THE

BOOTCAMP

BY

AARON STANNARD & ANDREW SKOTZKO

Table of Contents

Akka.NET Bootcamp

Welcome to Akka.NET Bootcamp! This is a free, self-directed learning course brought to you by the folks at Petabridge.

Over the three units of this bootcamp you will learn how to create fully-functional, real-world programs using Akka.NET actors and many other parts of the core Akka.NET framework!

We will start with some basic actors and have you progressively work your way up to larger, more sophisticated examples.

The course is self-directed learning. You can do it at whatever pace you wish. You can sign up here to have one Akka.NET Bootcamp lesson emailed to you daily if you'd like a little help pacing yourself throughout the course.

> NOTE: F# support is in progress (see the FSharp branch). We will happily accept F# pull requests. Feel free to send them in.

What will you learn?

In Akka.NET Bootcamp you will learn how to use Akka.NET actors to build reactive, concurrent applications.

You will learn how to build types of applications that may have seemed impossible or really, really hard to make prior to learning Akka.NET. You will walk away from this bootcamp with the confidence to handle bigger and harder problems than ever before!

Unit 1

In Unit 1, we will learn the fundamentals of how the actor model and Akka.NET work.

*NIX systems have the `tail` command built-in to monitor changes to a file (such as tailing log files), whereas Windows does not. We will recreate `tail` for Windows and use the process to learn the fundamentals.

In Unit 1 you will learn:

1. How to create your own `ActorSystem` and actors;
2. How to send messages actors and how to handle different types of messages;
3. How to use `Props` and `IActorRef`s to build loosely coupled systems.
4. How to use actor paths, addresses, and `ActorSelection` to send messages to actors.
5. How to create child actors and actor hierarchies, and how to supervise children with `SupervisionStrategy`.
6. How to use the Actor lifecycle to control actor startup, shutdown, and restart behavior.

Begin Unit 1.

Unit 2

In Unit 2, we're going to get into some more of the intermediate Akka.NET features to build a more sophisticated application than what we accomplished at the end of unit 1.

In Unit 2 you will learn:

1. How to use HOCON configuration to configure your actors via App.config and Web.config;
2. How to configure your actor's Dispatcher to run on the Windows Forms UI thread, so actors can make operations directly on UI elements without needing to change contexts;
3. How to handle more sophisticated types of pattern matching using `ReceiveActor`;
4. How to use the `Scheduler` to send recurring messages to actors;
5. How to use the Publish-subscribe (pub-sub) pattern between actors;
6. How and why to switch actor's behavior at run-time; and
7. How to `Stash` messages for deferred processing.

Begin Unit 2.

Unit 3

In Unit 3, we will learn how to use actors for parallelism and scale-out using Octokit and data from Github repos!

In Unit 3 you will learn:

1. How to perform work asynchronously inside your actors using `PipeTo`;
2. How to use `Ask` to wait inline for actors to respond to your messages;
3. How to use `ReceiveTimeout` to time out replies from other actors;
4. How to use `Group` routers to divide work among your actors;
5. How to use `Pool` routers to automatically create and manage pools of actors; and
6. How to use HOCON to configure your routers.

Begin Unit 3.

How to get started

Here's how Akka.NET bootcamp works!

Use Github to Make Life Easy

This Github repository contains Visual Studio solution files and other assets you will need to complete the bootcamp.

Thus, if you want to follow the bootcamp we recommend doing the following:

1. Sign up for Github, if you haven't already.
2. Fork this repository and clone your fork to your local machine.
3. As you go through the project, keep a web browser tab open to the Akka.NET Bootcamp ReadMes so you can read all of the instructions clearly and easily.

Bootcamp Structure

Akka.NET Bootcamp consists of three modules:

- **Unit 1 - Beginning Akka.NET**
- **Unit 2 - Intermediate Akka.NET**

- **Unit 3 - Advanced Akka.NET**

Each module contains the following structure (using **Unit 1** as an example:)

```
src\Unit1\README.MD - table of contents and instructions for the module
src\Unit1\DoThis\ - contains the .SLN and project files that you will use through all lesson
-- lesson 1
src\Unit1\Lesson1\README.MD - README explaining lesson1
src\Unit1\Lesson1\DoThis\ - C# classes, images, text files, and other junk you'll need to co
src\Unit1\Lesson1\Completed\ - Got stuck on lesson1? This folder shows the "expected" output
-- repeat for all lessons
```

Start with the first lesson in each unit and follow the links through their README files on Github. We're going to begin with **Unit 1, Lesson 1**.

Lesson Layout

Each Akka.NET Bootcamp lesson contains a README which explains the following:

1. The Akka.NET concepts and tools you will be applying in the lesson, along with links to any relevant documentation or examples
2. Step-by-step instructions on how to modify the .NET project inside the `Unit-[Num]/DoThis/` to match the expected output at the end of each lesson.
3. If you get stuck following the step-by-step instructions, each lesson contains its own `/Completed/` folder that shows the full source code that will produce the expected output. You can compare this against your own code and see what you need to do differently.

When you're doing the lessons...

A few things to bear in mind when you're following the step-by-step instructions:

1. **Don't just copy and paste the code shown in the lesson's README**. You'll retain and learn all of the built-in Akka.NET functions if you type out the code as it's shown. Kinesthetic learning FTW!
2. **You might be required to fill in some blanks during individual lessons.** Part of helping you learn Akka.NET involves leaving some parts of the exercise up to you - if you ever feel lost, always check the contents of the `/Completed` folder for that

lesson.
3. **Don't be afraid to ask questions**. You can reach the Petabridge and Akka.NET teams in our Gitter chat here.

Docs

We will provide explanations of all key concepts throughout each lesson, but of course, you should bookmark (and feel free to use!) the Akka.NET docs.

Tools / prerequisites

This course expects the following:

- You have some programming experience and familiarity with C#
- A Github account and basic knowledge of Git.
- You are using a version of Visual Studio (it's free now!)
    - We haven't had a chance to test these in Xamarin / on Mono yet, but that will be coming soon. If you try them there, please let us know how it goes! We are planning on having everything on all platforms ASAP.

Enough talk, let's go!

Let's begin!

About Petabridge

Petabridge is a company dedicated to making it easier for .NET developers to build distributed applications.

Petabridge also offers advanced Akka.NET training, so please check out our advanced courses on Akka.Cluster, Akka.Remote, and Akka.NET Design Patterns and let's see what we can create together!

Akka.NET Bootcamp - Unit 1: Beginning Akka.NET

In Unit 1, we will learn the fundamentals of how the actor model and Akka.NET work.

Concepts you'll learn

*NIX systems have the `tail` command built-in to monitor changes to a file (such as tailing log files), whereas Windows does not. We will recreate `tail` for Windows, and use the process to learn the fundamentals.

In Unit 1 you will learn the following:

1. How to create your own `ActorSystem` and actors;
2. How to send messages actors and how to handle different types of messages;
3. How to use `Props` and `IActorRef`s to build loosely coupled systems.
4. How to use actor paths, addresses, and `ActorSelection` to send messages to actors.
5. How to create child actors and actor hierarchies, and how to supervise children with `SupervisionStrategy`.
6. How to use the Actor lifecycle to control actor startup, shutdown, and restart behavior.

Using Xamarin?

Since Unit 1 relies heavily on the console, you'll need to make a small tweaks before beginning. You need to set up your `WinTail` project file (not the solution) to use an **external console**.

To set this up:

1. Click on the `WinTail` project (not the solution)
2. Navigate to `Project > WinTail Options` in the menu
3. Inside `WinTail Options`, navigate to `Run > General`
4. Select `Run on external console`
5. Click `OK`

Here is a demonstration of how to set it up:

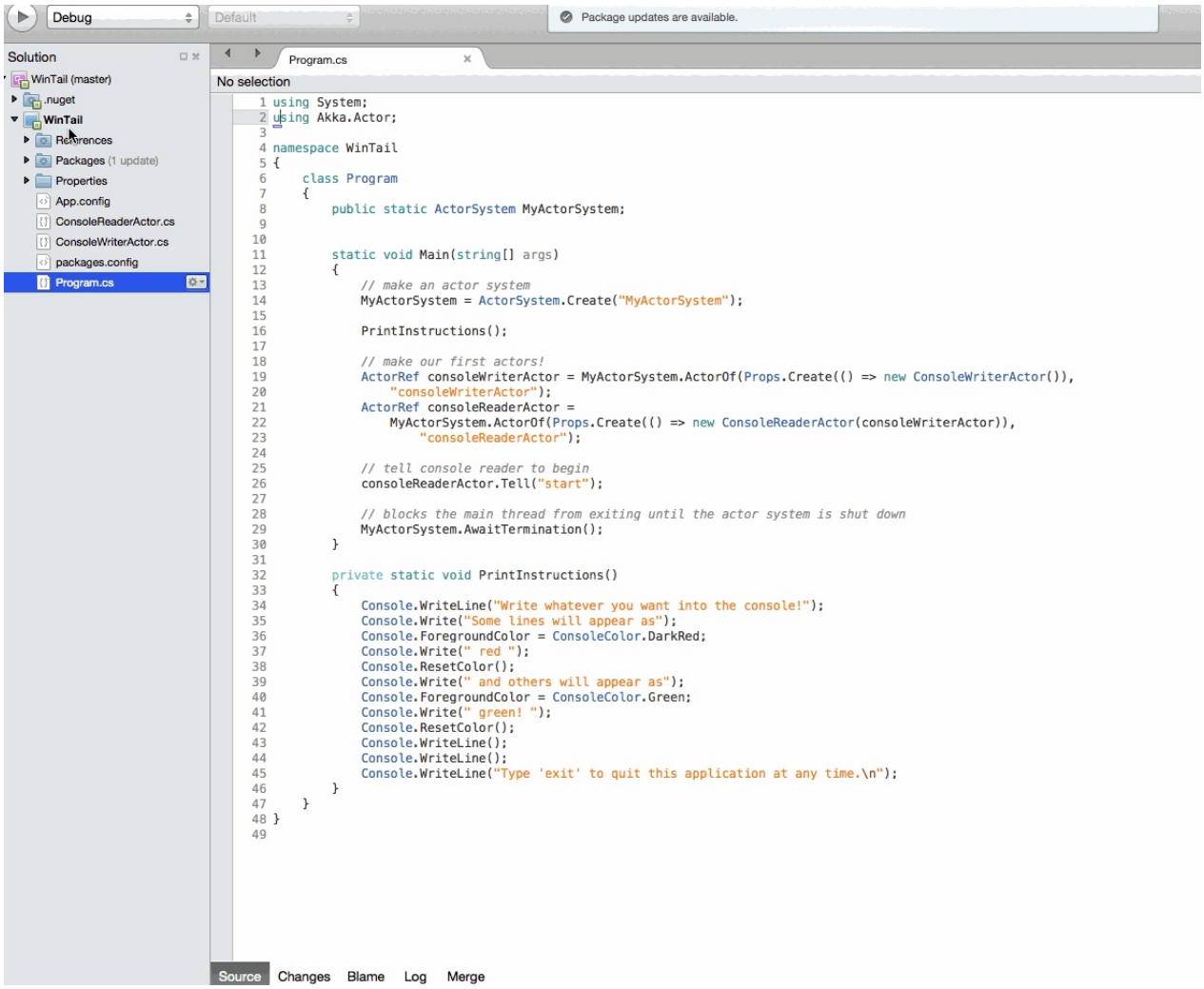


Table of Contents

Get Started

To get started, go to the /DoThis/ folder and open `WinTail.sln`.

And then go to Lesson 1.

Lesson 1.1: Actors and the `ActorSystem`

Here we go! Welcome to lesson 1.

In this lesson, you will make your first actors and be introduced to the fundamentals of Akka.NET.

Key concepts / background

In this first lesson, you will learn the basics by creating a console app with your first actor system and simple actors within it.

We will be creating two actors, one to read from the console, and one to write to it after doing some basic processing.

What is an actor?

An "actor" is really just an analog for a human participant in a system. It's an entity, an object, that can do things and communicate.

> We're going to assume that you're familiar with object-oriented programming (OOP). The actor model is very similar to object-oriented programming (OOP) - just like how everything is an object in OOP, in the actor model ***everything is an actor***.

Repeat this train of thought to yourself: everything is an actor. Everything is an actor. Everything is an actor! Think of designing your system like a hierarchy of people, with tasks being split up and delegated until they become small enough to be handled concisely by one actor.

For now, we suggest you think of it like this: in OOP, you try to give every object a single, well-defined purpose, right? Well, the actor model is no different, except now the objects that you give a clear purpose to just happen to be actors.

Further reading: What is an Akka.NET Actor?

How do actors communicate?

Actors communicate with each other just as humans do, by exchanging messages. These messages are just plain old C# classes.

```
//this is a message!
public class SomeMessage{
    public int SomeValue {get; set}
}
```

We go into messages in detail in the next lesson, so don't worry about it for now. All you need to know is that you send messages by `Tell()`ing them to another actor.

```
//send a string to an actor
someActorRef.Tell("this is a message too!");
```

What can an actor do?

Anything you can code. Really :)

You code actors to handle messages they receive, and actors can do whatever you need them to in order to handle a message. Talk to a database, write to a file, change an internal variable, or anything else you might need.

In addition to processing a message it receives, an actor can:

1. Create other actors
2. Send messages to other actors (such as the `Sender` of the current message)
3. Change its own behavior and process the next message it receives differently

Actors are inherently asynchronous (more on this in a future lesson), and there is nothing about the Actor Model that says which of the above an actor must do, or the order it has to do them in. It's up to you.

What kinds of actors are there?

All types of actors inherit from `UntypedActor`, but don't worry about that now. We'll cover

different actor types later.

In Unit 1 all of your actors will inherit from `UntypedActor`.

How do you make an actor?

There are 2 key things to know about creating an actor:

1. All actors are created within a certain context. That is, they are "actor of" a context.
2. Actors need `Props` to be created. A `Props` object is just an object that encapsulates the formula for making a given kind of actor.

We'll be going into `Props` in depth in lesson 3, so for now don't worry about it much. We've provided the `Props` for you in the code, so you just have to figure out how to use `Props` to make an actor.

The hint we'll give you is that your first actors will be created within the context of your actor system itself. See the exercise instructions for more.

What is an `ActorSystem`?

An `ActorSystem` is a reference to the underlying system and Akka.NET framework. All actors live within the context of this actor system. You'll need to create your first actors from the context of this `ActorSystem`.

By the way, the `ActorSystem` is a heavy object: create only one per application.

Aaaaaaand... go! That's enough conceptual stuff for now, so dive right in and make your first actors.

Exercise

Let's dive in!

> Note: Within the sample code there are sections clearly marked `"YOU NEED TO FILL IN HERE"` - find those regions of code and begin filling them in with the appropriate functionality in order to complete your goals.

Launch the fill-in-the-blank sample

Go to the DoThis folder and open WinTail in Visual Studio. The solution consists of a simple console application and only one Visual Studio project file.

You will use this solution file through all of Unit 1.

Install the latest Akka.NET NuGet package

In the Package Manager Console, type the following command:

```
Install-Package Akka
```

This will install the latest Akka.NET binaries, which you will need in order to compile this sample.

Then you'll need to add the `using` namespace to the top of `Program.cs`:

```
// in Program.cs
using Akka.Actor;
```

Make your first `ActorSystem`

Go to `Program.cs` and add this to create your first actor system:

```
MyActorSystem = ActorSystem.Create("MyActorSystem");
```

>

> **NOTE:** When creating `Props`, `ActorSystem`, or `ActorRef` you will very rarely see the `new` keyword. These objects must be created through the factory methods built into Akka.NET. If you're using `new` you might be making a mistake.

Make ConsoleReaderActor & ConsoleWriterActor

The actor classes themselves are already defined, but you will have to make your first actors.

Again, in `Program.cs`, add this just below where you made your `ActorSystem`:

```
var consoleWriterActor = MyActorSystem.ActorOf(Props.Create(() => new ConsoleWriterActor()))
var consoleReaderActor = MyActorSystem.ActorOf(Props.Create(() => new ConsoleReaderActor(con
```

We will get into the details of `Props` and `ActorRef`s in lesson 3, so don't worry about them much for now. Just know that this is how you make an actor.

Have ConsoleReaderActor Send a Message to ConsoleWriterActor

Time to put your first actors to work!

You will need to do the following:

1. ConsoleReaderActor is set up to read from the console. Have it send a message to ConsoleWriterActor containing the content that it just read.

   ```
   // in ConsoleReaderActor.cs
   _consoleWriterActor.Tell(read);
   ```

2. Have ConsoleReaderActor send a message to itself after sending a message to ConsoleWriterActor. This is what keeps the read loop going.

   ```
   // in ConsoleReaderActor.cs
   Self.Tell("continue");
   ```

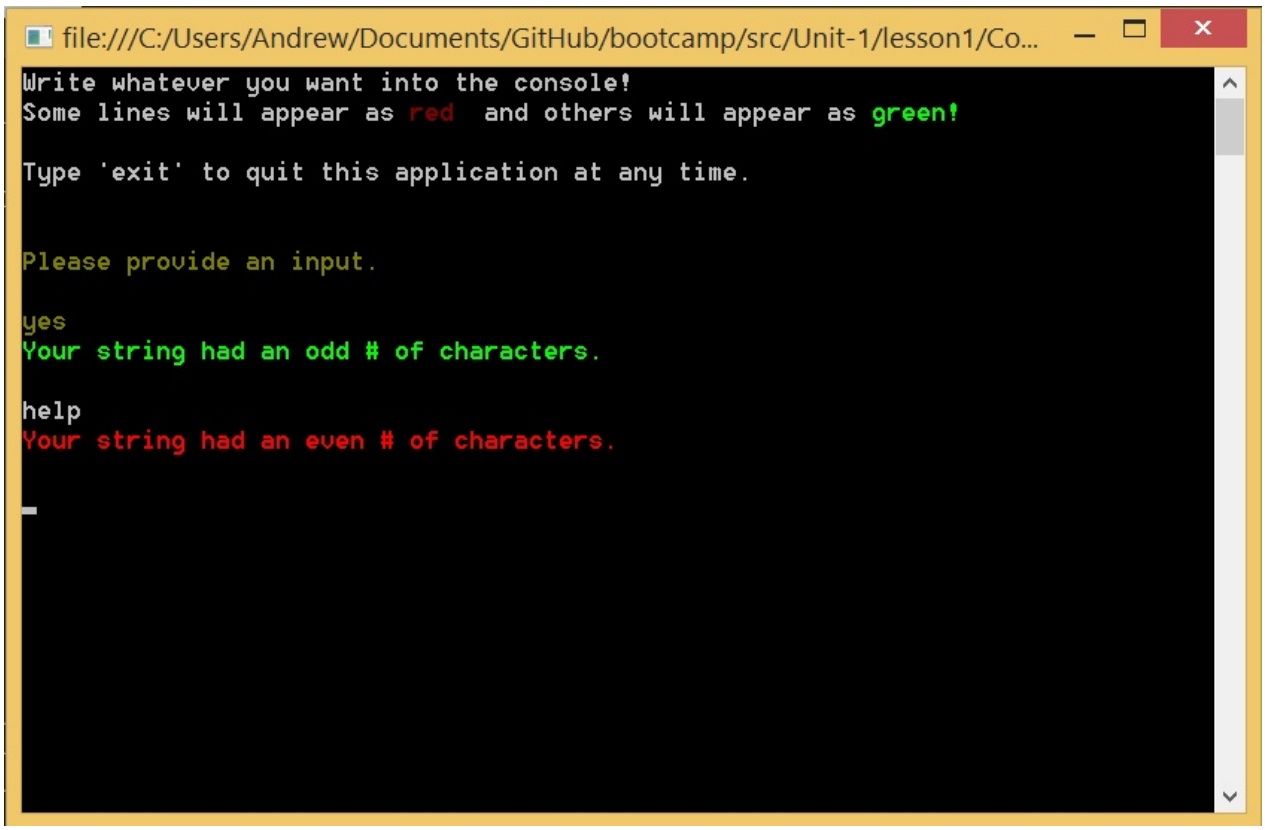
3. Send an initial message to ConsoleReaderActor in order to get it to start reading from the console.

   ```
   // in Program.cs
   consoleReaderActor.Tell("start");
   ```

Step 5: Build and Run!

Once you've made your edits, press `F5` to compile and run the sample in Visual Studio.

You should see something like this, when it is working correctly:



Once you're done

Compare your code to the code in the Completed folder to see what the instructors included in their samples.

Great job! Onto Lesson 2!

Awesome work! Well done on completing your first lesson.

Let's move onto Lesson 2 - Defining and Handling Different Types of Messages.

Any questions?

Don't be afraid to ask questions :).

Come ask any questions you have, big or small, in this ongoing Bootcamp chat with the Petabridge & Akka.NET teams.

Problems with the code?

If there is a problem with the code running, or something else that needs to be fixed in this lesson, please create an issue and we'll get right on it. This will benefit everyone going through Bootcamp.

Lesson 1.2: Defining and Handling Messages

In this lesson, you will make your own message types and use learn how to control processing flow within your actors based on your custom messages. Doing so will teach you the fundamentals of communicating in a message- and event-driven manner within your actor system.

This lesson picks up right where Lesson 1 left off, and continues extending our budding systems of console actors. In addition to defining our own messages, we'll also add some simple validation for the input we enter and take action based on the results of that validation.

Key concepts / background

What is a message?

Any POCO can be a message. A message can be a `string`, a value like `int`, a type, an object that implements an interface... whatever you want.

That being said, the recommended approach is to make your own custom messages into semantically named classes, and to encapsulate any state you want inside those classes (e.g. store a `Reason` inside a `ValidationFailed` class... hint, hint...).

How do I send an actor a message?

As you saw in the first lesson, you `Tell()` the actor the message.

How do I handle a message?

This is entirely up to you, and doesn't really have much to do with Akka.NET. You can handle (or not handle) a message as you choose within an actor.

What happens if my actor receives a message it doesn't

know how to handle?

Actors ignore messages they don't know how to handle. Whether or not this ignored message is logged as such depends on the type of actor.

With an `UntypedActor`, unhandled messages are not logged as unhandled unless you manually mark them as such, like so:

```
class MyActor : UntypedActor
{
    protected override void OnReceive(object message)
    {
         if (message is Messages.InputError)
        {
            var msg = message as Messages.InputError;
            Console.ForegroundColor = ConsoleColor.Red;
            Console.WriteLine(msg.Reason);
        }
        else
        {
            Unhandled(message);
        }
    }
}
```

However, in a `ReceiveActor`—which we cover in Unit 2—unhandled messages are automatically sent to `Unhandled` so the logging is done for you.

How do my actors respond to messages?

This is up to you - you can respond by simply processing the message, replying to the `Sender`, forwarding the message onto another actor, or doing nothing at all.

> **NOTE:** Whenever your actor receives a message, it will always have the sender of the current message available via the `Sender` property inside your actor.

Exercise

In this exercise, we will introduce some basic validation into our system. We will then use custom message types to signal the results of that validation back to the user.

Phase 1: Define your own message types

Add a new class called `Messages` and the corresponding file, `Messages.cs`.

This is the class we'll use to define system-level messages that we can use to signal events. The pattern we'll be using is to turn events into messages. That is, when an event occurs, we will send an appropriate message class to the actor(s) that need to know about it, and then listen for / respond to that message as needed in the receiving actors.

