer back. So, **`port = 0`** means dynamic address.

Client's entry point

In respect to the previous solution, as defined in Chapter 13, we don't have to make any changes to the **`CalculatorController`**, **`CalculatorActorInstance`**, and **`ICalculatorActorInstance`** definitions. Since we now have the **`CalculatorActor`** in the **`Common`** library, the only part that will change is the **`using`** section in the previously mentioned classes.

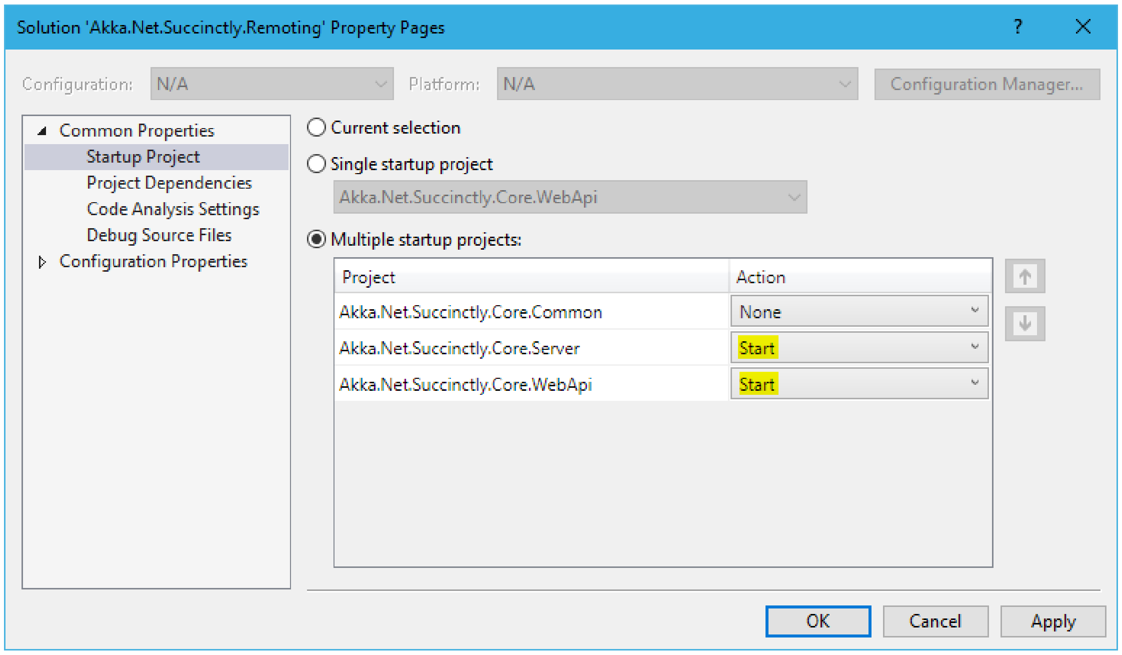
The only part that really changes—and this is the beauty of this solution—is the configuration of the actor system. We need to change the **`ConfigureServices`** method in the **`Startup`** class so that it includes the reading of the HOCON configuration, and inject that into the **`ActorSystem`** creation method.

The changed parts are highlighted in bold.

Code Listing 90: Configuration of the client—Injecting HOCON config into the ActorSystem

```
public void ConfigureServices(IServiceCollection services)
{
    var hocon = HoconLoader.FromFile("akka.net.hocon");
    var actorSystem = ActorSystem.Create("calculator-actor-system", hocon);

    services.AddSingleton(typeof(ActorSystem), (serviceProvider) =>
actorSystem);
    services.AddSingleton(typeof(ICalculatorActorInstance),
typeof(CalculatorActorInstance));
    services.AddMvc();
}
```

Before running the application, we have to Visual Studio up so that it starts the server before starting the client project. To do so, right-click on the solution name, and choose **Properties**. As shown in Figure 51, we select the **Multiple startup projects option** and set **`Akka.Net.Succintly.Core.Server`** and **`Akka.Net.Succinctly.Core.WebApi`** to **Start**.



Solution 'Akka.Net.Succinctly.Remoting' Property Pages

Configuration: N/A   Platform: N/A   Configuration Manager...