Add regions for each message type

Add three regions for different types of messages to the file. Next we'll be creating our own message classes that we'll use to signify events.

```
// in Messages.cs
#region Neutral/system messages
#endregion

#region Success messages
#endregion

#region Error messages
#endregion
```

In these regions we will define custom message types to signal these situations:

```
- user provided blank input
- user provided invalid input
- user provided valid input
```

Make `ContinueProcessing` message

Define a marker message class in the `Neutral/system messages` region that we'll use to signal to continue processing (the "blank input" case):

```
// in Messages.cs
```

```
#region Neutral/system messages
/// <summary>
/// Marker class to continue processing.
/// </summary>
public class ContinueProcessing { }
#endregion
```

Make `InputSuccess` message

Define an `InputSuccess` class in the `Success messages` region. We'll use this to signal that the user's input was good and passed validation (the "valid input" case):

```
#region Success messages
// in Messages.cs
/// <summary>
/// Base class for signalling that user input was valid.
/// </summary>
public class InputSuccess
{
    public InputSuccess(string reason)
    {
        Reason = reason;
    }

    public string Reason { get; private set; }
}
#endregion
```

Make `InputError` messages

Define the following `InputError` classes in the `Error messages` region. We'll use these messages to signal invalid input occurring (the "invalid input" cases):

```
// in Messages.cs
#region Error messages
/// <summary>
/// Base class for signalling that user input was invalid.
/// </summary>
public class InputError
{
    public InputError(string reason)
    {
        Reason = reason;
    }

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

```
}

/// <summary>
/// User provided blank input.
/// </summary>
public class NullInputError : InputError
{
    public NullInputError(string reason) : base(reason) { }
}

/// <summary>
/// User provided invalid input (currently, input w/ odd # chars)
/// </summary>
public class ValidationError : InputError
{
    public ValidationError(string reason) : base(reason) { }
}
#endregion
```

> **NOTE:** You can compare your final `Messages.cs` to Messages.cs to make sure you're set up right before we go on.

Phase 2: Turn events into messages and send them

Great! Now that we've got messages classes set up to wrap our events, let's use them in `ConsoleReaderActor` and `ConsoleWriterActor`.

Update `ConsoleReaderActor`

Add the following internal message type to `ConsoleReaderActor`:

```
// in ConsoleReaderActor
public const string StartCommand = "start";
```

Update the `Main` method to use `ConsoleReaderActor.StartCommand`:

Replace this:

```
// in Program.cs
// tell console reader to begin
consoleReaderActor.Tell("start");
```

with this:

```
// in Program.cs
// tell console reader to begin
consoleReaderActor.Tell(ConsoleReaderActor.StartCommand);
```

Replace the `OnReceive` method of `ConsoleReaderActor` as follows. Notice that we're now listening for our custom `InputError` messages, and taking action when we get an error.

```
// in ConsoleReaderActor
protected override void OnReceive(object message)
{
    if (message.Equals(StartCommand))
    {
        DoPrintInstructions();
    }
    else if (message is Messages.InputError)
    {
        _consoleWriterActor.Tell(message as Messages.InputError);
    }

    GetAndValidateInput();
}
```

While we're at it, let's add `DoPrintInstructions()`, `GetAndValidateInput()`, `IsValid()` to `ConsoleReaderActor`. These are internal methods that our `ConsoleReaderActor` will use to get input from the console and determine if it is valid. (Currently, "valid" just means that the input had an even number of characters. It's an arbitrary placeholder.)

```
// in ConsoleReaderActor, after OnReceive()
#region Internal methods
private void DoPrintInstructions()
{
    Console.WriteLine("Write whatever you want into the console!");
    Console.WriteLine("Some entries will pass validation, and some won't...\n\n");
    Console.WriteLine("Type 'exit' to quit this application at any time.\n");
}

/// <summary>
/// Reads input from console, validates it, then signals appropriate response
/// (continue processing, error, success, etc.).
/// </summary>
private void GetAndValidateInput()
{
    var message = Console.ReadLine();
```

```
        if (string.IsNullOrEmpty(message))
        {
            // signal that the user needs to supply an input, as previously
            // received input was blank
            Self.Tell(new Messages.NullInputError("No input received."));
        }
        else if (String.Equals(message, ExitCommand, StringComparison.OrdinalIgnoreCase))
        {
            // shut down the entire actor system (allows the process to exit)
            Context.System.Shutdown();
        }
        else
        {
            var valid = IsValid(message);
            if (valid)
            {
                _consoleWriterActor.Tell(new Messages.InputSuccess("Thank you! Message was valid

                // continue reading messages from console
                Self.Tell(new Messages.ContinueProcessing());
            }
            else
            {
                Self.Tell(new Messages.ValidationError("Invalid: input had odd number of charact
            }
        }
    }

    /// <summary>
    /// Validates <see cref="message"/>.
    /// Currently says messages are valid if contain even number of characters.
    /// </summary>
    /// <param name="message"></param>
    /// <returns></returns>
    private static bool IsValid(string message)
    {
        var valid = message.Length % 2 == 0;
        return valid;
    }
    #endregion
```

Update `Program`

First, remove the definition and call to `PrintInstructions()` from `Program.cs`.

Now that `ConsoleReaderActor` has its own well-defined `StartCommand`, let's go ahead and use that instead of hardcoding the string "start" into the message.

As a quick checkpoint, your `Main()` should now look like this:

```
static void Main(string[] args)
{
    // initialize MyActorSystem
    MyActorSystem = ActorSystem.Create("MyActorSystem");

    var consoleWriterActor = MyActorSystem.ActorOf(Props.Create(() => new ConsoleWriterActor
    var consoleReaderActor = MyActorSystem.ActorOf(Props.Create(() => new ConsoleReaderActor

    // tell console reader to begin
    consoleReaderActor.Tell(ConsoleReaderActor.StartCommand);

    // blocks the main thread from exiting until the actor system is shut down
    MyActorSystem.AwaitTermination();
}
```

Not much has changed here, just a bit of cleanup.

Update `ConsoleWriterActor`

Now, let's get `ConsoleWriterActor` to handle these new types of messages.

Change the `OnReceive` method of `ConsoleWriterActor` as follows:

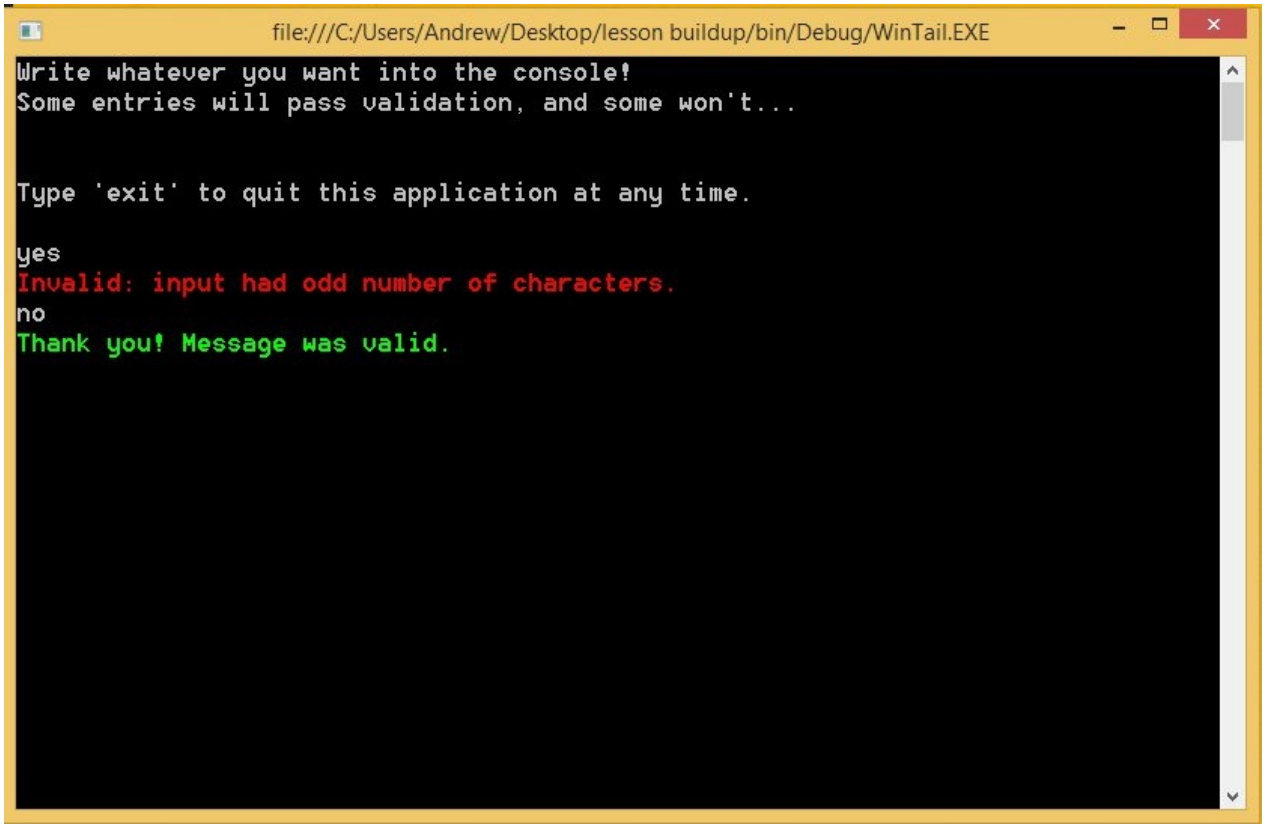
```
// in ConsoleWriterActor.cs
protected override void OnReceive(object message)
{
    if (message is Messages.InputError)
    {
        var msg = message as Messages.InputError;
        Console.ForegroundColor = ConsoleColor.Red;
        Console.WriteLine(msg.Reason);
    }
    else if (message is Messages.InputSuccess)
    {
        var msg = message as Messages.InputSuccess;
        Console.ForegroundColor = ConsoleColor.Green;
        Console.WriteLine(msg.Reason);
    }
    else
    {
        Console.WriteLine(message);
    }

    Console.ResetColor();
}
```

As you can see here, we are making `ConsoleWriterActor` pattern match against the type of message it receives, and take different actions according to what type of message it receives.

Phase 3: Build and run!

You should now have everything you need in place to be able to build and run. Give it a try!

If everything is working as it should, you should see an output like this:



Once you're done

Compare your code to the solution in the Completed folder to see what the instructors included in their samples.

Great job! Onto Lesson 3!

Awesome work! Well done on completing this lesson.

Let's move onto Lesson 3 - `Props` and `IActorRef`s.

Any questions?

Don't be afraid to ask questions :).

Come ask any questions you have, big or small, in this ongoing Bootcamp chat with the Petabridge & Akka.NET teams.

Problems with the code?

If there is a problem with the code running, or something else that needs to be fixed in this lesson, please create an issue and we'll get right on it. This will benefit everyone going through Bootcamp.

Lesson 1.3: `Props` and `IActorRef`s

In this lesson, we will review/reinforce the different ways you can create actors and send them messages. This lesson is more conceptual and has less coding for you to do, but it's an essential foundation and key to understanding the code you will see down the line.

In this lesson, the code has changed a bit. The change is that the `ConsoleReaderActor` no longer does any validation work, but instead, just passes off the messages it receives from the console to another actor for validation (the `ValidationActor`).

Key concepts / background

`IActorRef`s

What is an `IActorRef`?

An `IActorRef` is a reference or handle to an actor. The purpose of an `IActorRef` is to support sending messages to an actor through the `ActorSystem`. You never talk directly to an actor—you send messages to its `IActorRef` and the `ActorSystem` takes care of delivering those messages for you.

WTF? I don't actually talk to my actors? Why not?

You do talk to them, just not directly :) You have to talk to them via the intermediary of the `ActorSystem`.

Here are two of the reasons why it is an advantage to send messages to an `IActorRef` and let the underlying `ActorSystem` do the work of getting the messages to the actual actor.

- It gives you better information to work with and messaging semantics. The `ActorSystem` wraps all messages in an `Envelope` that contains metadata about the message. This metadata is automatically unpacked and made available in the context of your actor.
- It allows "location transparency": this is a fancy way of saying that you don't have to

> worry about which process or machine your actor lives in. Keeping track of all this is the system's job. This is essential for allowing remote actors, which is how you can scale an actor system up to handle massive amounts of data (e.g. have it work on multiple machines in a cluster). More on this later.

How do I know my message got delivered to the actor?

For now, this is not something you should worry about. The underlying `ActorSystem` of Akka.NET itself provides mechanisms to guarantee this, but `GuaranteedDeliveryActors` are an advanced topic.

For now, just trust that delivering messages is the `ActorSystem`s job, not yours. Trust, baby. :)

Okay, fine, I'll let the system deliver my messages. So how do I get an `IActorRef`?

There are two ways to get an `IActorRef`.

1) Create the actor

Actors form intrinsic supervision hierarchies (we cover in detail in lesson 5). This means there are "top level" actors, which essentially report directly to the `ActorSystem` itself, and there are "child" actors, which report to other actors.

To make an actor, you have to create it from its context. And **you've already done this!** Remember this?

```
// assume we have an existing actor system, "MyActorSystem"
IActorRef myFirstActor = MyActorSystem.ActorOf(Props.Create(() => new MyActorClass()), "myFi
```

As shown in the above example, you create an actor in the context of the actor that will supervise it (almost always). When you create the actor on the `ActorSystem` directly (as above), it is a top-level actor.

You make child actors the same way, except you create them from another actor, like so:

```
// have to create the child actor somewhere inside myFirstActor
// usually happens inside OnReceive or PreStart
class MyActorClass : UntypedActor{
    protected override void PreStart(){
        IActorRef myFirstChildActor = Context.ActorOf(Props.Create(() => new MyChildActorCla
    }
}
```

CAUTION: Do NOT call `new MyActorClass()` outside of `Props` and the `ActorSystem` to make an actor. We can't go into all the details here, but essentially, by doing so you're trying to create an actor outside the context of the `ActorSystem`. This will produce a completely unusable, undesirable object.

2) Look up the actor

All actors have an address (technically, an `ActorPath`) which represents where they are in the system hierarchy, and you can get a handle to them (get their `IActorRef`) by looking them up by their address.

We will cover this in much more detail in the next lesson.

Do I have to name my actors?

You may have noticed that we passed names into the `ActorSystem` when we were creating the above actors:

```
// last arg to the call to ActorOf() is a name
IActorRef myFirstActor = MyActorSystem.ActorOf(Props.Create(() => new MyActorClass()), "myFi
```

This name is not required. It is perfectly valid to create an actor without a name, like so:

```
// last arg to the call to ActorOf() is a name
IActorRef myFirstActor = MyActorSystem.ActorOf(Props.Create(() => new MyActorClass()))
```

That said, **the best practice is to name your actors**. Why? Because the name of your actor is used in log messages and in identifying actors. Get in the habit, and your future

self will thank you when you have to debug something and it has a nice label on it :)

Are there different types of `IActorRef`s?

Actually, yes. The most common, by far, is just a plain-old `IActorRef` or handle to an actor, as above.

However, there are also some other `IActorRef`s available to you within the context of an actor. As we said, all actors have a context. That context holds metadata, which includes information about the current message being processed. That information includes things like the `Parent` or `Children` of the current actor, as well as the `Sender` of the current message.

`Parent`, `Children`, and `Sender` all provide `IActorRef`s that you can use.

Props

What are `Props`?

Think of `Props` as a recipe for making an actor. Technically, `Props` is a configuration class that encapsulates all the information needed to make an instance of a given type of actor.

Why do we need `Props`?

`Props` objects are shareable recipes for creating an instance of an actor. `Props` get passed to the `ActorSystem` to generate an actor for your use.

Right now, `Props` probably feels like overkill. (If so, no worries.) But here's the deal.

The most basic `Props`, like we've seen, seem to only include the ingredients needed to make an actor—it's class and required args to its constructor.

BUT, what you haven't seen yet is that `Props` get extended to contain deployment information and other configuration details that are needed to do remote work. For example, `Props` are serializable, so they can be used to remotely create and deploy entire groups of actors on another machine somewhere on the network!

That's getting way ahead of ourselves though, but the short answer is that we need

`Props` to support a lot of the advanced features (clustering, remote actors, etc) that give Akka.NET the serious horsepower which makes it interesting.

How do I make `Props`?

Before we tell you how to make `Props`, let me tell you what NOT to do.

DO NOT TRY TO MAKE PROPS BY CALLING `new Props(...)`. Similar to trying to make an actor by calling `new MyActorClass()`, this is fighting the framework and not letting Akka's `ActorSystem` do its work under the hood to provide safe guarantees about actor restarts and lifecycle management.

There are 3 ways to properly create `Props`, and they all involve a call to `Props.Create()`.

1. **The `typeof` syntax:**

   ```
   Props props1 = Props.Create(typeof(MyActor));
   ```

   While it looks simple, **we recommend that you do not use this approach.** Why? *Because it has no type safety and can easily introduce bugs where everything compiles fine, and then blows up at runtime*.

2. **The lambda syntax**:

   ```
   Props props2 = Props.Create(() => new MyActor(..), "...");
   ```

   This is a mighty fine syntax, and our favorite. You can pass in the arguments required by the constructor of your actor class inline, along with a name.

3. **The generic syntax**:

   ```
   Props props3 = Props.Create<MyActor>();
   ```

   Another fine syntax that we whole-heartedly recommend.

How do I use `Props`?

You actually already know this, and have done it. You pass the `Props` —the actor recipe —to the call to `Context.ActorOf()` and the underlying `ActorSystem` reads the recipe, et voila! Whips you up a fresh new actor.

Enough of this conceptual business. Let's get to it!

Exercise

Before we can get into the meat of this lesson (`Props` and `IActorRef`s), we have to do a bit of cleanup.

Phase 1: Move validation into its own actor

We're going to move all our validation code into its own actor. It really doesn't belong in the `ConsoleReaderActor`. Validation deserves to have its own actor (similar to how you want single-purpose objects in OOP).

Create `ValidationActor` class

Make a new class called `ValidationActor` and put it into its own file. Fill it with all the validation logic that is currently in `ConsoleReaderActor`:

```
// ValidationActor.cs
using Akka.Actor;

namespace WinTail
{
    /// <summary>
    /// Actor that validates user input and signals result to others.
    /// </summary>
    public class ValidationActor : UntypedActor
    {
        private readonly IActorRef _consoleWriterActor;

        public ValidationActor(IActorRef consoleWriterActor)
        {
            _consoleWriterActor = consoleWriterActor;
        }

        protected override void OnReceive(object message)
        {
            var msg = message as string;
```

```
            if (string.IsNullOrEmpty(msg))
            {
                // signal that the user needs to supply an input
                _consoleWriterActor.Tell(new Messages.NullInputError("No input received."));
            }
            else
            {
                var valid = IsValid(msg);
                if (valid)
                {
                    // send success to console writer
                    _consoleWriterActor.Tell(new Messages.InputSuccess("Thank you! Message w
                }
                else
                {
                    // signal that input was bad
                    _consoleWriterActor.Tell(new Messages.ValidationError("Invalid: input ha
                }
            }

            // tell sender to continue doing its thing (whatever that may be, this actor doe
            Sender.Tell(new Messages.ContinueProcessing());

        }

        /// <summary>
        /// Determines if the message received is valid.
        /// Currently, arbitrarily checks if number of chars in message received is even.
        /// </summary>
        /// <param name="msg"></param>
        /// <returns></returns>
        private static bool IsValid(string msg)
        {
            var valid = msg.Length % 2 == 0;
            return valid;
        }
    }
}
```

Phase 2: Making `Props`, our actor recipes

Okay, now we can get to the good stuff! We are going to use what we've learned about `Props` and tweak the way we make our actors.

Again, we do not recommend using the `typeof` syntax. For practice, use both of the lambda and generic syntax!

> **Remember**: do NOT try to create `Props` by calling `new Props(...)`.

> When you do that, kittens die, unicorns vanish, Mordor wins and all manner of badness happens. Let's just not.

In this section, we're going to split out the `Props` objects onto their own lines for easier reading. In practice, we usually inline them into the call to `ActorOf`.

Delete existing `Props` and `IActorRef`s

In `Main()`, remove your existing actor declarations so we have a clean slate.

Your code should look like this right now:

```
// Program.cs
static void Main(string[] args)
{
    // initialize MyActorSystem
    MyActorSystem = ActorSystem.Create("MyActorSystem");

    // nothing here where our actors used to be!

    // tell console reader to begin
    consoleReaderActor.Tell(ConsoleReaderActor.StartCommand);

    // blocks the main thread from exiting until the actor system is shut down
    MyActorSystem.AwaitTermination();
}
```

Make `consoleWriterProps`

Go to `Program.cs`. Inside of `Main()`, split out `consoleWriterProps` onto its own line like so:

```
// Program.cs
Props consoleWriterProps = Props.Create(typeof (ConsoleWriterActor));
```

Here you can see we're using the typeof syntax, just to show you what it's like. But again, *we do not recommend using the `typeof` syntax in practice*.

Going forward, we'll only use the lambda and generic syntaxes for `Props`.

Make `validationActorProps`

Add this just to `Main()` also:

```
// Program.cs
Props validationActorProps = Props.Create(() => new ValidationActor(consoleWriterActor));
```

As you can see, here we're using the lambda syntax.

Make `consoleReaderProps`

Add this just to `Main()` also:

```
// Program.cs
Props consoleReaderProps = Props.Create<ConsoleReaderActor>(validationActor);
```

This is the generic syntax. `Props` accepts the actor class as a generic type argument, and then we pass in whatever the actor's constructor needs.

Phase 3: Making `IActorRef`s using various `Props`

Great! Now that we've got `Props` for all the actors we want, let's go make some actors!

Remember: do not try to make an actor by calling `new Actor()` outside of a `Props` object and/or outside the context of the `ActorSystem` or another `IActorRef`. Mordor and all that, remember?

Make a new `IActorRef` for `consoleWriterActor`

Add this to `Main()` on the line after `consoleWriterProps`:

```
// Program.cs
IActorRef consoleWriterActor = MyActorSystem.ActorOf(consoleWriterProps, "consoleWriterActor
```

Make a new `IActorRef` for `validationActor`

Add this to `Main()` on the line after `validationActorProps`:

```
// Program.cs
IActorRef validationActor = MyActorSystem.ActorOf(validationActorProps, "validationActor");
```

Make a new `IActorRef` for `consoleReaderActor`

Add this to `Main()` on the line after `consoleReaderProps`:

```
// Program.cs
IActorRef consoleReaderActor = MyActorSystem.ActorOf(consoleReaderProps, "consoleReaderActor
```

Calling out a special `IActorRef`: `Sender`

You may not have noticed it, but we actually are using a special `IActorRef` now: `Sender`. Go look for this in `ValidationActor.cs`:

```
// tell sender to continue doing its thing (whatever that may be, this actor doesn't care)
Sender.Tell(new Messages.ContinueProcessing());
```

This is the special `Sender` handle that is made available within an actors `Context` when it is processing a message. The `Context` always makes this reference available, along with some other metadata (more on that later).

Phase 4: A bit of cleanup

Just a bit of cleanup since we've changed our class structure. Then we can run our app again!

Update `ConsoleReaderActor`

Now that `ValidationActor` is doing our validation work, we should really slim down `ConsoleReaderActor`. Let's clean it up and have it just hand the message off to the

`ValidationActor` for validation.

We'll also need to store a reference to `ValidationActor` inside the `ConsoleReaderActor`, and we don't need a reference to the the `ConsoleWriterActor` anymore, so let's do some cleanup.

Modify your version of `ConsoleReaderActor` to match the below:

```
// ConsoleReaderActor.cs
// removing validation logic and changing store actor references
using System;
using Akka.Actor;

namespace WinTail
{
    /// <summary>
    /// Actor responsible for reading FROM the console.
    /// Also responsible for calling <see cref="ActorSystem.Shutdown"/>.
    /// </summary>
    class ConsoleReaderActor : UntypedActor
    {
        public const string StartCommand = "start";
        public const string ExitCommand = "exit";
        private readonly IActorRef _validationActor;

        public ConsoleReaderActor(IActorRef validationActor)
        {
            _validationActor = validationActor;
        }

        protected override void OnReceive(object message)
        {
            if (message.Equals(StartCommand))
            {
                DoPrintInstructions();
            }

            GetAndValidateInput();
        }


        #region Internal methods
        private void DoPrintInstructions()
        {
            Console.WriteLine("Write whatever you want into the console!");
            Console.WriteLine("Some entries will pass validation, and some won't...\n\n");
            Console.WriteLine("Type 'exit' to quit this application at any time.\n");
        }