- Common Properties
  - Startup Project
  - Project Dependencies
  - Code Analysis Settings
  - Debug Source Files
- Configuration Properties

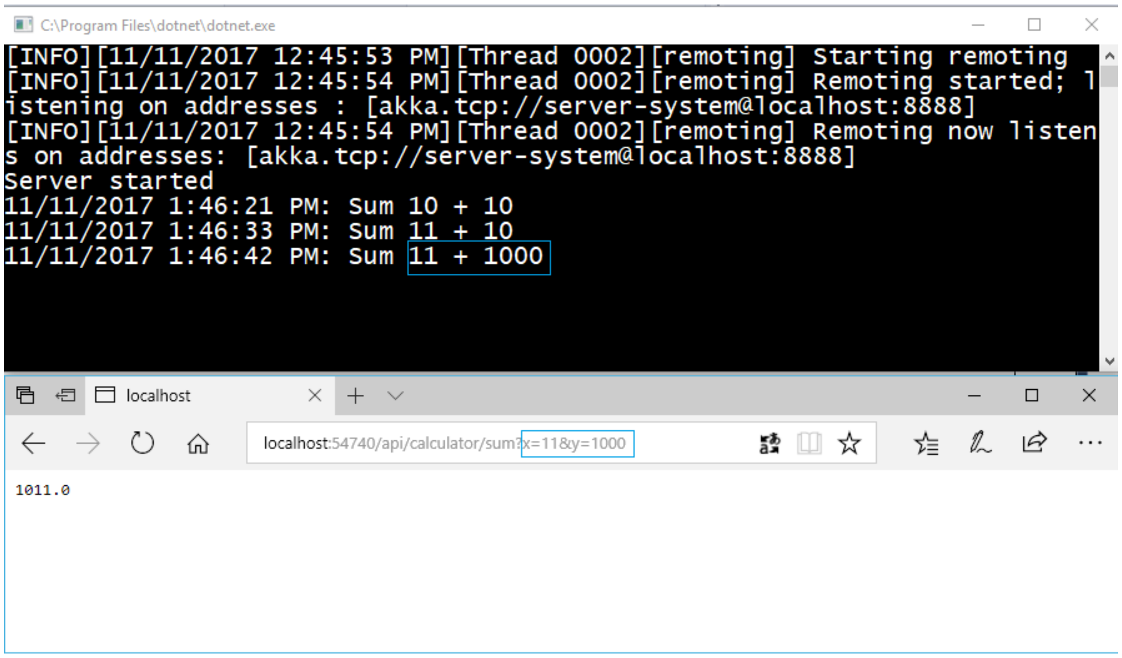
Current selection
Single startup project
Akka.Net.Succinctly.Core.WebApi
[✓ Multiple startup projects:]

Project	Action
Akka.Net.Succinctly.Core.Common	None
Akka.Net.Succinctly.Core.Server	Start
Akka.Net.Succinctly.Core.WebApi	Start

Figure 51: Setting up multiple starting projects

Now we are ready to run the application—press **F5** in Visual Studio. The server and the browser will open, and we will be ready to query our **`Calculator`** controller.

By placing the console application (server) and the browser (client) side by side, we can see that every time we query the browser, the console will show an entry. This is evidence that communication between the two systems is working properly.



C:\Program Files\dotnet\dotnet.exe

```
[INFO][11/11/2017 12:45:53 PM][Thread 0002][remoting] Starting remoting
[INFO][11/11/2017 12:45:54 PM][Thread 0002][remoting] Remoting started; l
istening on addresses : [akka.tcp://server-system@localhost:8888]
[INFO][11/11/2017 12:45:54 PM][Thread 0002][remoting] Remoting now listen
s on addresses: [akka.tcp://server-system@localhost:8888]
Server started
11/11/2017 1:46:21 PM: Sum 10 + 10
11/11/2017 1:46:33 PM: Sum 11 + 10
11/11/2017 1:46:42 PM: Sum 11 + 1000
```

localhost:54740/api/calculator/sum?x=11&y=1000

```
1011.0
```

Figure 52: Communication between the client and the server

Final Words

In this book, which was a great challenge for me, we have touched upon some of the most important aspects of the Akka.NET framework:

- Introduction to the actor model.
- Working with actors.
- Testing actors.
- Remoting actors and applying different strategies for routing.

Unfortunately, the format of this book didn't allow me to talk about Akka.NET persistence, Akka.NET clustering, and Akka.NET streams, which are some more advanced topics that would enable the creation of real enterprise-scale applications, fully scalable and persisted. That said, with the baseline provided in this book, you can learn those topics without big issues, since they are based on the same concepts we have already seen.

As with other books in the Syncfusion *Succinctly* series, what is mentioned in this book should give you a good starting point not only for designing an application, but also to start exploring more advanced options on your own.

Thank you for reading this book—I hope I've managed to fulfill the expectations you might have had when you started it. It certainly helped me to clear up some ideas of my own about the framework: I learned a lot about this fantastic framework during this journey.