        /// <summary>
```

```
        /// Reads input from console, validates it, then signals appropriate response
        /// (continue processing, error, success, etc.).
        /// </summary>
        private void GetAndValidateInput()
        {
            var message = Console.ReadLine();
            if (!string.IsNullOrEmpty(message) && String.Equals(message, ExitCommand, String
            {
                // if user typed ExitCommand, shut down the entire actor system (allows the
                Context.System.Shutdown();
                return;
            }

            // otherwise, just hand message off to validation actor (by telling its actor re
            _validationActor.Tell(message);
        }
        #endregion
    }
}
```

As you can see, we're now handing off the input from the console to the `ValidationActor` for validation and decisions. `ConsoleReaderActor` is now only responsible for reading from the console and handing the data off to another more sophisticated actor.

Fix that first `Props` call...

We can't very well recommend you not use the `typeof` syntax and then let it stay there. Real quick, go back to `Main()` and update `consoleWriterProps` to be use the generic syntax.

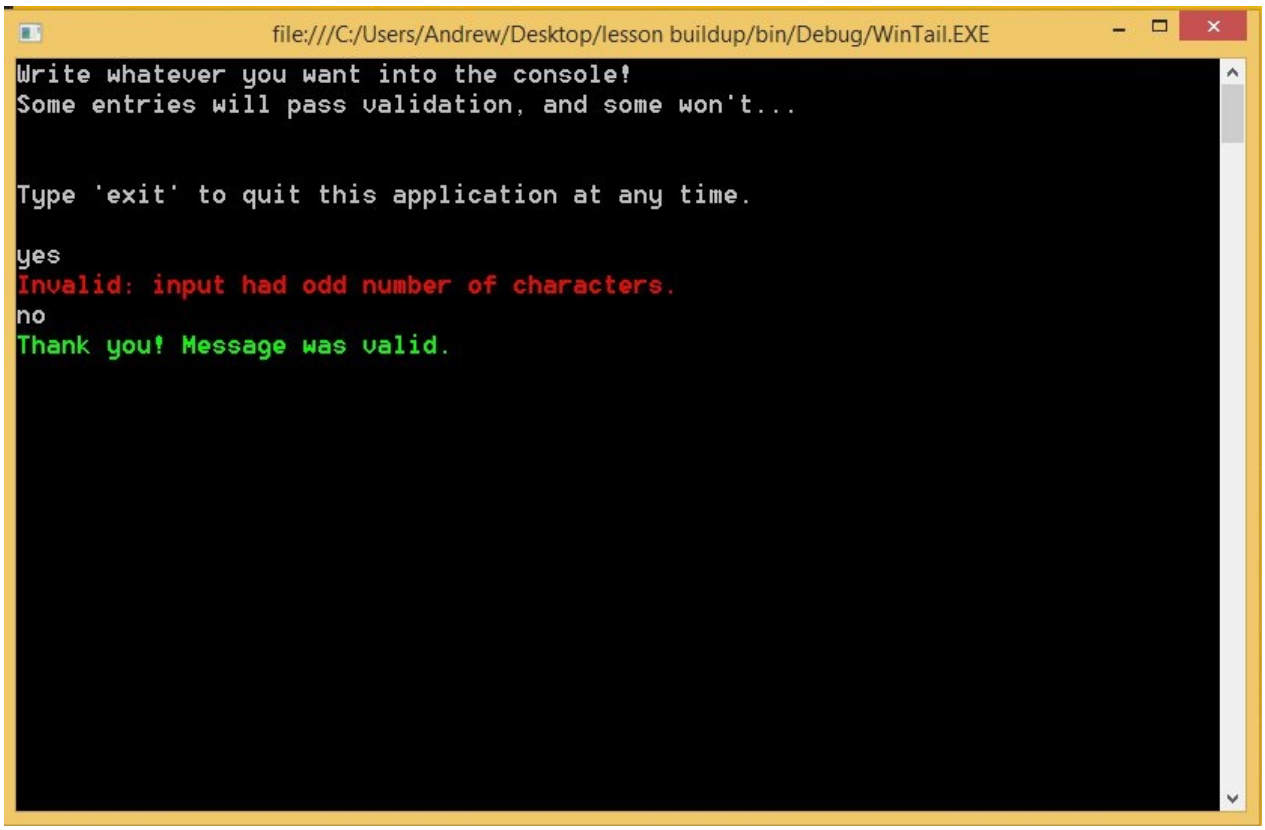
```
Props consoleWriterProps = Props.Create<ConsoleWriterActor>();
```

There. That's better.

Once you're done

Compare your code to the solution in the Completed folder to see what the instructors included in their samples.

If everything is working as it should, the output you see should be identical to last time:



Experience the danger of the `typeof` syntax for `Props` yourself

Since we harped on it earlier, let's illustrate the risk of using the `typeof Props` syntax and why we avoid it.

We've left a little landmine as a demonstration. You should blow it up just to see what happens.

1. Open up Completed/Program.cs.
2. Find the lines containing `fakeActorProps` and `fakeActor` (should be around line 18).
3. Uncomment these lines.
    - Look at what we're doing here—intentionally substituting a non-actor class into a `Props` object! Ridiculous! Terrible!
    - While this is an unlikely and frankly ridiculous example, that is exactly the point. It's just leaving open the door for mistakes, even with good intentions.
4. Build the solution. Watch with horror as this ridiculous piece of code *compiles without error!*
5. Run the solution.
6. Try to shield yourself from everything melting down when your program reaches that line of code.

Okay, so what was the point of that? Contrived as that example was, it should show you that *using the* `typeof` *syntax for* `Props` *has no type safety and is best avoided unless you have a damn good reason to use it.*

Great job! Onto Lesson 4!

Awesome work! Well done on completing your this lesson. It was a big one.

Let's move onto Lesson 4 - Child Actors, Actor Hierarchies, and Supervision.

Any questions?

Don't be afraid to ask questions :).

Come ask any questions you have, big or small, in this ongoing Bootcamp chat with the Petabridge & Akka.NET teams.

Problems with the code?

If there is a problem with the code running, or something else that needs to be fixed in this lesson, please create an issue and we'll get right on it. This will benefit everyone going through Bootcamp.

Lesson 1.4: Child Actors, Actor Hierarchies, and Supervision

This lesson will make a big jump forward in both the capabilities of our codebase, and in your understanding of how the actor model works.

This lesson is our most challenging one yet, so let's get right to it!

Key concepts / background

Before we get into the details of the actor hierarchy itself, let's stop and ask: why do we need a hierarchy at all?

There are two key reasons actors exist in a hierarchy:

1. To atomize work and turn massive amounts of data into manageable chunks
2. To contain errors and make the system resilient

Hierarchies atomize work

Having a hierarchy helps our system to break down work into smaller and smaller pieces, and to allow for different skill specializations at different levels of the hierarchy.

A common way this is realized in an actor systems is that large data streams get atomized, broken down over and over again until they are small and can easily be dealt with by a small code footprint.

Let's take Twitter as an example (users of JVM Akka). Using Akka, Twitter is able to break up their massive ingestion of data into small, manageable streams of information that they can react to. For instance - Twitter can break up their giant firehose of tweets into individual streams for the timeline of each user currently on the site, and they can use Akka to push messages that have arrived for that user into their stream via websocket / etc.

What's the pattern? Take a lot of work. Break it down recursively until it is easily dealt with. Respond as needed.

Hierarchies enable resilient systems

A hierarchy allows for different levels of risk and specialization to exist that could not otherwise.

Think of how an army works. An army has a general setting strategy and overseeing everything, but she is usually not going to be on the front line of the battle where there is the most risk. However, she has wide leverage and guides everything. At the same time, there are lower-ranking soldiers who are on the front lines, doing risky operations and carrying out the orders that they receive.

This is exactly how an actor system operates.

Higher-level actors are more supervisional in nature, and this allows the actor system to push risk down and to the edges. By pushing risky operations to the edges of the hierarchy, the system can isolate risk and recover from errors without the whole system crashing.

Both of these concepts are important, but for the rest of this lesson we'll put our emphasis on how actor systems use hierarchies to be resilient.

How is this achieved? **Supervision.**

What is supervision? Why should I care?

Supervision is the basic concept that allows your actor system to quickly isolate and recover from failures.

Every actor has another actor that supervises it, and helps it recover when errors occur. This is true from the top all the way to the bottom of the hierarchy.

This supervision ensures that when part of your application encounters an unexpected failure (unhandled exception, network timeout, etc.), that failure will be contained to only the affected part of your actor hierarchy.

All other actors will keep on working as though nothing happened. We call this "failure isolation" or "containment."

How is this accomplished? Let's find out…

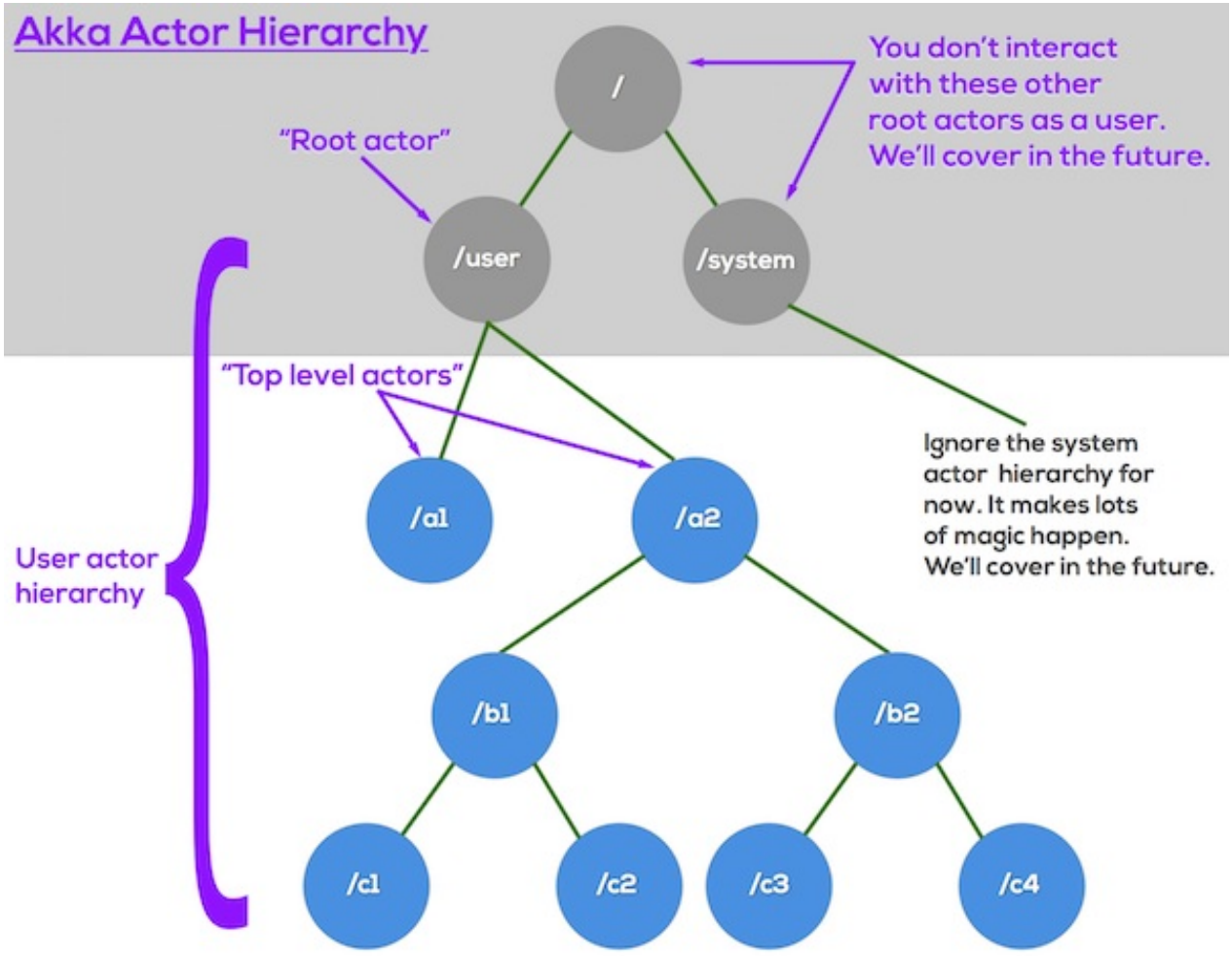
Actor Hierarchies

First, a key point: Every actor has a parent, and some actors have children. Parents supervise their children.

Since parents supervise their children, this means that ***every actor has a supervisor, and every actor can also BE a supervisor.***

Within your actor system, actors are arranged into a hierarchy. This means there are "top level" actors, which essentially report directly to the `ActorSystem` itself, and there are "child" actors, which report to other actors.

The overall hierarchy looks like this (we'll go through piece by piece in a moment):

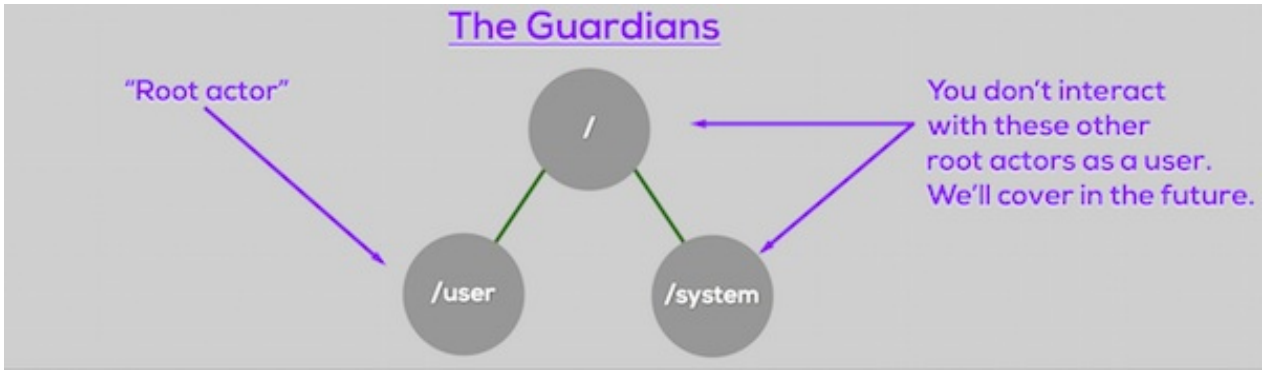


What are the levels of the hierarchy?

The base of it all: The "Guardians"

The "guardians" are the root actors of the entire system.

I'm referring to these three actors at the very top of the hierarchy:



The `/` actor

The `/` actor is the base actor of the entire actor system, and may also be referred to as "The Root Guardian." This actor supervises the `/system` and `/user` actors (the other "Guardians").

All actors require another actor as their parent, except this one. This actor is also sometimes called the "bubble-walker" since it is "out of the bubble" of the normal actor system. For now, don't worry about this actor.

The `/system` actor

The `/system` actor may also be referred to as "The System Guardian". The main job of this actor is to ensure that the system shuts down in an orderly manner, and to maintain/supervise other system actors which implement framework level features and utilities (logging, etc). We'll discuss the system guardian and the system actor hierarchy in a future post.

The `/user` actor

This is where the party starts! And this is where you'll be spending all your time as a developer.

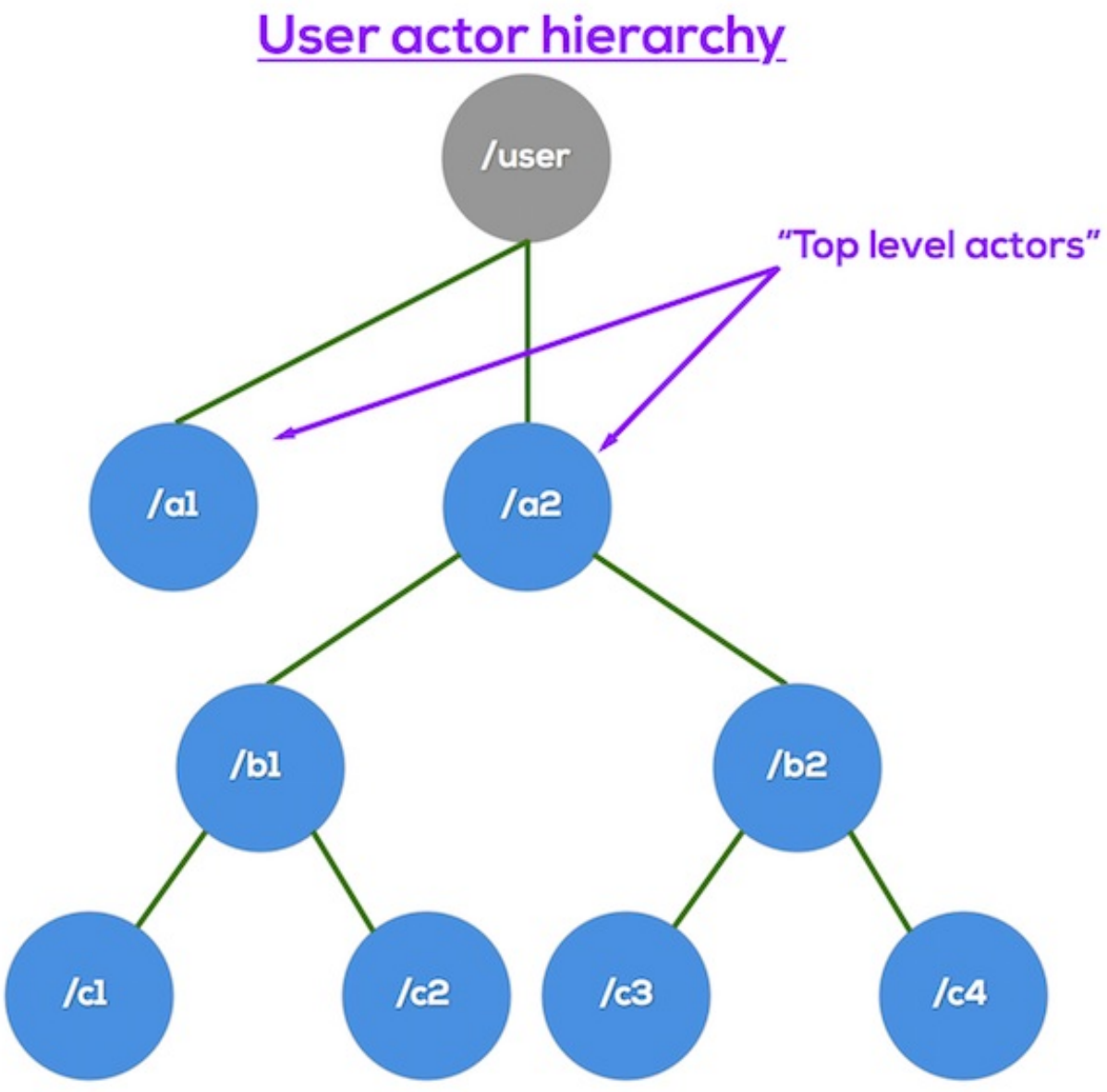
The `/user` actor may also be referred to as "The Guardian Actor". But from a user perspective, `/user` is the root of your actor system and is usually just called the "root actor."

> Generally, "root actor" refers to the `/user` actor.

As a user, you don't really need to worry too much about the Guardians. We just have to make sure that we use supervision properly under `/user` so that no exception can bubble up to the Guardians and crash the whole system.

The `/user` actor hierarchy

This is the meat and potatoes of the actor hierarchy: all of the actors you define in your applications.



> The direct children of the `/user` actor are called "top level actors."

Actors are always created as a child of some other actor.

Whenever you make an actor directly from the context of the actor system itself, that new actor is a top level actor, like so:

```
// create the top level actors from above diagram
IActorRef a1 = MyActorSystem.ActorOf(Props.Create<BasicActor>(), "a1");
IActorRef a2 = MyActorSystem.ActorOf(Props.Create<BasicActor>(), "a2");
```

Now, let's make child actors for `a2` by creating them inside the context of `a2`, our parent-to-be:

```
// create the children of actor a2
// this is inside actor a2
IActorRef b1 = Context.ActorOf(Props.Create<BasicActor>(), "b1");
IActorRef b2 = Context.ActorOf(Props.Create<BasicActor>(), "b2");
```

Actor path == actor position in hierarchy

Every actor has an address. To send a message from one actor to another, you just have to know it's address (AKA its "ActorPath"). This is what a full actor address looks like:

> *The "Path" portion of an actor address is just a description of where that actor is in your actor hierarchy. Each level of the hierarchy is separated by a single slash ('/').*

For example, if we were running on `localhost`, the full address of actor `b2` would be `akka.tcp://MyActorSystem@localhost:9001/user/a2/b2`.

One question that comes up a lot is, "Do my actor classes have to live at a certain point in the hierarchy?" For example, if I have an actor class, `FooActor`—can I only deploy that actor as a child of `BarActor` on the hierarchy? Or can I deploy it anywhere?

The answer is **any actor may be placed anywhere in your actor hierarchy**.

> *Any actor may be placed anywhere in your actor hierarchy.*

Okay, now that we've got this hierarchy business down, let's do something interesting with it. Like supervising!

How supervision works in the actor hierarchy

Now that you know how actors are organized, know this: actors supervise their children. *But, they only supervise the level that is immediately below them in the hierarchy (actors do not supervise their grandchildren, great-grandchildren, etc).*

> Actors only supervise their children, the level immediately below them in the hierarchy.

When does supervision come into play? Errors!

When things go wrong, that's when! Whenever a child actor has an unhandled exception and is crashing, it reaches out to its parent for help and to tell it what to do.

Specifically, the child will send its parent a message that is of the `Failure` class. Then it's up to the parent to decide what to do.

How can the parent resolve the error?

There are two factors that determine how a failure is resolved:

1. How the child failed (what type of `Exception` did the child include in its `Failure` message to its parent.)
2. What Directive the parent actor executes in response to a child `Failure`. This is determined by the parent's `SupervisionStrategy`.

Here's the sequence of events when an error occurs:

1. Unhandled exception occurs in child actor (`c1`), which is supervised by its parent (`b1`).
2. `c1` suspends operations.
3. The system sends a `Failure` message from `c1` to `b1`, with the `Exception` that was raised.
4. `b1` issues a directive to `c1` telling it what to do.

5. Life goes on, and the affected part of the system heals itself without burning down the whole house. Kittens and unicorns, handing out free ice cream and coffee to be enjoyed while relaxing on a pillowy rainbow. Yay!

Supervision directives

When it receives an error from its child, a parent can take one of the following actions ("directives"). The supervision strategy maps different exception types to these directives, allowing you to handle different types of errors as appropriate.

Types of supervision directives (i.e. what decisions a supervisor can make):

- **Restart** the child (default): this is the common case, and the default.
- **Stop** the child: this permanently terminates the child actor.
- **Escalate** the error (and stop itself): this is the parent saying "I don't know what to do! I'm gonna stop everything and ask MY parent!"
- **Resume** processing (ignores the error): you generally won't use this. Ignore it for now.

> *The critical thing to know here is that **whatever action is taken on a parent propagates to its children**. If a parent is halted, all its children halt. If it is restarted, all its children restart.*

Supervision strategies

There are two built-in supervision strategies:

1. One-For-One Strategy (default)
2. All-For-One Strategy

   The basic difference between these is how widespread the effects of the error-resolution directive will be.

One-For-One says that that the directive issued by the parent only applies to the failing child actor. It has no effect on the siblings of the failing child. This is the default strategy if you don't specify one. (You can also define your own custom supervision strategy.)

All-For-One says that that the directive issued by the parent applies to the failing child actor AND all of its siblings.

The other important choice you make in a supervision strategy is how many times a child can fail within a given period of time before it is shut down (e.g. "no more than 10 errors within 60 seconds, or you're shut down").

Here's an example supervision strategy:

```
public class MyActor : UntypedActor
{
    // if any child of MyActor throws an exception, apply the rules below
    // e.g. Restart the child, if 10 exceptions occur in 30 seconds or
    // less, then stop the actor
    protected override SupervisorStrategy SupervisorStrategy()
    {
        return new OneForOneStrategy(// or AllForOneStrategy
            maxNrOfRetries: 10,
            withinTimeRange: TimeSpan.FromSeconds(30),
            localOnlyDecider: x =>
            {
                // Maybe ArithmeticException is not application critical
                // so we just ignore the error and keep going.
                if (x is ArithmeticException) return Directive.Resume;

                // Error that we have no idea what to do with
                else if (x is InsanelyBadException) return Directive.Escalate;

                // Error that we can't recover from, stop the failing child
                else if (x is NotSupportedException) return Directive.Stop;

                // otherwise restart the failing child
                else return Directive.Restart;
            });
    }

    ...
}
```

What's the point? Containment.

The whole point of supervision strategies and directives is to contain failure within the system and self-heal, so the whole system doesn't crash. How do we do this?

We push potentially-dangerous operations from a parent to a child, whose only job is to carry out the dangerous task.

For example, let's say we're running a stats system during the World Cup, that keeps scores and player statistics from a bunch of games in the World Cup.

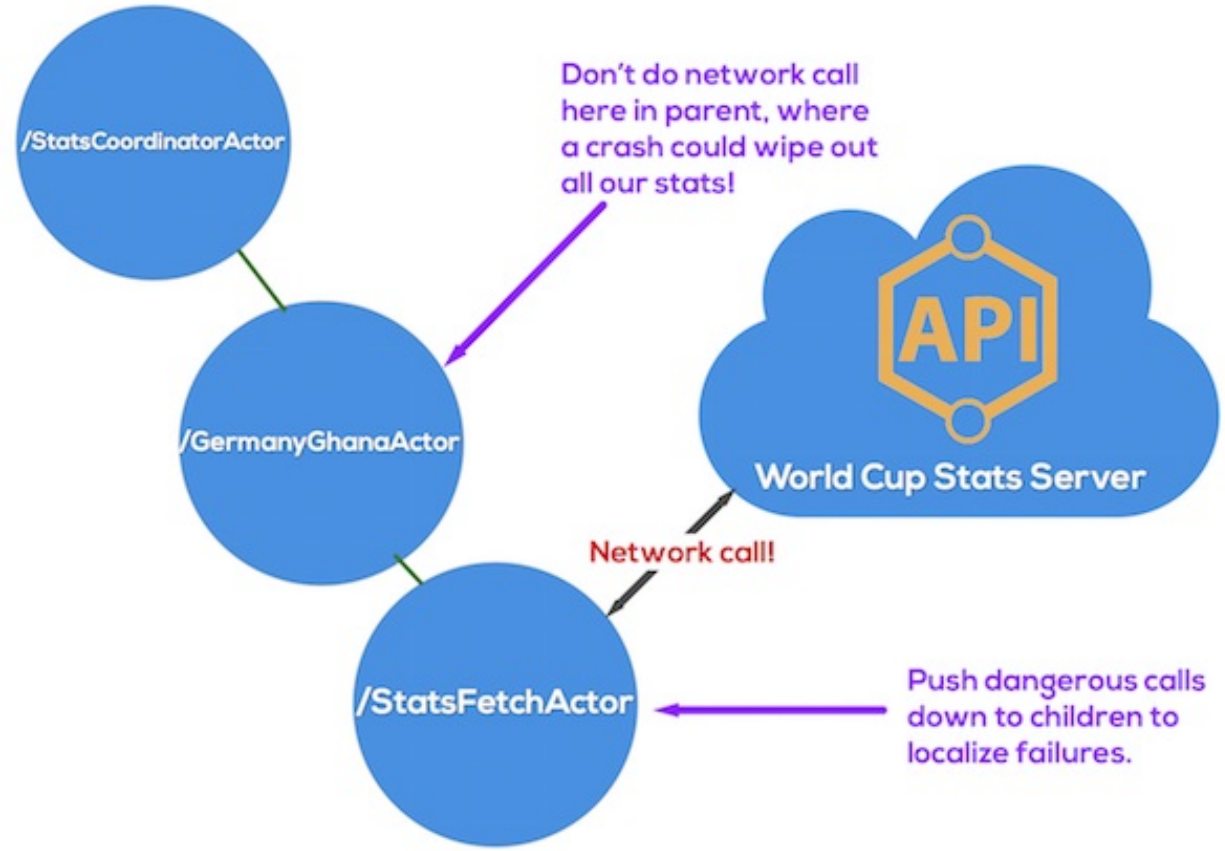
Now, being the World Cup, there could be huge demand on that API and it could get throttled, start rate-limiting, or just plain crash (no offense FIFA, I love you guys and the Cup). We'll use the epic Germany-Ghana match as an example.

But our scorekeeper has to periodically update its data as the game progresses. Let's assume it has to call to an external API maintained by FIFA to get the data it needs.

This network call is dangerous! If the request raises an error, it will crash the actor that started the call. So how do we protect ourselves?

We keep the stats in a parent actor, and push that nasty network call down into a child actor. That way, if the child crashes, it doesn't affect the parent, which is holding on to all the important data. By doing this, we are **localizing the failure** and keeping it from spreading throughout the system.

Here's an example of how we could structure the actor hierarchy to safely accomplish the goal:



Recall that we could have many clones of this exact structure working in parallel, with one clone per game we are tracking. **And we wouldn't have to write any new code to scale it out!** Beautiful.

> You may also hear people use the term "error kernel," which refers to how much of the system is affected by the failure. You may also hear "error kernel pattern," which is just fancy shorthand for the approach I just explained where we push dangerous behavior to child actors to isolate/protect the parent.

Exercise

To start off, we need to do some upgrading of our system. We are going to add in the components which will enable our actor system to actually monitor a file for changes. We have most of the classes we need, but there are a few pieces of utility code that we need to add.

We're almost done! We really just need to add the `TailCoordinatorActor`, `TailActor`, and the `FileObserver`.

The goal of this exercise is to show you how to make a parent/child actor relationship.

Phase 1: A quick bit of prep

Replace `ValidationActor` with `FileValidatorActor`

Since we're shifting to actually looking at files now, go ahead and replace `ValidationActor` with `FileValidatorActor`.

Add a new class, `FileValidatorActor`, with this code:

```
// FileValidatorActor.cs
using System.IO;
using Akka.Actor;

namespace WinTail
{
    /// <summary>
    /// Actor that validates user input and signals result to others.
    /// </summary>
    public class FileValidatorActor : UntypedActor
    {
        private readonly IActorRef _consoleWriterActor;
        private readonly IActorRef _tailCoordinatorActor;

        public FileValidatorActor(IActorRef consoleWriterActor, IActorRef tailCoordinatorAct
        {
```

```
            _consoleWriterActor = consoleWriterActor;
            _tailCoordinatorActor = tailCoordinatorActor;
        }

        protected override void OnReceive(object message)
        {
            var msg = message as string;
            if (string.IsNullOrEmpty(msg))
            {
                // signal that the user needs to supply an input
                _consoleWriterActor.Tell(new Messages.NullInputError("Input was blank. Pleas

                // tell sender to continue doing its thing (whatever that may be, this actor
                Sender.Tell(new Messages.ContinueProcessing());
            }
            else
            {
                var valid = IsFileUri(msg);
                if (valid)
                {
                    // signal successful input
                    _consoleWriterActor.Tell(new Messages.InputSuccess(string.Format("Starti

                    // start coordinator
                    _tailCoordinatorActor.Tell(new TailCoordinatorActor.StartTail(msg, _cons
                }
                else
                {
                    // signal that input was bad
                    _consoleWriterActor.Tell(new Messages.ValidationError(string.Format("{0}

                    // tell sender to continue doing its thing (whatever that may be, this a
                    Sender.Tell(new Messages.ContinueProcessing());
                }
            }



        }

        /// <summary>
        /// Checks if file exists at path provided by user.
        /// </summary>
        /// <param name="path"></param>
        /// <returns></returns>
        private static bool IsFileUri(string path)
        {
            return File.Exists(path);
        }
    }
}
```

You'll also want to make sure to update the `Props` instance in `Main` that references the

class:

```
// Program.cs
Props validationActorProps = Props.Create(() => new FileValidatorActor(consoleWriterActor));
```

Update `DoPrintInstructions`

Just making a slight tweak to our instructions here, since we'll be using a text file on disk going forward instead of prompting the user for input.

Update `DoPrintInstructions()` to this:

```
// ConsoleReaderActor.cs
private void DoPrintInstructions()
{
    Console.WriteLine("Please provide the URI of a log file on disk.\n");
}
```

Add `FileObserver`

This is a utility class that we're providing for you to use. It does the low-level work of actually watching a file for changes.

Create a new class called `FileObserver` and type in the code for FileObserver.cs. If you're running this on Mono, note the extra environment variable that has to be uncommented in the `Start()` method:

```
// FileObserver.cs
using System;
using System.IO;
using Akka.Actor;

namespace WinTail
{
    /// <summary>
    /// Turns <see cref="FileSystemWatcher"/> events about a specific file into messages for
    /// </summary>
    public class FileObserver : IDisposable
    {
        private readonly IActorRef _tailActor;
```

```
        private readonly string _absoluteFilePath;
        private FileSystemWatcher _watcher;
        private readonly string _fileDir;
        private readonly string _fileNameOnly;

        public FileObserver(IActorRef tailActor, string absoluteFilePath)
        {
            _tailActor = tailActor;
            _absoluteFilePath = absoluteFilePath;
            _fileDir = Path.GetDirectoryName(absoluteFilePath);
            _fileNameOnly = Path.GetFileName(absoluteFilePath);
        }

        /// <summary>
        /// Begin monitoring file.
        /// </summary>
        public void Start()
        {
            // Need this for Mono 3.12.0 workaround
            // Environment.SetEnvironmentVariable("MONO_MANAGED_WATCHER", "enabled"); // unc

            // make watcher to observe our specific file
            _watcher = new FileSystemWatcher(_fileDir, _fileNameOnly);

            // watch our file for changes to the file name, or new messages being written to
            _watcher.NotifyFilter = NotifyFilters.FileName | NotifyFilters.LastWrite;

            // assign callbacks for event types
            _watcher.Changed += OnFileChanged;
            _watcher.Error += OnFileError;

            // start watching
            _watcher.EnableRaisingEvents = true;

        }

        /// <summary>
        /// Stop monitoring file.
        /// </summary>
        public void Dispose()
        {
            _watcher.Dispose();
        }

        /// <summary>
        /// Callback for <see cref="FileSystemWatcher"/> file error events.
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        void OnFileError(object sender, ErrorEventArgs e)
        {
            _tailActor.Tell(new TailActor.FileError(_fileNameOnly, e.GetException().Message)
        }

        /// <summary>
        /// Callback for <see cref="FileSystemWatcher"/> file change events.
```

```
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        void OnFileChanged(object sender, FileSystemEventArgs e)
        {
            if (e.ChangeType == WatcherChangeTypes.Changed)
            {
                // here we use a special ActorRefs.NoSender
                // since this event can happen many times, this is a little microoptimizatio
                _tailActor.Tell(new TailActor.FileWrite(e.Name), ActorRefs.NoSender);
            }

        }

    }
}
```

Phase 2: Make your first parent/child actors!

Great! Now we're ready to create our actor classes that will form a parent/child relationship.

Recall that in the hierarchy we're going for, there is a `TailCoordinatorActor` that coordinates child actors to actually monitor and tail files. For now it will only supervise one child, `TailActor`, but in the future it can easily expand to have many children, each observing/tailing a different file.

Add `TailCoordinatorActor`

Create a new class called `TailCoordinatorActor` in a file of the same name.

Add the following code, which defines our coordinator actor (which will soon be our first parent actor).

```
// TailCoordinatorActor.cs
using System;
using Akka.Actor;

namespace WinTail
{
    public class TailCoordinatorActor : UntypedActor
    {
        #region Message types
        /// <summary>
```

```
        /// Start tailing the file at user-specified path.
        /// </summary>
        public class StartTail
        {
            public StartTail(string filePath, IActorRef reporterActor)
            {
                FilePath = filePath;
                ReporterActor = reporterActor;
            }

            public string FilePath { get; private set; }

            public IActorRef ReporterActor { get; private set; }
        }

        /// <summary>
        /// Stop tailing the file at user-specified path.
        /// </summary>
        public class StopTail
        {
            public StopTail(string filePath)
            {
                FilePath = filePath;
            }

            public string FilePath { get; private set; }
        }

        #endregion

        protected override void OnReceive(object message)
        {
            if (message is StartTail)
            {
                var msg = message as StartTail;
                // YOU NEED TO FILL IN HERE
            }

        }
    }
}
```

Create `IActorRef` for `TailCoordinatorActor`

In `Main()`, create a new `IActorRef` for `TailCoordinatorActor` and then pass it into `fileValidatorActorProps`, like so:

```
// Program.Main
// make tailCoordinatorActor
Props tailCoordinatorProps = Props.Create(() => new TailCoordinatorActor());
IActorRef tailCoordinatorActor = MyActorSystem.ActorOf(tailCoordinatorProps, "tailCoordinato
```

```
// pass tailCoordinatorActor to fileValidatorActorProps (just adding one extra arg)
Props fileValidatorActorProps = Props.Create(() => new FileValidatorActor(consoleWriterActor
IActorRef validationActor = MyActorSystem.ActorOf(fileValidatorActorProps, "validationActor"
```

Add `TailActor`

Now, add a class called `TailActor` in its own file. This actor is the actor that is actually responsible for tailing a given file. `TailActor` will be created and supervised by `TailCoordinatorActor` in a moment.

For now, add the following code in `TailActor.cs`:

```
// TailActor.cs
using System.IO;
using System.Text;
using Akka.Actor;

namespace WinTail
{
    /// <summary>
    /// Monitors the file at <see cref="_filePath"/> for changes and sends file updates to c
    /// </summary>
    public class TailActor : UntypedActor
    {
        #region Message types

        /// <summary>
        /// Signal that the file has changed, and we need to read the next line of the file.
        /// </summary>
        public class FileWrite
        {
            public FileWrite(string fileName)
            {
                FileName = fileName;
            }

            public string FileName { get; private set; }
        }

        /// <summary>
        /// Signal that the OS had an error accessing the file.
        /// </summary>
        public class FileError
        {
            public FileError(string fileName, string reason)
            {
                FileName = fileName;
```

```
            Reason = reason;
        }

        public string FileName { get; private set; }

        public string Reason { get; private set; }
    }

    /// <summary>
    /// Signal to read the initial contents of the file at actor startup.
    /// </summary>
    public class InitialRead
    {
        public InitialRead(string fileName, string text)
        {
            FileName = fileName;
            Text = text;
        }

        public string FileName { get; private set; }
        public string Text { get; private set; }
    }

    #endregion

    private readonly string _filePath;
    private readonly IActorRef _reporterActor;
    private readonly FileObserver _observer;
    private readonly Stream _fileStream;
    private readonly StreamReader _fileStreamReader;

    public TailActor(IActorRef reporterActor, string filePath)
    {
        _reporterActor = reporterActor;
        _filePath = filePath;

        // start watching file for changes
        _observer = new FileObserver(Self, Path.GetFullPath(_filePath));
        _observer.Start();

        // open the file stream with shared read/write permissions (so file can be writt
        _fileStream = new FileStream(Path.GetFullPath(_filePath), FileMode.Open, FileAcc
            FileShare.ReadWrite);
        _fileStreamReader = new StreamReader(_fileStream, Encoding.UTF8);

        // read the initial contents of the file and send it to console as first message
        var text = _fileStreamReader.ReadToEnd();
        Self.Tell(new InitialRead(_filePath, text));
    }

    protected override void OnReceive(object message)
    {
        if (message is FileWrite)
        {
            // move file cursor forward
            // pull results from cursor to end of file and write to output
```

```
                // (this is assuming a log file type format that is append-only)
                var text = _fileStreamReader.ReadToEnd();
                if (!string.IsNullOrEmpty(text))
                {
                    _reporterActor.Tell(text);
                }

            }
            else if (message is FileError)
            {
                var fe = message as FileError;
                _reporterActor.Tell(string.Format("Tail error: {0}", fe.Reason));
            }
            else if (message is InitialRead)
            {
                var ir = message as InitialRead;
                _reporterActor.Tell(ir.Text);
            }
        }
    }
}
```

Add `TailActor` as a child of `TailCoordinatorActor`

Quick review: `TailActor` is to be a child of `TailCoordinatorActor` and will therefore be supervised by `TailCoordinatorActor`.

This also means that `TailActor` must be created in the context of `TailCoordinatorActor`.

Go to `TailCoordinatorActor.cs` and replace `OnReceive()` with the following code to create your first child actor!

```
// TailCoordinatorActor.OnReceive
protected override void OnReceive(object message)
{
    if (message is StartTail)
    {
        var msg = message as StartTail;
        // here we are creating our first parent/child relationship!
        // the TailActor instance created here is a child
        // of this instance of TailCoordinatorActor
        Context.ActorOf(Props.Create(() => new TailActor(msg.ReporterActor, msg.FilePath)));
    }

}
```

BAM!

You have just established your first parent/child actor relationship!

Phase 3: Implement a `SupervisorStrategy`

Now it's time to add a supervision strategy to your new parent, `TailCoordinatorActor`.

The default `SupervisorStrategy` is a One-For-One strategy (docs) w/ a Restart directive (docs).

Add this code to the bottom of `TailCoordinatorActor`:

```
// TailCoordinatorActor.cs
protected override SupervisorStrategy SupervisorStrategy()
{
    return new OneForOneStrategy (
        10, // maxNumberOfRetries
        TimeSpan.FromSeconds(30), // withinTimeRange
        x => // localOnlyDecider
        {
            //Maybe we consider ArithmeticException to not be application critical
            //so we just ignore the error and keep going.
            if (x is ArithmeticException) return Directive.Resume;

            //Error that we cannot recover from, stop the failing actor
            else if (x is NotSupportedException) return Directive.Stop;

            //In all other cases, just restart the failing actor
            else return Directive.Restart;
        });
}
```

Phase 4: Build and Run!

Awesome! It's time to fire this baby up and see it in action.

Get a text file you can tail

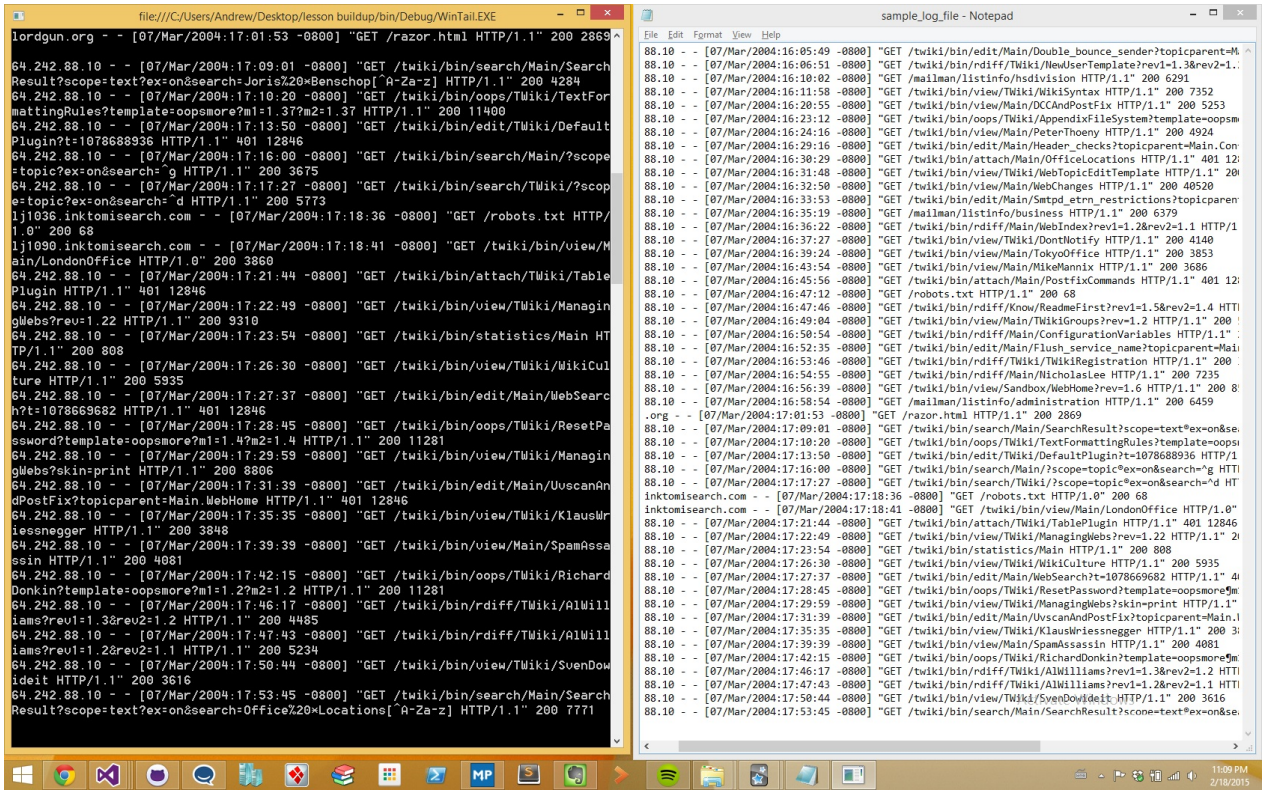
We recommend a log file like this sample one, but you can also just make a plain text file and fill it with whatever you want.

Open the text file up and put it on one side of your screen.

Fire it up

Check the starting output

Run the application and you should see a console window open up and print out the starting contents of your log file. The starting state should look like this if you're using the sample log file we provided:

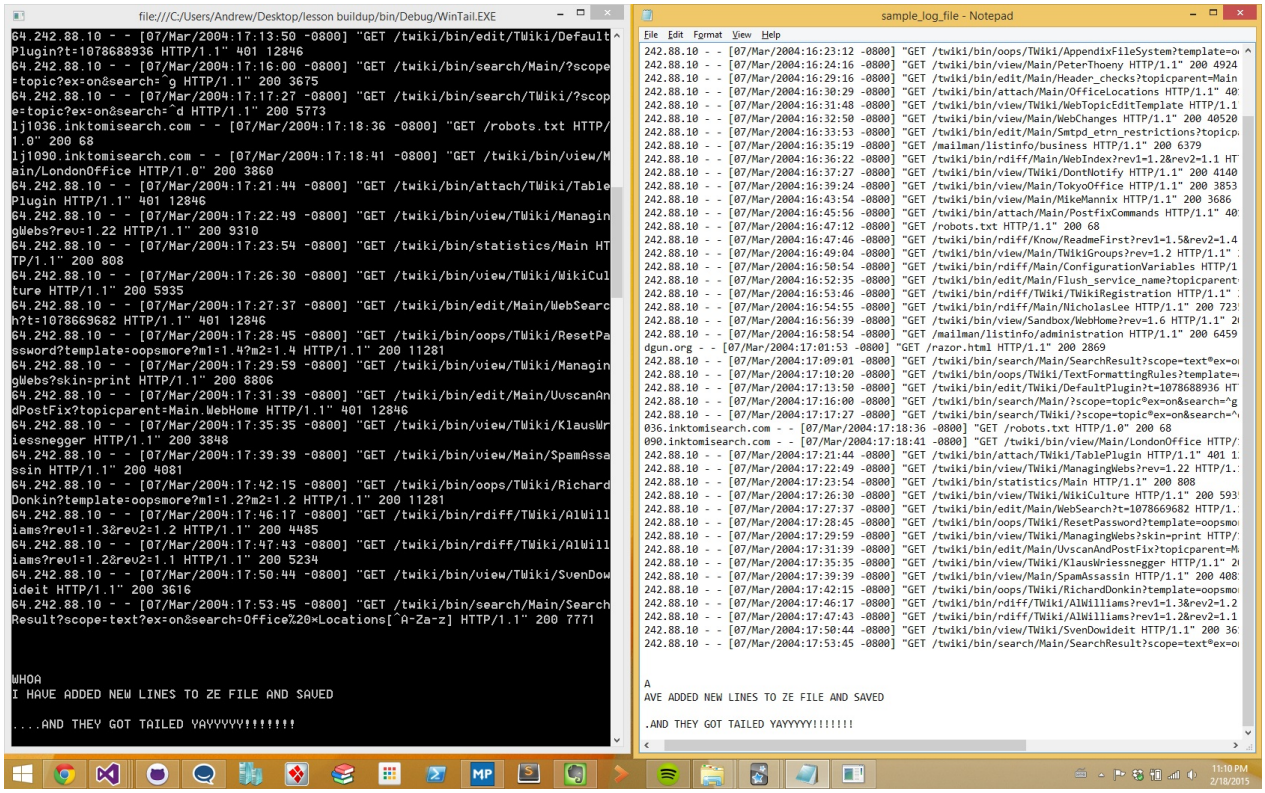


Leave both the console and the file open, and then...

Add text and see if the `tail` works!

Add some lines of text to the text file, save it, and watch it show up in the `tail`!

It should look something like this:



Congrats! YOU HAVE JUST MADE A PORT OF `tail` IN .NET!

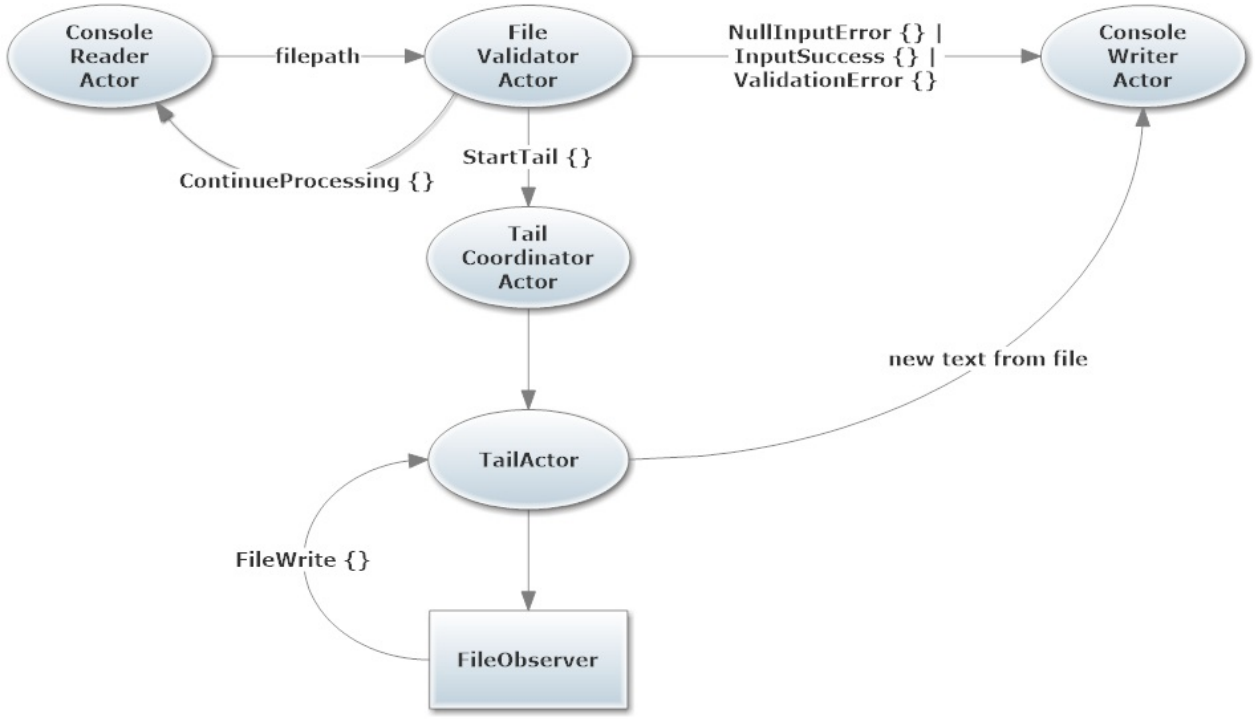
Once you're done

Compare your code to the solution in the Completed folder to see what the instructors included in their samples.

Great job! Onto Lesson 5!

Awesome work! Well done on completing this lesson, we know it was a bear! It was a big jump forward for our system and in your understanding.

Here is a high-level overview of our working system!



Let's move onto Lesson 5 - Looking up Actors by Address with `ActorSelection`.

Supervision FAQ

How long do child actors have to wait for their supervisor?

This is a common question we get: What if there are a bunch of messages already in the supervisor's mailbox waiting to be processed when a child reports an error? Won't the crashing child actor have to wait until those are processed until it gets a response?

Actually, no. When an actor reports an error to its supervisor, it is sent as a special type of "system message." *System messages jump to the front of the supervisor's mailbox and are processed before the supervisor returns to its normal processing.*

> *System messages jump to the front of the supervisor's mailbox and are processed before the supervisor returns to its normal processing.*

Parents come with a default SupervisorStrategy object (or you can provide a custom one) that makes decisions on how to handle failures with their child actors.

But what happens to the current message when an actor fails?

The current message being processed by an actor when it is halted (regardless of whether the failure happened to it or its parent) can be saved and re-processed after restarting. There are several ways to do this. The most common approach used is during `preRestart()`, the actor can stash the message (if it has a stash) or it can send the message to another actor that will send it back once restarted. (Note: If the actor has a stash, it will automatically unstash the message once it successfully restarts.)

Any questions?

Don't be afraid to ask questions :).

Come ask any questions you have, big or small, in this ongoing Bootcamp chat with the Petabridge & Akka.NET teams.

Problems with the code?

If there is a problem with the code running, or something else that needs to be fixed in this lesson, please create an issue and we'll get right on it. This will benefit everyone going through Bootcamp.

Lesson 1.5: Looking up Actors by Address with `ActorSelection`

Welcome to lesson 5! Wow, we've come a long way together. First, just **take a quick moment to appreciate that, and give yourself credit for investing your time and energy into your craft**.

Mmm, that was nice.

Okay, let's get on with it!

In this lesson, we're going to learn how to decouple our actors from each other a bit and a new way of communicating between actors: `ActorSelection`. This lesson is shorter than the previous ones, now that we've laid down a solid conceptual foundation.

Key concepts / background

`ActorSelection` is a natural extension of actor hierarchies, which we covered in the last lesson. Now that we understand that actors live in hierarchies, it begs the question: now that actors aren't all on the same level, does this change the way they communicate?

We know that we need a handle to an actor in order to send it a message and get it to do work. But now we have actors all over the place in this hierarchy, and don't always have a direct link (`IActorRef`) to the actor(s) we want to send messages to.

So how do we send a message to an actor somewhere else in the hierarchy, that we don't have a stored `IActorRef` for? What then?

Enter `ActorSelection`.

What is `ActorSelection`?

`ActorSelection` is nothing more than using an `ActorPath` to get a handle to an actor or actors so you can send them a message, without having to store their actual `IActorRef`s.

Instead of getting a handle to an actor by creating or passing around its `IActorRef`, you're

"looking up" a handle to the actor by its `ActorPath` (recall that the `ActorPath` is the address for an actor's position in the system hierarchy). It's kind of like looking someone up on Skype by their email when you don't already have their username.

However, be aware that while `ActorSelection` is how you look up an `IActorRef`, it's not inherently a 1-1 lookup to a single actor.

Technically, the `ActorSelection` object you get when you do a lookup does not point to a specific `IActorRef`. It's actually a handle that internally points to every `IActorRef` that matches the expression you looked up. Wildcards are supported in this expression, so it's an expression that selects 0+ actors. (More on this later.)

An `ActorSelection` will also match two different `IActorRef`s with the same name if the first one dies and is replaced by another (not restarted, in which case it would be the same `IActorRef`).

Is it an object? A process? Both?

We think of `ActorSelection` as both a process and an object: the process of looking actor(s) up by `ActorPath`, and the object returned from that lookup, which allows us to send messages to the actor(s) matched by the expression we looked up.

Why should I care about `ActorSelection`?

Always a great question, glad you asked! There are a number of benefits that `ActorSelection` gives you.

Location transparency

What location transparency actually means is that whenever you send a message to an actor, you don't need to know where they are within an actor system, which might span hundreds of computers. You don't care if your actors are all in one process or spread across 100 machines around the world. You just have to know that actors' address (its `ActorPath`).

Think of it like calling someone's cell phone number - you don't need to know that your friend Bob is in Seattle, Washington, USA in order to place a call to them. You just need to dial Bob's cell phone number and your cellular network provider will take care of the rest.

The location transparency (enabled by `ActorSelection`) is essential for creating scalable systems that can handle high-availability requirements. We'll go into this more in Units 2 & 3.

Loose coupling

Since you don't have to constantly be holding on to `IActorRef`s to store and pass around, your actors don't get tightly coupled to each other. Just like in object-oriented programming, this is a Very Good Thing. It means the components of your system stay loose and easily adaptable / reusable. It lowers the cost of maintaining your codebase.

Dynamic behavior

Dynamic behavior is an advanced concept that we dive into in the beginning of Unit 2, but for now just be aware that the behavior of a given actor can be very flexible. This lets actors easily represent things like Finite State Machines so a small code footprint can easily handle complex situations.

Where does `ActorSelection` come into play on this? Well, if you want to have a very dynamic and adaptable system, there are probably going to be lots of actors coming and going from the hierarchy and trying to store / pass around handles to all of them would be a real pain. `ActorSelection` lets you easily just send messages to well known addresses of the key actor(s) you need to communicate with, and not worry about getting/passing/storing handles to the things you need.

You also can build extremely dynamic actors where not even the `ActorPath` needed to do an `ActorSelection` is hardcoded, but can instead be represented by a message that is passed into your actor.

Flexible communication patterns == adaptable system

Let's run w/ this idea of adaptability, because it's important for your happiness as a developer, the resilience of your system, and the speed at which your organization can move.

Since you don't have to couple everything together to make it work, this will speed up your development cycles. You can introduce new actors and entirely new sections into the actor hierarchy without having to go back and change everything you've already written. Your system has a much more flexible communication structure that can expand

and accommodate new actors (and requirements) easily.

In a nutshell: `ActorSelection` makes your system much more adaptable to change and also enables it to be more powerful.

Okay, got it. So when do I actually use `ActorSelection`?

Talking to top-level actors

The most common case we're aware of for using `ActorSelection` is to send messages to top-level actors with well known names.

For example, let's imagine you have a top level actor that handles all authentication for your system. Other actors can send a message to that actor to find out if a user is authenticated or has permission to carry out a certain operation. Let's call this actor `AuthenticationActor`.

Since this is a top-level actor, we now know thanks to our knowledge of hierarchies that its `ActorPath` is going to be `/user/AuthenticationActor`. Using this well-known address, **ANY** actor in the entire system can easily talk to it without needing to previously have its `IActorRef`, as in this example:

```
// send username to AuthenticationActor for authentication
// this is an "absolute" actor selection, since it starts at top-level /user/
Context.ActorSelection("akka://MyActorSystem/user/AuthenticationActor").Tell(username);
```

> NOTE: `ActorSelection`s can be either absolute or relative. An absolute `ActorSelection` includes the root `/user/` actor in the path. However, an `ActorSelection` could also be relative, such as `Context.ActorSelection("../validationActor")`.

Doing elastic processing of large data streams

A very common extension of the well-known actor pattern is talking to routers (docs). Routers are a more advanced topic we go in-depth on in unit 2.

As a quick example, imagine that you had a system that needed to process a large data stream in real time. Let's say for fun that you had a very popular consumer app that had peak usage once or twice a day, like a social app that people love to use on their breaks at work. Each user is generating a flurry of activity that has to be processed and rolled up in realtime. For the sake of this example, imagine that you had an actor running per user tracking the user's activity, and feeding that data to other parts of the system.

(*Remember: actors are cheap! It's totally reasonable from an overhead POV to create actors for every user. Last we checked, Akka.NET could run between 2.5-3 million actors per gig of RAM.*)

There's a lot of data coming in, and you want to make sure the system stays responsive under high loads. What do you do? One good option is to create a router (or job coordinator, as some call it) and have that coordinator manage a pool of workers that do the processing. This pool of workers can then elastically expand/contract to meet the changing processing demands of the system on the fly.

As all these per-user actors are being created and shutting down when users leave the app, how do you consistently feed the data to the right place? You send the data to the `ActorSelection`s for all the coordinators you need to hand off processing to.

When not just replying

A very commonly used `IActorRef` is `Sender`, which is the `IActorRef` made available to every actor context and is a handle to the sender of the message currently being processed. In an earlier lesson, we used this inside `FileValidatorActor`'s `OnReceive` method to easily send a confirmation message back to the sender of the message that was being validated by `FileValidatorActor`.

But what if, as part of processing a message, you need to send a message to another actor who is not the `Sender` of your current message? `ActorSelection` FTW.

Reporting to multiple endpoints at once

Another common case is that you may have some piece of information you want to report to multiple other actors, that perhaps each run a stats service. Using `ActorSelection`, you could send the same piece of data as a message to all of those services at once, if they shared a similar well-known naming scheme. This is one good use case for a wildcard `ActorSelection`.

Caution: Don't pass `ActorSelection`s around

We encourage you NOT to pass around `ActorSelection`s as pararmeters, the way you do `IActorRef`s. The reason for this is that `ActorSelection`s can be relative instead of absolute, in which case it wouldn't produce the intended effects when passed to an actor with a different location in the hierarchy.

How do I make an `ActorSelection`?

Very simple: `var selection = Context.ActorSelection("/path/to/actorName")`

> NOTE: **the path to an actor includes the name you assign to an actor when you instantiate it, NOT its class name. If you don't assign a name to an actor when you create it, the system will auto-generate a unique name for you**. For example:

```
class FooActor : UntypedActor {}
Props props = Props.Create<FooActor>();

// the ActorPath for myFooActor is "/user/barBazActor"
// NOT "/user/myFooActor" or "/user/FooActor"
IActorRef myFooActor = MyActorSystem.ActorOf(props, "barBazActor");

// if you don't specify a name on creation as below, the system will
// auto generate a name for you, so the actor path will
// be something like "/user/$a"
IActorRef myFooActor = MyActorSystem.ActorOf(props);
```

Do I send a message differently to an `ActorSelection` vs an `IActorRef`?

Nope. You `Tell()` an `ActorSelection` a message just the same as an `IActorRef`:

```
var selection = Context.ActorSelection("/path/to/actorName");
selection.Tell(message);
```

Exercise

Alright, let's get to it. This exercise will be short. We are only making some small optimizations to our system.

Phase 1: Decouple `ConsoleReaderActor` and `FileValidatorActor`

Right now, our `ConsoleReaderActor` needs to be given an `IActorRef` to be able to send the messages it reads from the console off for validation. In the current design, that's easy enough.

BUT, imagine that `ConsoleReaderActor` was far away in the hierarchy from where the `FileValidatorActor` instance was created (`Program.cs` currently). In this case, there is no clean/easy way to pass in the needed `IActorRef` to `ConsoleReaderActor` without also passing it through every intermediary first.

Without `ActorSelection`, you'd have to pass the necessary `IActorRef` through every object between where the handle is created and used. That is coupling more and more objects together--**yuck**!

Let's fix that by **removing the `validationActor` `IActorRef` that we're passing in**. The top of `ConsoleReaderActor` should now look like this:

```
// ConsoleReaderActor.cs
// note: we don't even need our own constructor anymore!
public const string StartCommand = "start";
public const string ExitCommand = "exit";

protected override void OnReceive(object message)
{
    if (message.Equals(StartCommand))
    {
        DoPrintInstructions();
    }

    GetAndValidateInput();
}
```

Then, let's update the call for message validation inside `ConsoleReaderActor` so that the actor doesn't have to hold onto a specific `IActorRef` and can just forward the message read from the console onto an `ActorPath` where it knows validation occurs.

```
// ConsoleReaderActor.GetAndValidateInput

// otherwise, just send the message off for validation
Context.ActorSelection("akka://MyActorSystem/user/validationActor").Tell(message);
```

Finally, let's update `consoleReaderProps` accordingly in `Program.cs` since its constructor no longer takes any arguments:

```
// Program.Main
Props consoleReaderProps = Props.Create<ConsoleReaderActor>();
```

Phase 2: Decouple `FileValidatorActor` and `TailCoordinatorActor`

Just as with `ConsoleReaderActor` and `FileValidatorActor`, the `FileValidatorActor` currently requires an `IActorRef` for the `TailCoordinatorActor` which it does not need. Let's fix that.

First, **remove the `tailCoordinatorActor` argument to the constructor of `FileValidatorActor` and remove the accompanying field on the class**. The top of `FileValidatorActor.cs` should now look like this:

```
// FileValidatorActor.cs
// note that we're no longer storing _tailCoordinatorActor field
private readonly IActorRef _consoleWriterActor;

public FileValidatorActor(IActorRef consoleWriterActor)
{
    _consoleWriterActor = consoleWriterActor;
}
```

Then, let's use `ActorSelection` to communicate between `FileValidatorActor` and `TailCoordinatorActor`! Update `FileValidatorActor` like this:

```
// FileValidatorActor.cs
// start coordinator
Context.ActorSelection("akka://MyActorSystem/user/tailCoordinatorActor").Tell(new TailCoordi
```

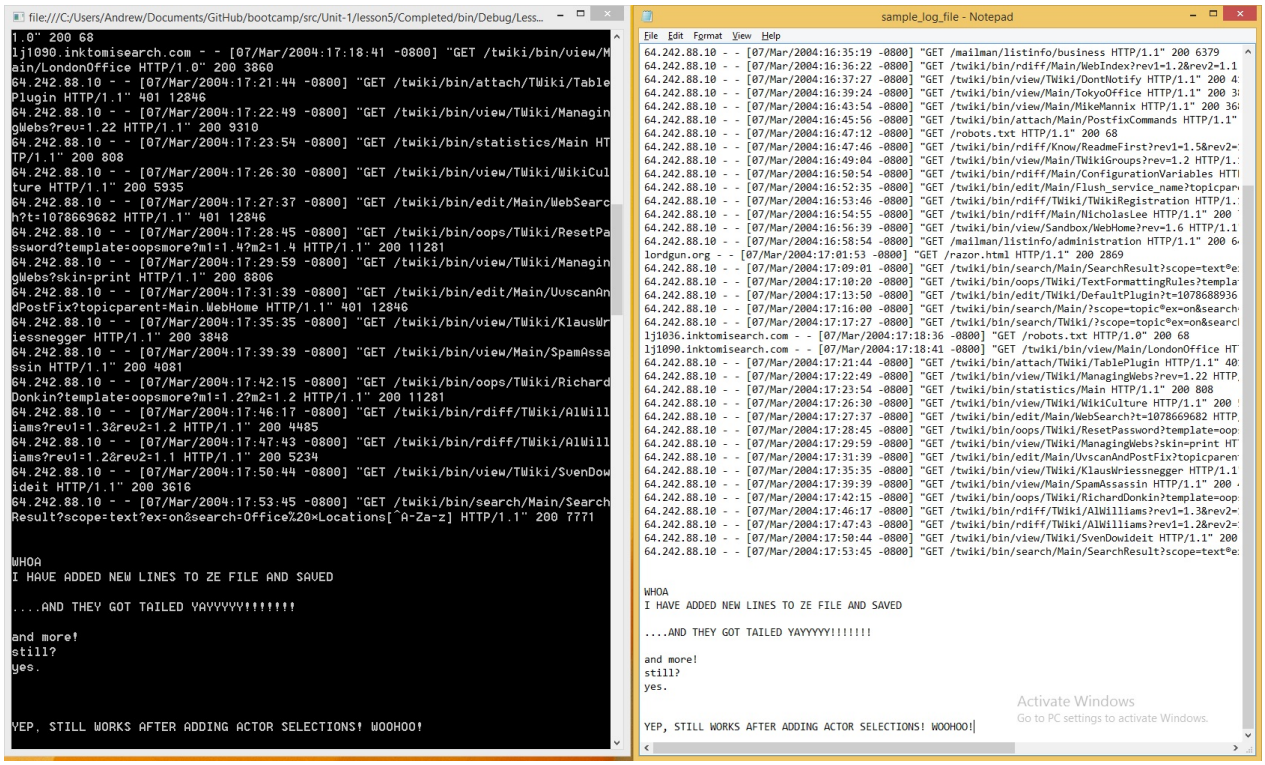
And finally, let's update `fileValidatorProps` in `Program.cs` to reflect the different constructor arguments:

```
// Program.Main
Props fileValidatorActorProps = Props.Create(() => new FileValidatorActor(consoleWriterActor
```

Phase 3: Build and Run!

Awesome! It's time to fire this baby up and see it in action.

Just as with the last lesson, you should be able to hit `F5` and run your log/text file and see additions to it appear in your console.



Hey, wait, go back! What about that `consoleWriterActor` passed to `FileValidatorActor`? Wasn't that unnecessarily coupling actors?

Oh. You're good, you.

We assume you're talking about this `IActorRef` that is still getting passed into

`FileValidatorActor`:

```
// FileValidatorActor.cs
private readonly IActorRef _consoleWriterActor;

public FileValidatorActor(IActorRef consoleWriterActor)
{
    _consoleWriterActor = consoleWriterActor;
}
```

This one is a little counter-intuitive. Here's the deal.

In this case, we aren't using the handle for `consoleWriterActor` to talk directly to it. Instead we are putting that `IActorRef` inside a message that is getting sent somewhere else in the system for processing. When that message is received, the receiving actor will know everything it needs to in order to do its job.

This is actually a good design pattern in the actor model, because it makes the message being passed entirely self-contained and keeps the system as a whole flexible, even if this one actor (`FileValidatorActor`) needs an `IActorRef` passed in and is a little coupled.

Think about what is happening in the `TailCoordinatorActor` which is receiving this message: the job of the `TailCoordinatorActor` is to manage `TailActor`s which will actually observe and report file changes to... somewhere. We get to specify that somewhere up front.

`TailActor` should not have the reporting output location written directly into it. The reporting output location is a task-level detail that should be encapsulated as an instruction within the incoming message. In this case, that task is our custom `StartTail` message, which indeed contains the `IActorRef` for the previously mentioned `consoleWriterActor` as the `reporterActor`.

So, a little counter-intuitively, this pattern actually promotes loose coupling. You'll see it a lot as you go through Akka.NET, especially given the widespread use of the pattern of turning events into messages.

Once you're done

Compare your code to the solution in the Completed folder to see what the instructors included in their samples.

Great job! Almost Done! Onto Lesson 6!

Awesome work! Well done on completing this lesson! We're on the home stretch of Unit 1, and you're doing awesome.

Let's move onto Lesson 6 - The Actor Lifecycle.

Any questions?

Don't be afraid to ask questions :).

Come ask any questions you have, big or small, in this ongoing Bootcamp chat with the Petabridge & Akka.NET teams.

Problems with the code?

If there is a problem with the code running, or something else that needs to be fixed in this lesson, please create an issue and we'll get right on it. This will benefit everyone going through Bootcamp.

Lesson 1.6: The Actor Lifecycle

Wow! Look at you--made it all the way to the end of Unit 1! Congratulations. Seriously. We appreciate and commend you on your dedication to learning and growing as a developer.

This last lesson will wrap up our "fundamentals series" on working with actors, and it ends with a critical concept: actor life cycle.

Key concepts / background

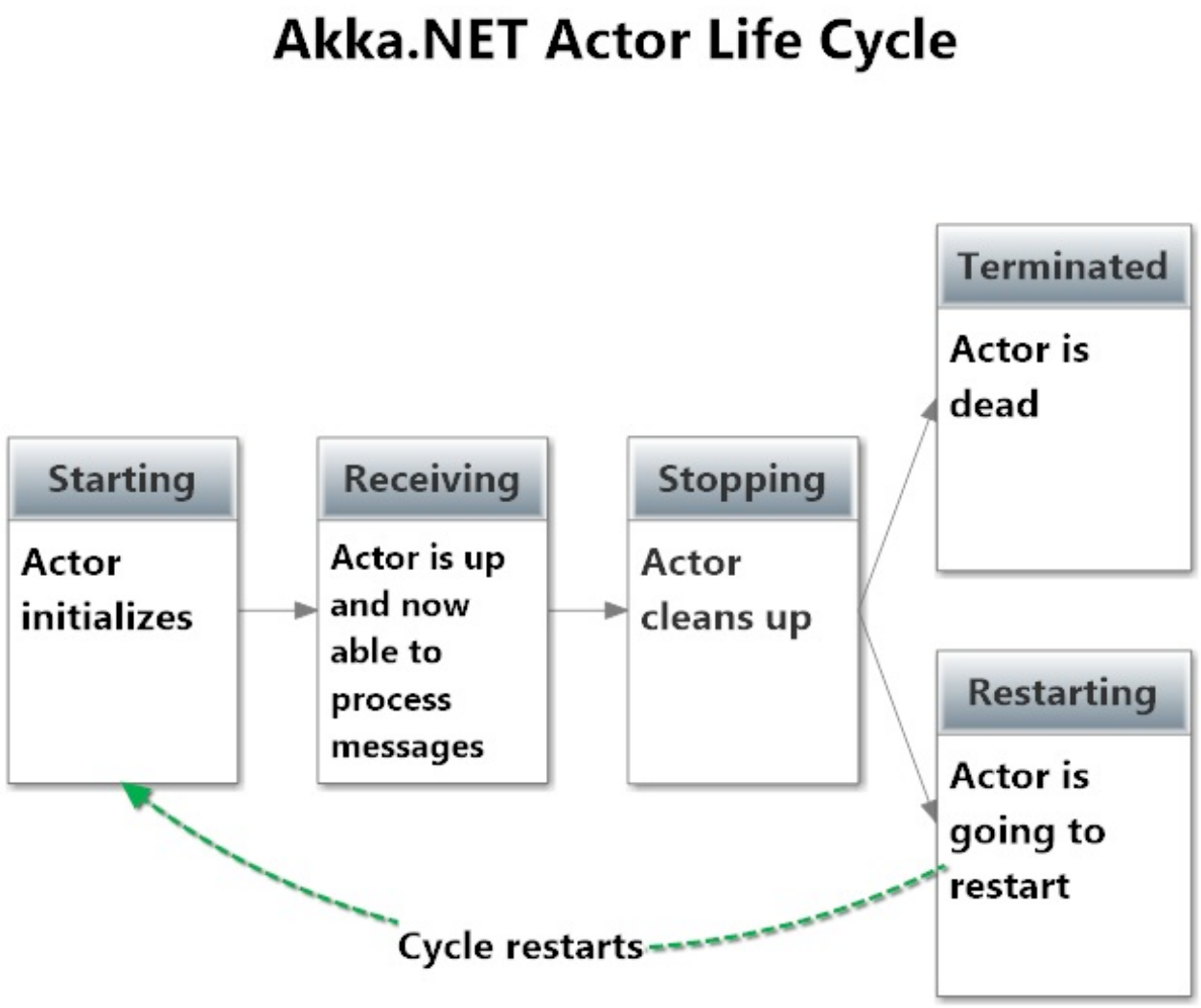
What is the actor life cycle?

Actors have a well-defined life cycle. Actors are created and started, and then they spend most of their lives receiving messages. In the event that you no longer need an actor, you can terminate or "stop" an actor.

What are the stages of the actor life cycle?

There are 5 stages of the actor life cycle in Akka.NET:

1. `Starting`
2. `Receiving`
3. `Stopping`
4. `Terminated`, or
5. `Restarting`



Let's take them in turn.

`Starting`

The actor is waking up! This is the initial state of the actor, when it is being initialized by the `ActorSystem`.

`Receiving`

The actor is now available to process messages. Its `Mailbox` (more on that later) will begin delivering messages into the `OnReceive` method of the actor for processing.

`Stopping`

During this phase, the actor is cleaning up its state. What happens during this phase depends on whether the actor is being terminated, or restarted.

If the actor is being restarted, it's common to save state or messages during this phase to be processed once the actor is back in its Receiving state after the restart.

If the actor is being terminated, all the messages in its `Mailbox` will be sent to the `DeadLetters` mailbox of the `ActorSystem`. `DeadLetters` is a store of undeliverable messages, usually undeliverable because an actor is dead.

`Terminated`

The actor is dead. Any messages sent to its former `IActorRef` will now go to `DeadLetters` instead. The actor cannot be restarted, but a new actor can be created at its former address (which will have a new `IActorRef` but an identical `ActorPath`).

`Restarting`

The actor is about to restart and go back into a `Starting` state.

Life cycle hook methods

So, how can you link into the actor life cycle? Here are the 4 places you can hook in.

`PreStart`

`PreStart` logic gets run before the actor can begin receiving messages and is a good place to put initialization logic. Gets called during restarts too.

`PreRestart`

If your actor accidentally fails (i.e. throws an unhandled Exception) the actor's parent will restart the actor. `PreRestart` is where you can hook in to do cleanup before the actor restarts, or to save the current message for reprocessing later.

`PostStop`

`PostStop` is called once the actor has stopped and is no longer receiving messages. This is a good place to include clean-up logic. PostStop does not get called during actor restarts - only when an actor is being terminated.

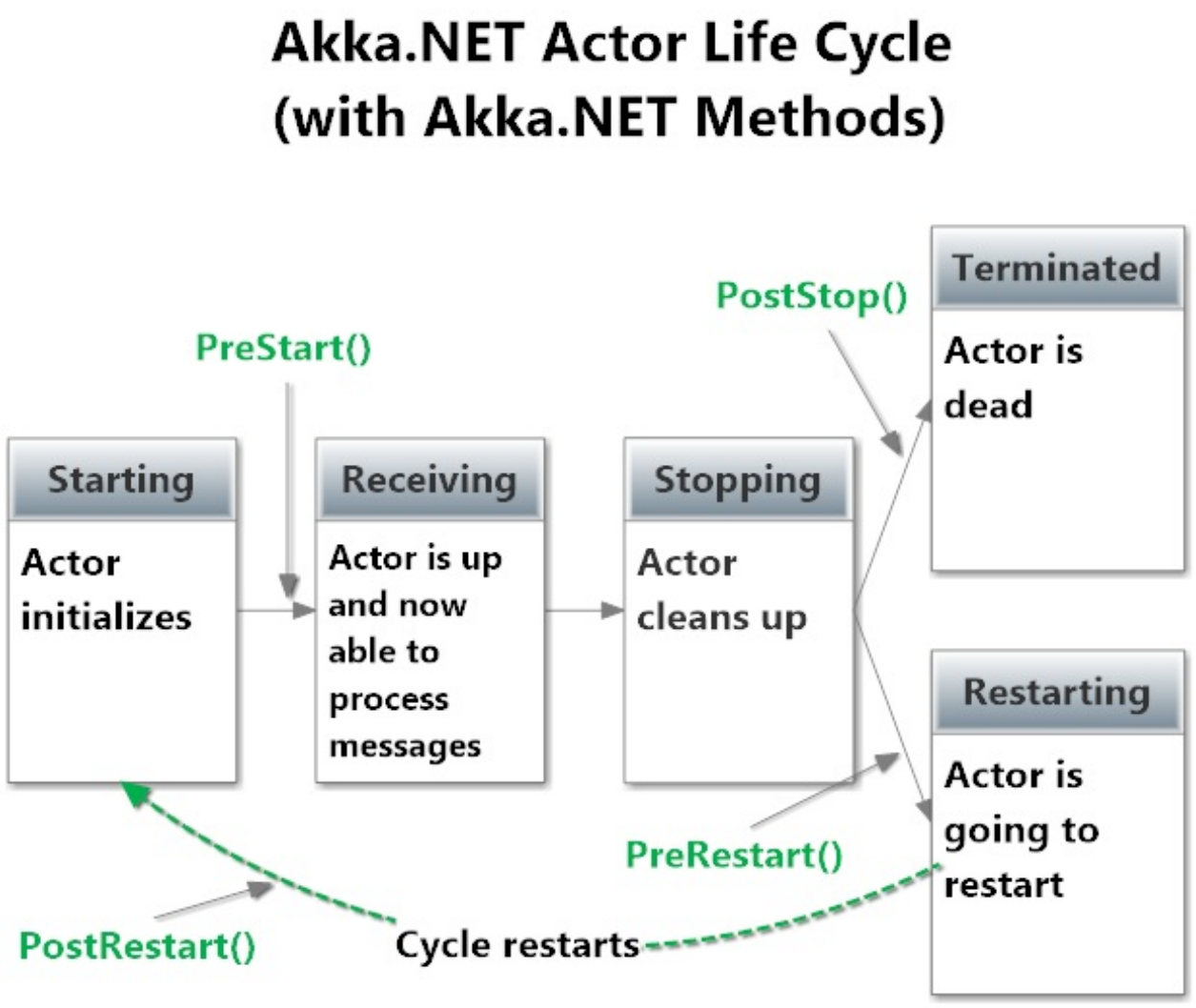
`DeathWatch` is also when an actor notifies any other actors that have subscribed to be alerted when it terminates. `DeathWatch` is just a pub/sub system built into framework for

any actor to be alerted to the termination of any other actor.

`PostRestart`

`PostRestart` is called during restarts after PreRestart but before PreStart. This is a good place to do any additional reporting or diagnosis on the error that caused the actor to crash, beyond what Akka.NET already does for you.

Here's where the hook methods fit into the stages of the life cycle:



How do I hook into the life cycle?

To hook in, you just override the method you want to hook into, like this:

```
 /// <summary>
/// Initialization logic for actor
/// </summary>
```

```
protected override void PreStart()
{
    // do whatever you need to here
}
```

Which are the most commonly used life cycle methods?

`PreStart`

`PreStart` is far and away the most common hook method used. It is used to set up initial state for the actor and run any custom initialization logic your actor needs.

`PostStop`

The second most common place to hook into the life cycle is in `PostStop`, to do custom cleanup logic. For example, you may want to make sure your actor releases file system handles or any other resources it is consuming from the system before it terminates.

`PreRestart`

`PreRestart` is in a distant third to the above methods, but you will occasionally use it. What you use it for is highly dependent on what the actor does, but one common case is to stash a message or otherwise take steps to get it back for reprocessing once the actor restarts.

How does this relate to supervision?

In the event that an actor accidentally crashes (i.e. throws an unhandled Exception,) the actor's supervisor will automatically restart the actor's lifecycle from scratch - without losing any of the remaining messages still in the actor's mailbox.

As we covered in lesson 4 on the actor hierarchy/supervision, what occurs in the case of an unhandled error is determined by the `SupervisionDirective` of the parent. That parent can instruct the child to terminate, restart, or ignore the error and pick up where it left off. The default is to restart, so that any bad state is blown away and the actor starts clean. Restarts are cheap.

Exercise

This final exercise is very short, as our system is already complete. We're just going to use it to optimize the initialization and shutdown of `TailActor`.

Move initialization logic from `TailActor` constructor to `PreStart()`

See all this in the constructor of `TailActor`?

```
// TailActor.cs constructor
// start watching file for changes
_observer = new FileObserver(Self, Path.GetFullPath(_filePath));
_observer.Start();

// open the file stream with shared read/write permissions (so file can be written to while
_fileStream = new FileStream(Path.GetFullPath(_filePath), FileMode.Open, FileAccess.Read,
    FileShare.ReadWrite);
_fileStreamReader = new StreamReader(_fileStream, Encoding.UTF8);

// read the initial contents of the file and send it to console as first message
var text = _fileStreamReader.ReadToEnd();
Self.Tell(new InitialRead(_filePath, text));
```

While it works, initialization logic really belongs in the `PreStart()` method.

Time to use your first life cycle method!

Pull all of the above init logic out of the `TailActor` constructor and move it into `PreStart()`. We'll also need to change `_observer`, `_fileStream`, and `_fileStreamReader` to non-readonly fields since they're moving out of the constructor.

The top of `TailActor.cs` should now look like this

```
// TailActor.cs
private FileObserver _observer;
private Stream _fileStream;
private StreamReader _fileStreamReader;

public TailActor(IActorRef reporterActor, string filePath)
{
    _reporterActor = reporterActor;
    _filePath = filePath;
}
```

```
// we moved all the initialization logic from the constructor
// down below to PreStart!

/// <summary>
/// Initialization logic for actor that will tail changes to a file.
/// </summary>
protected override void PreStart()
{
    // start watching file for changes
    _observer = new FileObserver(Self, Path.GetFullPath(_filePath));
    _observer.Start();

    // open the file stream with shared read/write permissions (so file can be written to wh
    _fileStream = new FileStream(Path.GetFullPath(_filePath), FileMode.Open, FileAccess.Read
        FileShare.ReadWrite);
    _fileStreamReader = new StreamReader(_fileStream, Encoding.UTF8);

    // read the initial contents of the file and send it to console as first message
    var text = _fileStreamReader.ReadToEnd();
    Self.Tell(new InitialRead(_filePath, text));
}
```

Much better! Okay, what's next?

Let's clean up and take good care of our `FileSystem` resources

`TailActor` instances are each storing OS handles in `_fileStreamReader` and `FileObserver`. Let's use `PostStop()` to make sure those handles are cleaned up and we are releasing all our resources back to the OS.

Add this to `TailActor`:

```
// TailActor.cs
/// <summary>
/// Cleanup OS handles for <see cref="_fileStreamReader"/> and <see cref="FileObserver"/>.
/// </summary>
protected override void PostStop()
{
    _observer.Dispose();
    _observer = null;
    _fileStreamReader.Close();
    _fileStreamReader.Dispose();
    base.PostStop();
}
```

Phase 4: Build and Run!

That's it! Hit `F5` to run the solution and it should work exactly the same as before, albeit a little more optimized. :)

Once you're done

Compare your code to the solution in the Completed folder to see what the instructors included in their samples.

Great job!

WOW! YOU WIN! Phenomenal work finishing Unit 1.

Ready for more? Start Unit 2 now.

Any questions?

Don't be afraid to ask questions :).

Come ask any questions you have, big or small, in this ongoing Bootcamp chat with the Petabridge & Akka.NET teams.

Problems with the code?

If there is a problem with the code running, or something else that needs to be fixed in this lesson, please create an issue and we'll get right on it. This will benefit everyone going through Bootcamp.

Akka.NET Bootcamp - Unit 2: Intermediate Akka.NET

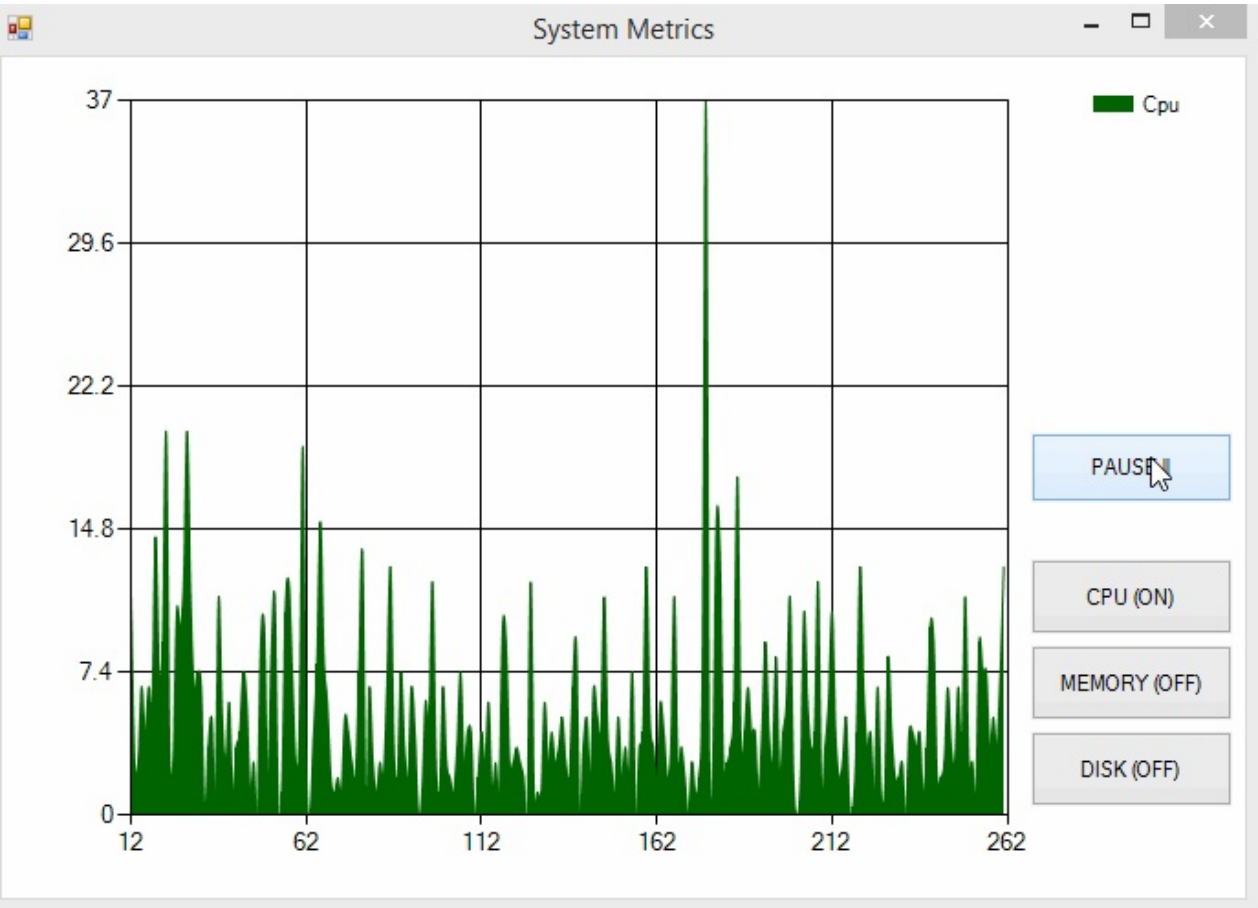
In **Unit 1**, we learned some of the fundamentals of Akka.NET and the Actor Model.

In Unit 2 we will learn some of the more sophisticated concepts behind Akka.NET, such as pattern matching, basic Akka.NET configuration, scheduled messages, and much more!

Concepts you'll learn

In Unit 2 you're going to build your own version of Resource Monitor using Windows Forms, data visualization tools built into .NET, and Performance Counters.

In fact, here's what the final output from lesson 5 looks like:



You're going to build this whole thing using actors, and you'll be surprised at how small your code footprint is when we're done.

In Unit 2 you will learn:

1. How to use HOCON configuration to configure your actors via App.config and Web.config;
2. How to configure your actor's Dispatcher to run on the Windows Forms UI thread, so actors can make operations directly on UI elements without needing to change contexts;
3. How to handle more sophisticated types of pattern matching using `ReceiveActor`;
4. How to use the `Scheduler` to send recurring messages to actors;
5. How to use the Publish-subscribe (pub-sub) pattern between actors;
6. How and why to switch actor's behavior at run-time; and
7. How to `Stash` messages for deferred processing.

Table of Contents

2. **Lesson 2: Using `ReceiveActor` for Better Message Handling**
3. **Lesson 3: Using the `Scheduler` to Send Recurring Messages**
4. **Lesson 4: Switching Actor Behavior at Run-time with `BecomeStacked` and `UnbecomeStacked`**
5. **Lesson 5: Using a `Stash` to Defer Processing of Messages**

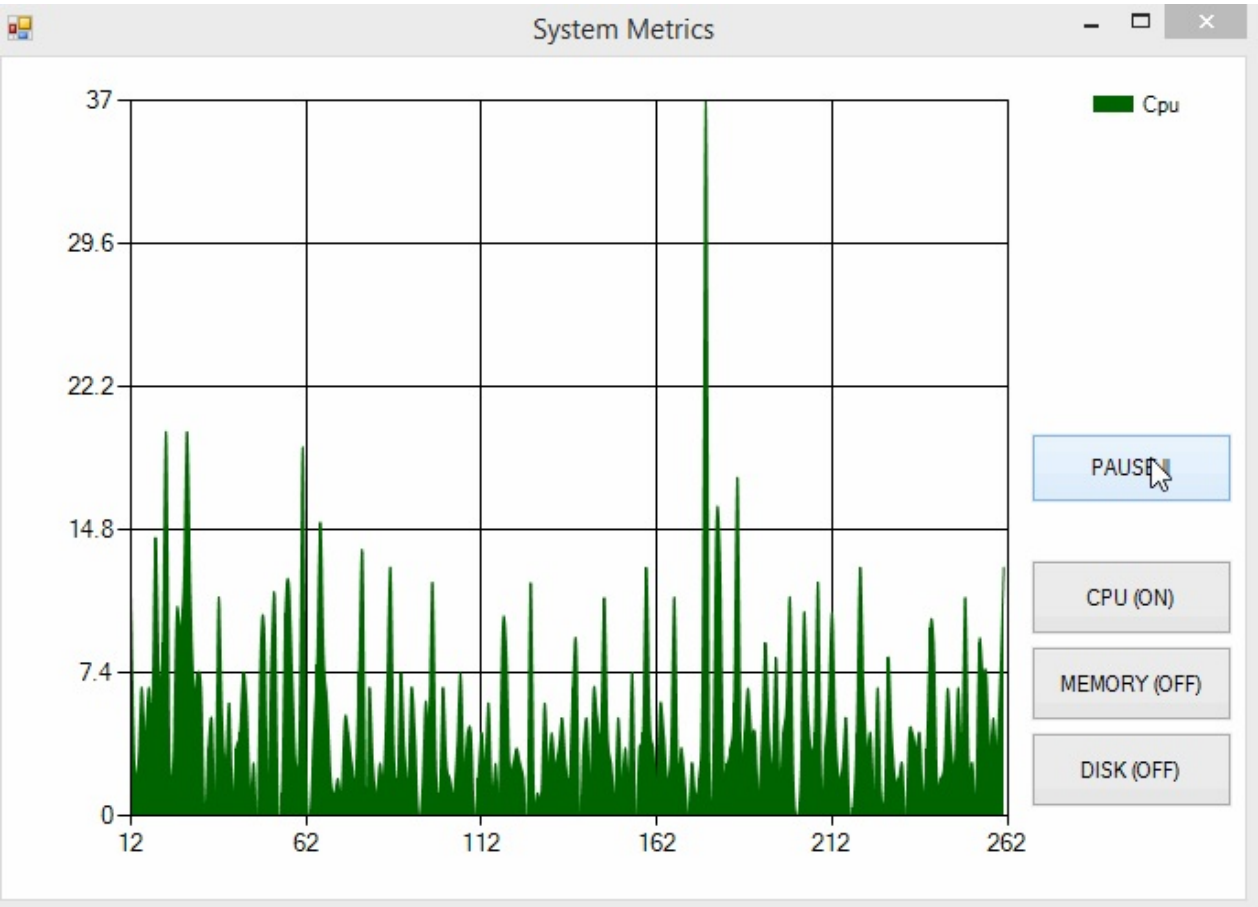
Get Started

To get started, go to the /DoThis/ folder and open `SystemCharting.sln`.

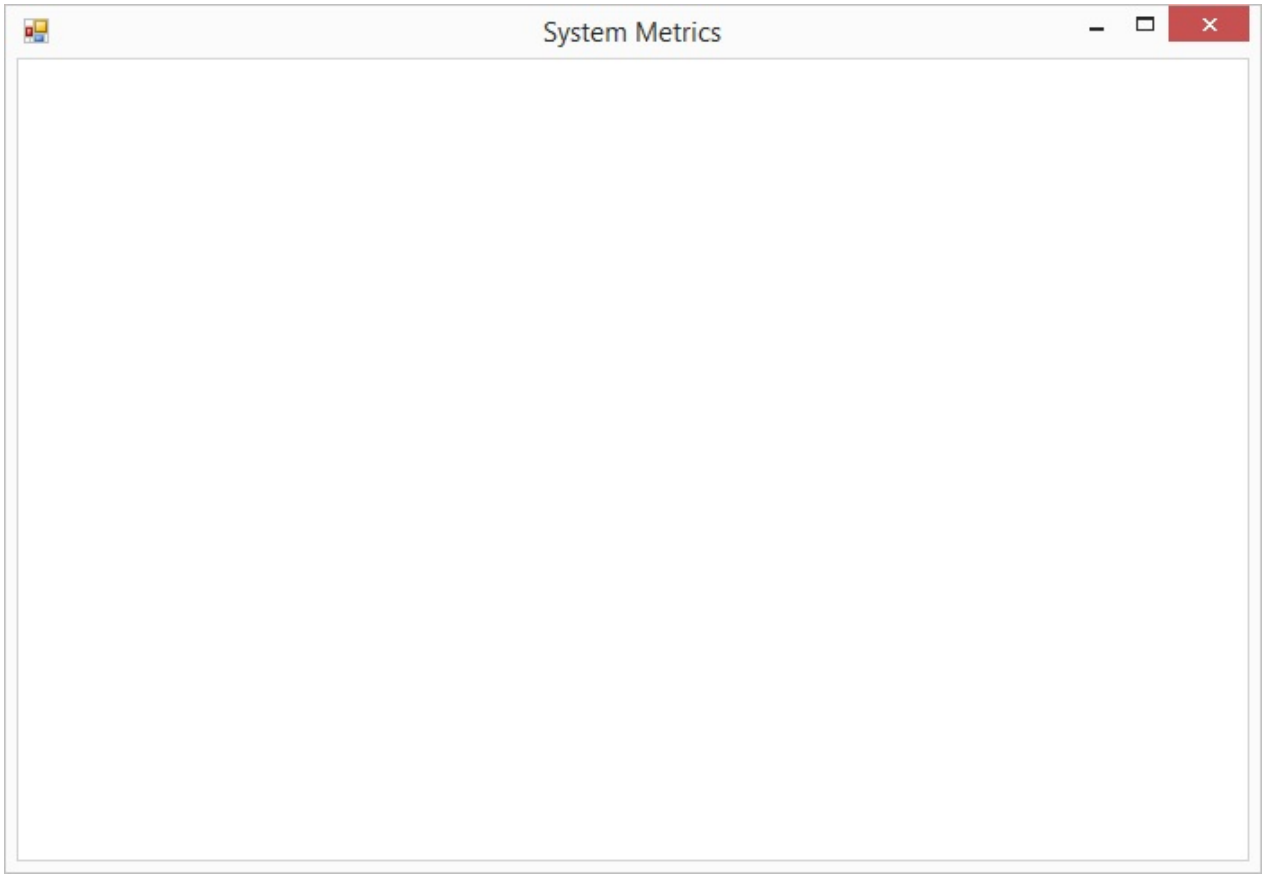
And then go to Lesson 1.

Lesson 2.1: Using HOCON Configuration to Configure Akka.NET

We're going to be spending most of our time in Unit 2 working with the `ChartingActor`, an actor that is responsible for actually plotting all the data on this chart:



BUT, if you try to build and run `SystemCharting.sln` right now (in the /DoThis/ folder) for Unit 2 right away, you'll see the following output:



Well, that's not very exciting. Aren't we supposed to be building a real-time data visualization application in Unit 2? What gives?

Oh wait, there's an exception in the **Debug** window. What does it say?

> [ERROR][2/24/2015 11-48-34 AM][Thread 0010][akka://ChartActors/user/charting] Cross-thread operation not valid: Control 'sysChart' accessed from a thread other than the thread it was created on. Cause: System.InvalidOperationException: Cross-thread operation not valid: Control 'sysChart' accessed from a thread other than the thread it was created on.

What's wrong here?

None of the events we want to chart are getting graphed. Hmm... sounds like our events aren't getting to the UI thread. We need to find some way to dispatch and synchronize events with the UI thread so that our chart is updated!

Does this mean we have to rewrite `ChartingActor` with some evil code to manually synchronize with the UI thread?

Nope! We can relax.

We can solve this problem using HOCON configuration in Akka.NET without updating any of the code that defines `ChartingActor`.

But first, we need to understand `Dispatcher`s.

Key Concepts / Background

`Dispatcher`

What is a `Dispatcher`?

A `Dispatcher` is the piece of glue that pushes messages from your actor's mailbox into your actor instances themselves. That is, the `Dispatcher` is what pushes messages into the `OnReceive()` method of your actors. All actors which share a given `Dispatcher` also share that `Dispatcher`'s threads for parallel execution.

The default dispatcher in Akka.NET is the `ThreadPoolDispatcher`. As you can probably guess, this dispatcher runs all of our actors on top of the CLR `ThreadPool`.

What kinds of `Dispatcher`s are there?

There are several types of `Dispatcher`s we can use with our actors:

`SingleThreadDispatcher`

This `Dispatcher` runs multiple actors on a single thread.

`ThreadPoolDispatcher` (default)

This `Dispatcher` runs actors on top of the CLR `ThreadPool` for maximum concurrency.

`SynchronizedDispatcher`

This `Dispatcher` schedules all actor messages to be processed in the same synchronization context as the caller. 99% of the time, this is where you're going to run actors that need access to the UI thread, such as in client applications.

The `SynchronizedDispatcher` uses the *current* SynchronizationContext to schedule executions.

> **Note:** As a general rule, actors running in the `SynchronizedDispatcher` shouldn't do much work. Avoid doing any extra work that may be done by actors running in other pools.

In this lesson, we're going to use the `SynchronizedDispatcher` to ensure that the `ChartingActor` runs on the UI thread of our WinForms application. That way, the `ChartingActor` can update any UI element it wants without having to do any cross-thread marshalling - the actor's `Dispatcher` can automatically take care of that for us!

`ForkJoinDispatcher`

This `Dispatcher` runs actors on top of a dedicated group of threads, for tunable concurrency.

This is meant for actors that need their own dedicated threads in order to run (that need isolation guarantees). This is primarily used by `System` actors so you won't touch it much.

Is it a bad idea to run actors on the UI thread?

The short answer is "no".

Running actors on the UI thread is fine, as long as those actors don't perform any long-running operations such as disk or network I/O. In fact, *running actors on the UI thread is a smart thing to do for handling UI events and updates*.

Why? Because *running actors on the UI thread eliminates all of the normal synchronization worries* you'd otherwise have to do in a multi-threaded WPF or WinForms app.

> **Remember: Akka.NET actors are lazy**. They don't do any work when they're not receiving messages. They don't consume resources when they're inactive.

How do `Dispatcher`s relate to our broken chart?

As we realized before, our chart isn't updating because the actor doing the graphing (`ChartingActor`) is not synchronizing its events with the UI thread.

To solve this problem, all we have to do is change the `ChartingActor` to use the `CurrentSynchronizationContextDispatcher`, and it will automatically run on the UI thread for us!

BUT: we want to do this without touching our actual actor code. How can we deploy the `ChartingActor` so that it uses the `CurrentSynchronizationContextDispatcher` without modifying the actor itself?

Time to meet HOCON.

HOCON

Akka.NET leverages a configuration format, called HOCON, to allow you to configure your Akka.NET applications with whatever level of granularity you want.

What is HOCON?

HOCON (Human-Optimized Config Object Notation) is a flexible and extensible configuration format. It will allow you to configure everything from Akka.NET's `IActorRefProvider` implementation, logging, network transports, and more commonly - how individual actors are deployed.

Values returned by HOCON are strongly typed (i.e. you can fetch out an `int`, a `Timespan`, etc).

What can I do with HOCON?

HOCON allows you to embed easily-readable configuration inside of the otherwise hard-to-read XML in App.config and Web.config. HOCON also lets you query configs by their section paths, and those sections are exposed strongly typed and parsed values you can use inside your applications.

HOCON also lets you nest and/or chain sections of configuration, creating layers of granularity and providing you a semantically namespaced config.

What is HOCON usually used for?

HOCON is commonly used for tuning logging settings, enabling special modules (such as `Akka.Remote`), or configuring deployments such as the `Dispatcher` for our `ChartingActor` in this lesson.

For example, let's configure an `ActorSystem` with HOCON:

```
var config = ConfigurationFactory.ParseString(@"
akka.remote.helios.tcp {
              transport-class =
           ""Akka.Remote.Transport.Helios.HeliosTcpTransport, Akka.Remote""
              transport-protocol = tcp
              port = 8091
              hostname = ""127.0.0.1""
          }");

var system = ActorSystem.Create("MyActorSystem", config);
```

As you can see in that example, a HOCON `Config` object can be parsed from a `string` using the `ConfigurationFactory.ParseString` method. Once you have a `Config` object, you can then pass this to your `ActorSystem` inside the `ActorSystem.Create` method.

> NOTE: In this example we configured a specific network transport for use with `Akka.Remote`, a concept that goes well beyond what's covered in Unit 2. Don't worry about the specifics for now.

"Deployment"? What's that?

Deployment is a vague concept, but it's closely tied to HOCON. An actor is "deployed" when it is instantiated and put into service within the `ActorSystem` somewhere.

When an actor is instantiated within the `ActorSystem` it can be deployed in one of two places: inside the local process or in another process (this is what `Akka.Remote` does.)

When an actor is deployed by the `ActorSystem`, it has a range of configuration settings. These settings control a wide range of behavior options for the actor, such as: is this actor going to be a router? What `Dispatcher` will it use? What type of mailbox will it have? (More on these concepts in later lessons.)

We haven't gone over what all these options mean, but *the key thing to know for now is that the settings used by the `ActorSystem` to deploy an actor into service can be set within HOCON.*

This also means that you can change the behavior of actors dramatically (by changing these settings) without having to actually touch the actor code itself.

Flexible config FTW!

HOCON can be used inside `App.config` and `Web.config`

Parsing HOCON from a `string` is handy for small configuration sections, but what if you want to be able to take advantage of Configuration Transforms for `App.config` and `Web.config` and all of the other nice tools we have in the `System.Configuration` namespace?

As it turns out, you can use HOCON inside these configuration files too!

Here's an example of using HOCON inside `App.config`:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="akka"
             type="Akka.Configuration.Hocon.AkkaConfigurationSection, Akka" />
  </configSections>

  <akka>
    <hocon>
      <![CDATA[
          akka {
            # here we are configuring log levels
            log-config-on-start = off
            stdout-loglevel = INFO
            loglevel = ERROR
            # this config section will be referenced as akka.actor
            actor {
              provider = "Akka.Remote.RemoteActorRefProvider, Akka.Remote"
              debug {
                  receive = on
                  autoreceive = on
                  lifecycle = on
                  event-stream = on
                  unhandled = on
              }
            }
            # here we're configuring the Akka.Remote module
            remote {
              helios.tcp {
                  transport-class =
            "Akka.Remote.Transport.Helios.HeliosTcpTransport, Akka.Remote"
                  #applied-adapters = []
                  transport-protocol = tcp
                  port = 8091
                  hostname = "127.0.0.1"
              }
            log-remote-lifecycle-events = INFO
          }
      ]]>
```

```
      </hocon>
    </akka>
  </configuration>
```

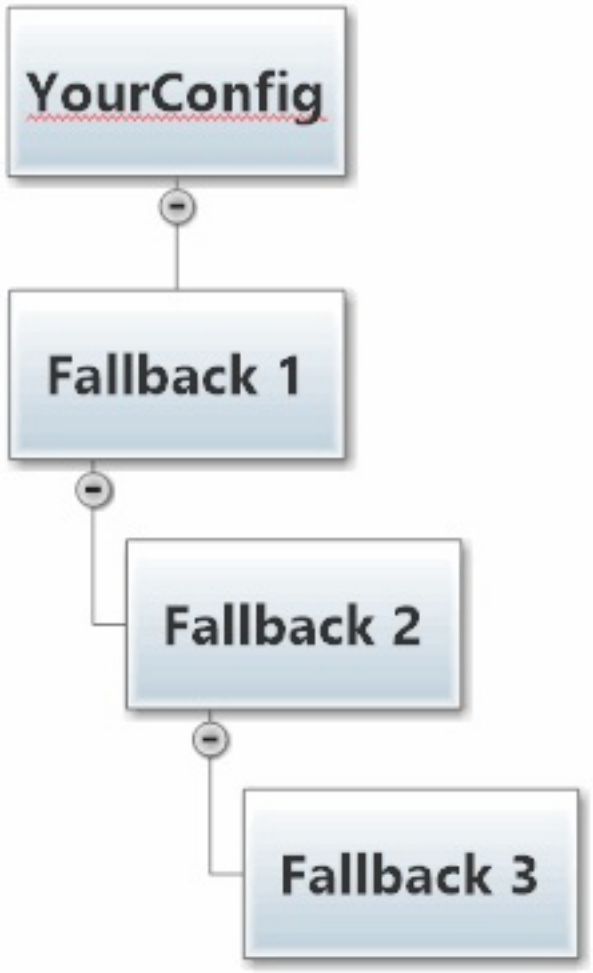
And then we can load this configuration section into our `ActorSystem` via the following code:

```
var system = ActorSystem.Create("Mysystem");
// Loads section.AkkaConfig from App or Web.config automatically
// FYI, section.AkkaConfig is built into Akka.NET for you
```

HOCON Configuration Supports Fallbacks

Although this isn't a concept we leverage explicitly in Unit 2, it's a powerful trait of the `Config` class that comes in handy in lots of production use cases.

HOCON supports the concept of "fallback" configurations - it's easiest to explain this concept visually.



To create something that looks like the diagram above, we have to create a `Config` object that has three fallbacks chained behind it using syntax like this:

```
var f0 = ConfigurationFactory.ParseString("a = bar");
var f1 = ConfigurationFactory.ParseString("b = biz");
var f2 = ConfigurationFactory.ParseString("c = baz");
var f3 = ConfigurationFactory.ParseString("a = foo");

var yourConfig = f0.WithFallback(f1)
                   .WithFallback(f2)
                   .WithFallback(f3);
```

If we request a value for a HOCON object with key "a", using the following code:

```
var a = yourConfig.GetString("a");
```

Then the internal HOCON engine will match the first HOCON file that contains a definition for key `a`. In this case, that is `f0`, which returns the value "bar".

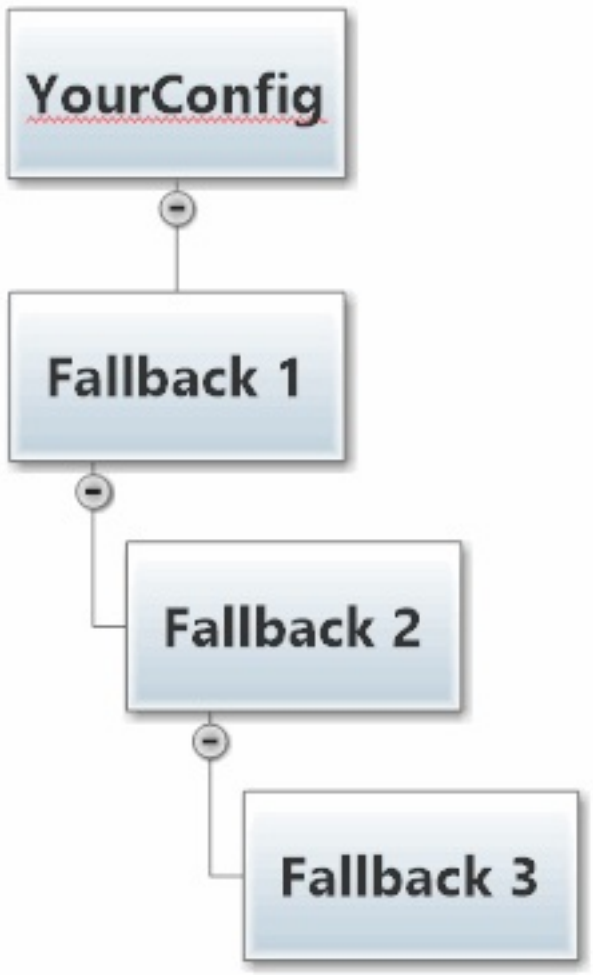
Why wasn't "foo" returned as the value for "a"?

The reason is because HOCON only searches through fallback `Config` objects if a match is NOT found earlier in the `Config` chain. If the top-level `Config` object has a match for `a`, then the fallbacks won't be searched. In this case, a match for `a` was found in `f0` so the `a=foo` in `f3` was never reached.

What happens when there is a HOCON key miss?

What happens if we run the following code, given that `c` isn't defined in `f0` or `f1`?

```
var c = yourConfig.GetString("c");
```



In this case `yourConfig` will fallback twice to `f2` and return "baz" as the value for key `c`.

Now that we understand HOCON, let's use it to fix the `Dispatcher` for `ChartingActor`!

Exercise

We need to configure `ChartingActor` to use the `SynchronizedDispatcher` in order to make our charting work correctly on the UI thread.

Add Akka.NET Config Section to `App.config`

The first thing you need to do is declare the `AkkaConfigurationSection` at the top of your `App.config`:

```
<!-- in App.config file -->
<!-- add this right after the opening <configuration> tag -->
<configSections>
    <section name="akka" type="Akka.Configuration.Hocon.AkkaConfigurationSection, Akka" />
</configSections>
```

Next, add the content of the `AkkaConfigurationSection` to `App.config`:

```
<!-- in App.config file -->
<!-- add this anywhere after <configSections> -->
<akka>
  <hocon>
    <![CDATA[
        akka {
          actor {
            deployment {
              # this nested section will be accessed by akka.actor.deployment
              # used to configure our ChartingActor
              /charting {
                 # causes ChartingActor to run on the UI thread for WinForms
                dispatcher = akka.actor.synchronized-dispatcher
              }
            }
          }
        }
    ]]>
  </hocon>
</akka>
```

We should point out that `akka.actor.synchronized-dispatcher` is the shorthand name built into Akka.NET's default configuration for the `CurrentSynchronizationContextDispatcher`. So you don't need to use a fully-qualified type name.

You might have also noticed that the configuration section that pertains to the `ChartingActor` was declared as `/charting` - **this is because actor deployment is done by the path and name of the actor, not the actor's type**.

Here's how we create the `ChartingActor` inside `Main.cs`:

```
_chartActor = Program.ChartActors.ActorOf(Props.Create(() => new ChartingActor(sysChart)),
```

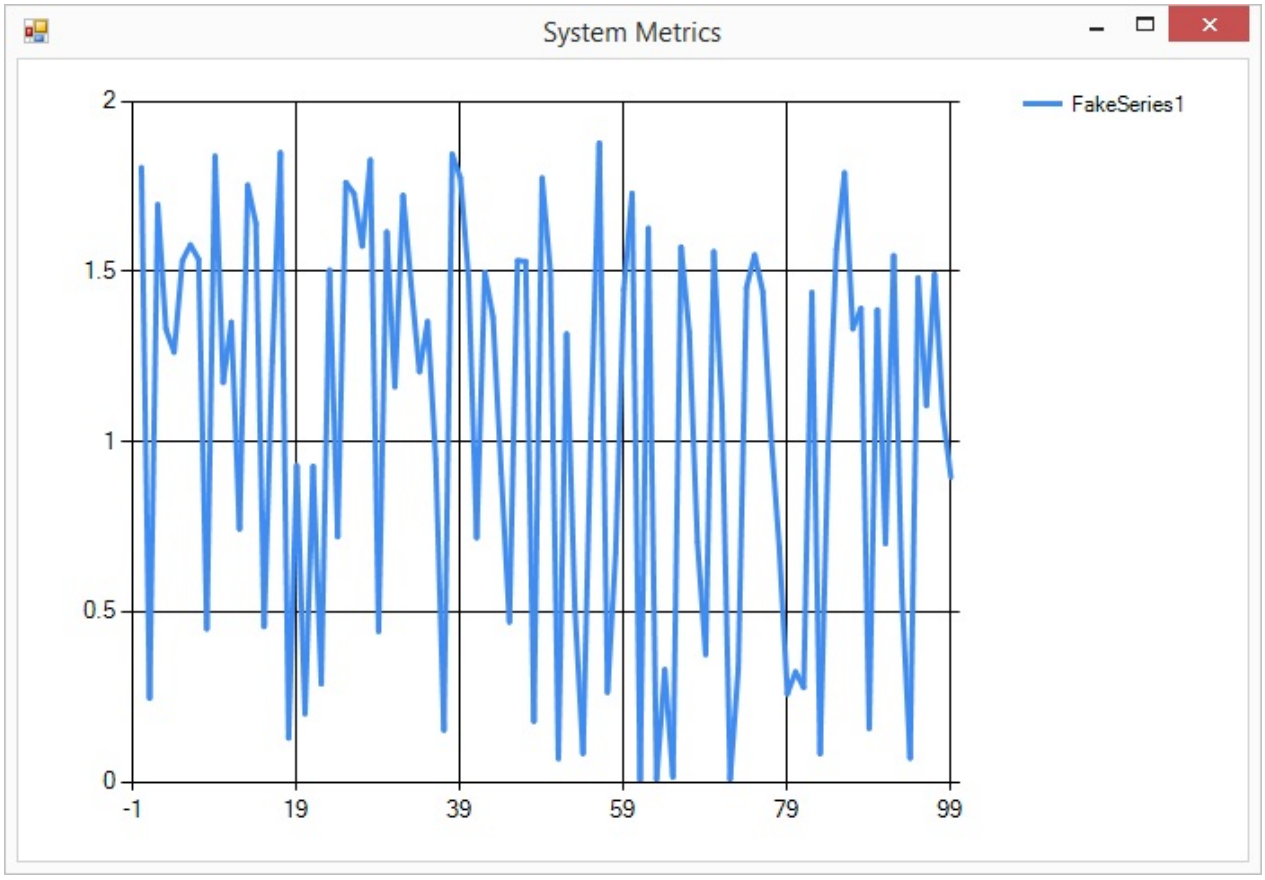
When we call `ActorSystem.ActorOf` the `ActorOf` method will automatically look for any deployments declared in the `akka.actor.deployment` configuration section that correspond to the path of this actor. In this case, the path of this actor is `/user/charting`, which corresponds to the `akka.actor.deployment` values for `/charting` in the config section above.

> As the Akka.NET end-user, you can only specify deployment settings for actors created inside the `/user/` hierarchy. Because of this, you don't need to specify `/user` when you declare your deployment settings - **it's implicit**.
>
> By extension, you also cannot specify how to deploy the `/system` actors. This is up to the `ActorSystem`.

And... we're finished!

Once you're done

Build and run `SystemCharting.sln` and you should see the following:



Compare your code to the code in the /Completed/ folder to compare your final output to what the instructors produced.

Great job!

Nice work on completing your first lesson in Unit 2! We covered a lot of concepts and hopefully you're going to walk away from this with an appreciation for just how powerful Akka.NET's configuration model truly is.

Let's move onto Lesson 2 - Using `ReceiveActor` for Smarter Message Handling.

Further reading

As you probably guessed while reading the HOCON configs above, any line with `#` at the front of it is treated as a comment in HOCON. Learn more about HOCON syntax here.

Any questions?

Don't be afraid to ask questions :).

Come ask any questions you have, big or small, in this ongoing Bootcamp chat with the Petabridge & Akka.NET teams.

Problems with the code?

If there is a problem with the code running, or something else that needs to be fixed in this lesson, please create an issue and we'll get right on it. This will benefit everyone going through Bootcamp.

Lesson 2.2: Using `ReceiveActor` for Smarter Message Handling

In the first unit, you learned how to use the `UntypedActor` (docs) to build your first actors and handle some simple message types.

In this lesson we're going to show you how to use the `ReceiveActor` (docs) to easily handle more sophisticated types of pattern matching and message handling in Akka.NET.

Key Concepts / Background

Pattern matching

Actors in Akka.NET depend heavily on the concept of pattern matching - being able to selectively handle messages based on their .NET Type and/or values.

In the first module, you learned how to use the `UntypedActor` to handle and receive messages using blocks of code that looked a lot like this:

```
protected override void OnReceive(object message){
    if(message is Foo) {
        var foo = message as Foo;
        // do something with foo
    }
    else if(message is Bar) {
        var bar = message as Bar;
        // do something with bar
    }
    //.... other matches
    else {
        // couldn't match this message
        Unhandled(message);
    }
}
```

This method of pattern matching in Akka.NET works great for simple matches, but what if your matching needs were more complex?

Consider how you would handle these use cases with the `UntypedActor` we've seen so far:

1. Match `message` if it's a `string` and begins with "AkkaDotNet" or
2. Match `message` if it's of type `Foo` and `Foo.Count` is less than 4 and `Foo.Max` is greater than 10?

Hmm... if we tried to do all of that inside an `UntypedActor` we'd end up with something like this:

```
protected override void OnReceive(object message) {
    if(message is string
        && message.AsInstanceOf<string>()
            .BeginsWith("AkkaDotNet")){
        var str = message as string;
        // do some work with str...
    }
    else if(message is Foo
            && message.AsInstanceOf<Foo>().Count < 4
            && message.AsInstanceOf<Foo>().Max > 10){
        var foo = message as Foo;
        // do something with foo
    }
    // ... other matches ...
    else {
        // couldn't match this message
        Unhandled(message);
    }
}
```

Yuck! There *has* to be a better way of doing this, right?

Yes, there is! Enter the `ReceiveActor`.

Introducing the `ReceiveActor`

The `ReceiveActor` is built on top of the `UntypedActor` and makes it easy to do sophisticated pattern matching and message handling.

Here's what that ugly code sample from a few moments ago would look like, rewritten with a `ReceiveActor`:

```
public class FooActor : ReceiveActor
```

```
{
    public FooActor()
    {
        Receive<string>(s => s.StartsWith("AkkaDotNet"), s =>
        {
            // handle string
        });

        Receive<Foo>(foo => foo.Count < 4 && foo.Max > 10, foo =>
        {
            // handle foo
        });
    }
}
```

Much better.

So, what's the secret sauce that helped us simplify and clean up all of that pattern matching code from earlier?

The secret sauce of `ReceiveActor`

The secret sauce that cleans up all that pattern matching code is **the `Receive<T>` handler**.

```
// this is what makes the ReceiveActor powerful!
Receive<T>(Predicate<T>, Action<T>);
```

A `ReceiveActor` lets you easily add a layer of strongly typed, compile-time pattern matching to your actors.

You can match messages easily based on type, and then use typed predicates to perform additional checks or validations when deciding whether or not your actor can handle a specific message.

Are there different kinds of `Receive<T>` handlers?

Yes, there are. Here are the different ways to use a `Receive<T>` handler:

1) `Receive<T>(Action<T> handler)`

This executes the message handler only if the message is of type `T`.

2) `Receive<T>(Predicate<T> pred, Action<T> handler)`

This executes the message handler only if the message is of type `T` **AND** the predicate function returns true for this instance of `T`.

3) `Receive<T>(Action<T> handler, Predicate<T> pred)`

Same as the previous.

4) `Receive(Type type, Action<object> handler)`

This is a concrete version of the typed + predicate message handlers from before (no longer generic).

5) `Receive(Type type, Action<object> handler, Predicate<object> pred)`

Same as the previous.

6) `ReceiveAny()`

This is a catch-all handler which accepts all `object` instances. This is usually used to handle any messages that aren't handled by a previous, more specific `Receive()` handler.

The order in which you declare `Receive<T>` handlers matters

What happens if we need to handle overlapping types of messages?

Consider the below messages: they start with the same substring, but assume they need to be handled differently.

1. `string` messages that begin with `AkkaDotNetSuccess`, AND
2. `string` messages that begin with `AkkaDotNet`?

What would happen if our `ReceiveActor` was written like this?

```
public class StringActor : ReceiveActor
{
    public StringActor()
```

```
    {
        Receive<string>(s => s.StartsWith("AkkaDotNet"), s =>
        {
            // handle string
        });

        Receive<string>(s => s.StartsWith("AkkaDotNetSuccess"), s =>
        {
            // handle string
        });
    }
}
```

What happens in this case is that the second handler (for `s.StartsWith("AkkaDotNetSuccess")`) is never invoked. Why not?

The order of the `Receive<T>` handlers matters!

This is because `ReceiveActor` **will handle a message using the *first* matching handler, not the *best* matching handler** and it evaluates its handlers for each message in the order in which they were declared.

So, how do we solve the above problem, where our handler for strings starting with "AkkaDotNetSuccess" is never triggered?

Simple: *we fix this problem by making sure that the more specific handlers come first*.

```
public class StringActor : ReceiveActor
{
    public StringActor()
    {
        // Now works as expected
        Receive<string>(s => s.StartsWith("AkkaDotNetSuccess"), s =>
        {
            // handle string
        });

        Receive<string>(s => s.StartsWith("AkkaDotNet"), s =>
        {
            // handle string
        });
    }
}
```

Where do I define message handlers in a `ReceiveActor`?

`ReceiveActor`s do not have an `OnReceive()` method.

Instead, you must hook up `Receive` message handlers directly in the `ReceiveActor` constructor, or in a method called to by that constructor.

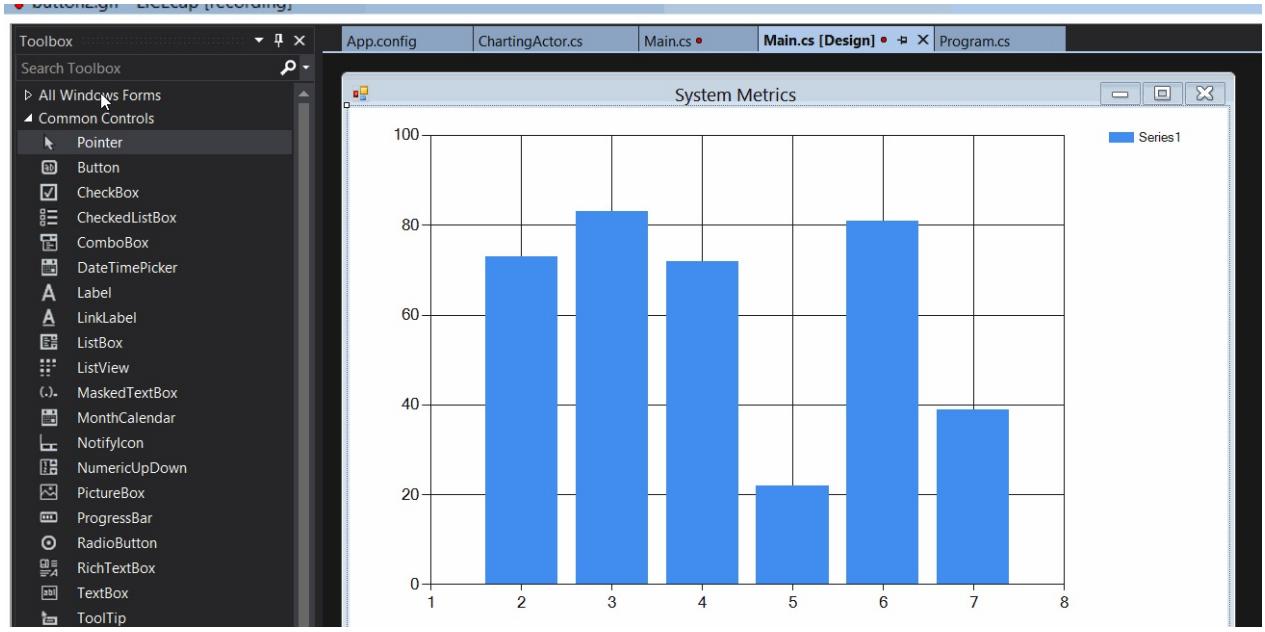
Knowing this, we can put `ReceiveActor` to work for us.

Exercise

In this exercise we're going to add the ability to add multiple data series to our chart, and we're going to modify the `ChartingActor` to handle commands to do this.

Step 1 - Add an "Add Series" Button to the UI

First thing we're going to do is add a new button called "Add Series" to our form. Go to the **[Design]** view of `Main.cs` and drag a `Button` onto the UI from the Toolbox. Here's where we put the button:



Double click on the button in the **[Design]** view and Visual Studio will automatically add a click handler for you inside `Main.cs`. That generated handler should look like this:

```
// automatically added inside Main.cs if you double click on button in designer
private void button1_Click(object sender, EventArgs e)
{
```

```
}
```

Leave this blank for now. We'll wire this button to our `ChartingActor` shortly.

Step 2 - Add an `AddSeries` Message Type to the `ChartingActor`

Let's define a new message class for putting additional `Series` on the `Chart` managed by the `ChartingActor`. We'll create the `AddSeries` message type to signify that we want to add a new `Series`.

Add the following code to `ChartingActor.cs` inside the `Messages` region:

```
// Actors/ChartingActor.cs, inside the #Messages region

/// <summary>
/// Add a new <see cref="Series"/> to the chart
/// </summary>
public class AddSeries
{
    public AddSeries(Series series)
    {
        Series = series;
    }

    public Series Series { get; private set; }
}
```

Step 3 - Have `ChartingActor` Inherit from `ReceiveActor`

Now for the meaty part - changing the `ChartingActor` from an `UntypedActor` to a `ReceiveActor`.

Now, let's change the declaration for `ChartingActor`. Change this:

```
// Actors/ChartingActor.cs
public class ChartingActor : UntypedActor
```

to this:

```
// Actors/ChartingActor.cs
public class ChartingActor : ReceiveActor
```

Remove the `OnReceive` method from `ChartingActor`

Don't forget to **delete the current `OnReceive` method for the `ChartingActor`**. Now that `ChartingActor` is a `ReceiveActor`, it doesn't need an `OnReceive()` method.

Step 4 - Define `Receive<T>` Handlers for `ChartingActor`

Right now our `ChartingActor` can't handle any messages that are sent to it - so let's fix that by defining some `Receive<T>` handlers for the types of messages we want to accept.

First things first, add the following method to the `Individual Message Type Handlers` region of the `ChartingActor`:

```
// Actors/ChartingActor.cs in the ChartingActor class (Individual Message Type Handlers regi

private void HandleAddSeries(AddSeries series)
{
    if(!string.IsNullOrEmpty(series.Series.Name) && !_seriesIndex.ContainsKey(series.Series.
    {
        _seriesIndex.Add(series.Series.Name, series.Series);
        _chart.Series.Add(series.Series);
    }
}
```

And now let's modify the constructor of the `ChartingActor` to set a `Recieve<T>` hook for `InitializeChart` and `AddSeries`.

```
// Actors/ChartingActor.cs in the ChartingActor constructor

public ChartingActor(Chart chart, Dictionary<string, Series> seriesIndex)
{
    _chart = chart;
    _seriesIndex = seriesIndex;

    Receive<InitializeChart>(ic => HandleInitialize(ic));
    Receive<AddSeries>(addSeries => HandleAddSeries(addSeries));
}
```

> **NOTE**: The other constructor for `ChartingActor`, `ChartingActor(Chart chart)` doesn't need to be modified, as it calls `ChartingActor(Chart chart, Dictionary<string, Series> seriesIndex)` anyway.

And with that, our `ChartingActor` should now be able to receive and process both types of messages easily.

Step 5 - Have the Button Clicked Handler for "Add Series" Button Send `ChartingActor` an `AddSeries` Message

Let's go back to the click handler we added for the button in Step 1.

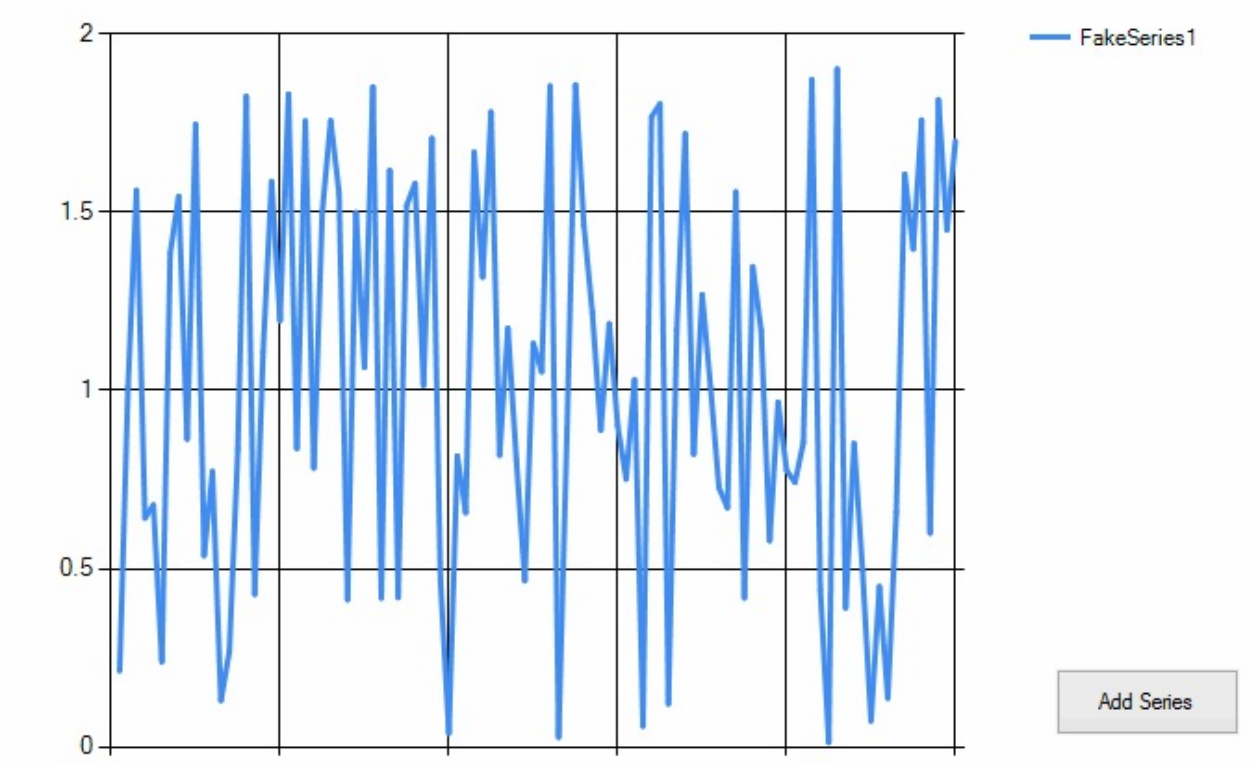
In `Main.cs`, add this code to the body of the click handler:

```
// Main.cs - class Main
private void button1_Click(object sender, EventArgs e)
{
    var series = ChartDataHelper.RandomSeries("FakeSeries" + _seriesCounter.GetAndIncrement(
    _chartActor.Tell(new ChartingActor.AddSeries(series));
}
```

And that should do it!

Once you're done

Build and run `SystemCharting.sln` and you should see the following:



Compare your code to the code in the /Completed/ folder to compare your final output to what the instructors produced.

Great job!

Nice work, again. After having completed this lesson you should have a much better understanding of pattern matching in Akka.NET and an appreciation for how `ReceiveActor` is different than `UntypedActor`.

Let's move onto Lesson 3 - Using the `Scheduler` to Send Recurring Messages.

Any questions?

Don't be afraid to ask questions :).

Come ask any questions you have, big or small, in this ongoing Bootcamp chat with the Petabridge & Akka.NET teams.

Problems with the code?

If there is a problem with the code running, or something else that needs to be fixed in

this lesson, please create an issue and we'll get right on it. This will benefit everyone going through Bootcamp.

Lesson 2.3: Using the `Scheduler` to Send Messages Later

Welcome to Lesson 2.3!

Where are we? At this point, we have our basic chart set up, along with our `ChartingActor` which is supposed to be graphing system metrics. Except right now, `ChartingActor` isn't actually graphing anything! It's time to change that.

In this lesson, we'll be hooking up the various components of our system to make our Resource Monitor application actually chart system resource consumption! **This is a big lesson—it's the core of Unit 2—so get your coffee and get comfortable!**

To make our resource monitoring app work as intended, we need to wire up `ChartingActor` to the actual system Performance Counters for the graph data. This needs to happen on an ongoing basis so that our chart regularly updates.

One of the most powerful capabilities Akka.NET exposes is the ability to schedule messages to be sent in the future, including regularly occurring messages. And it turns out, this is exactly the functionality we need to have `ChartingActor` regularly update our graphs.

In this lesson you'll learn two powerful Akka.NET concepts:

1. How to use the `Scheduler`, and
2. How to implement the Publish-subscribe (pub-sub) pattern using actors. This is a powerful technique for creating reactive systems.

Key Concepts / Background

How do you get an actor to do something in the future? And what if you want that actor to do something on a recurring basis in the future?

Perhaps you want an actor to periodically fetch information, or to occasionally ping another actor within the system for its status.

Akka.NET provides a mechanism for doing just this sort of thing. Meet your new best friend: the `Scheduler`.

What is the `Scheduler`?

The `ActorSystem.Scheduler` (docs) is a singleton within every `ActorSystem` that allows you to schedule messages to be sent to an actor in the future. The `Scheduler` can send both one-off and recurring messages.

How do I use the `Scheduler`?

As we mentioned, you can schedule one-off or recurring messages to an actor.

You can also schedule an `Action` to occur in the future, instead of sending a message to an actor.

Access the `Scheduler` via the `ActorSystem`

`Scheduler` must be accessed through the `ActorSystem`, like so:

```
// inside Main.cs we have direct handle to the ActorSystem
var system = ActorSystem.Create("MySystem");
system.Scheduler.ScheduleTellOnce(TimeSpan.FromMinutes(30),
                              someActor,
                              someMessage, ActorRefs.Nobody);

// but inside an actor, we access the ActorSystem via the ActorContext
Context.System.Scheduler.ScheduleTellOnce(TimeSpan.FromMinutes(30),
                                          someActor,
                                          someMessage, ActorRefs.Nobody);
```

Schedule one-off messages with `ScheduleTellOnce()`

Let's say we want to have one of our actors fetch the latest content from an RSS feed 30 minutes in the future. We can use `IScheduler.ScheduleTellOnce()` to do that:

```
var system = ActorSystem.Create("MySystem");
var someActor = system.ActorOf<SomeActor>("someActor");
var someMessage = new FetchFeed() {Url = ...};
// schedule the message
system
```

```
    .Scheduler
    .ScheduleTellOnce(TimeSpan.FromMinutes(30), // initial delay of 30 min
              someActor, someMessage, ActorRefs.Nobody);
```

Voila! `someActor` will receive `someMessage` in 30 minutes time.

Schedule recurring messages with `ScheduleTellRepeatedly()`

Now, **what if we want to schedule this message to be delivered once *every 30 minutes*?**

For this we can use the following `IScheduler.ScheduleTellRepeatedly()` overload.

```
var system = ActorSystem.Create("MySystem");
var someActor = system.ActorOf<SomeActor>("someActor");
var someMessage = new FetchFeed() {Url = ...};
// schedule recurring message
system
   .Scheduler
   .ScheduleTellRepeatedly(TimeSpan.FromMinutes(30), // initial delay of 30 min
              TimeSpan.FromMinutes(30), // recur every 30 minutes
              someActor, someMessage, ActorRefs.Nobody);
```

That's it!

How do I cancel a scheduled message?

What happens if we need to cancel a scheduled or recurring message? We use a `ICancelable`, which we can create using a `Cancelable` instance.

First, the message must be scheduled so that it can be cancelled. If a message is cancelable, we then just have to call `Cancel()` on our handle to the `ICancelable` and it will not be delivered. For example:

```
var system = ActorSystem.Create("MySystem");
var cancellation = new Cancelable(system.Scheduler);
var someActor = system.ActorOf<SomeActor>("someActor");
var someMessage = new FetchFeed() {Url = ...};

// first, set up the message so that it can be canceled
system
```

```
    .Scheduler
    .ScheduleTellRepeatedly(TimeSpan.FromMinutes(30), TimeSpan.FromMinutes(30)
                  someActor, someMessage, ActorRefs.Nobody,
                    cancellation); // add cancellation support

// here we actually cancel the message and prevent it from being delivered
cancellation.Cancel();
```

Alternative: get an `ICancelable` task using `ScheduleTellRepeatedlyCancelable`

One of the new `IScheduler` methods we introduced in Akka.NET v1.0 is the `ScheduleTellRepeatedlyCancelable` extension method.This extension method inlines the process of creating an `ICancelable` instance for your recurring messages and simply returns an `ICancelable` for you.

```
var system = ActorSystem.Create("MySystem");
var someActor = system.ActorOf<SomeActor>("someActor");
var someMessage = new FetchFeed() {Url = ...};

// cancellable recurring message send created automatically
var cancellation =  system
   .Scheduler
   .ScheduleTellRepeatedlyCancelable(TimeSpan.FromMinutes(30), TimeSpan.FromMinutes(30)
                  someActor, someMessage, ActorRefs.Nobody);

// here we actually cancel the message and prevent it from being delivered
cancellation.Cancel();
```

This is a more concise alternative to the previous example, and we recommend using it going forward even though we won't be using it in this bootcamp.

How precise is the timing of scheduled messages?

Scheduled messages are more than precise enough for all the use cases we've come across.

That said, there are two situations of imprecision that we're aware of:

1. Scheduled messages are scheduled onto the CLR threadpool and use `Task.Delay` under the hood. If there is a high load on the CLR threadpool, the task might finish a little later than planned. There is no guarantee that the task will execute at

EXACTLY the millisecond you expect.
2. If your scheduling requirements demand precision below 15 milliseconds then the `Scheduler` is not precise enough for you. Nor is any typical operating system such as Windows, OSX, or Linux. This is because ~15ms is the interval in which Windows and other general OSes update their system clock ("clock resolution"), so these OSs can't support any timing more precise than their own system clocks.

What are the various overloads of `Schedule` and `ScheduleOnce`?

Here are all the overload options you have for scheduling a message.

Overloads of `ScheduleTellRepeatedly`

These are the various API calls you can make to schedule recurring messages.

Refer to the `IScheduler` API documentation.

Overloads of `ScheduleTellOnce`

These are the various API calls you can make to schedule one-off messages.

Refer to the `IScheduler` API documentation.

How do I do Pub/Sub with Akka.NET Actors?

It's actually very simple. Many people expect this to be very complicated and are suspicious that there isn't more code involved. Rest assured, there's nothing magic about pub/sub with Akka.NET actors. It can literally be as simple as this:

```
public class PubActor : ReceiveActor
{
  // HashSet automatically eliminates duplicates
  private HashSet<IActorRef> _subscribers;

  PubActor()
  {
    _subscribers = new HashSet<IActorRef>();

    Receive<Subscribe>(sub =>
    {
```

```
        _subscribers.Add(sub.IActorRef);
      });

      Receive<MessageSubscribersWant>(message =>
      {
        // notify each subscriber
        foreach (var sub in _subscribers)
        {
          sub.Tell(message);
        }
      });

      Receive<Unsubscribe>(unsub =>
      {
        _subscribers.Remove(unsub.IActorRef);
      });
    }
  }
```

Pub/sub is trivial to implement in Akka.NET and it's a pattern you can feel comfortable using regularly when you have scenarios that align well with it.

Now that you're familiar with how the `Scheduler` works, lets put it to use and make our charting UI reactive!

Exercise

HEADS UP: This section is where 90% of the work happens in all of Unit 2. We're going to add a few new actors who are responsible for setting up pub/sub relationships with the `ChartingActor` in order to graph `PeformanceCounter` data at regular intervals.

Step 1 - Delete the "Add Series" Button and Click Handler from Lesson 2

We're not going to need it. **Delete the "Add Series" button** from the **[Design]** view of `Main.cs` and remove the click handler:

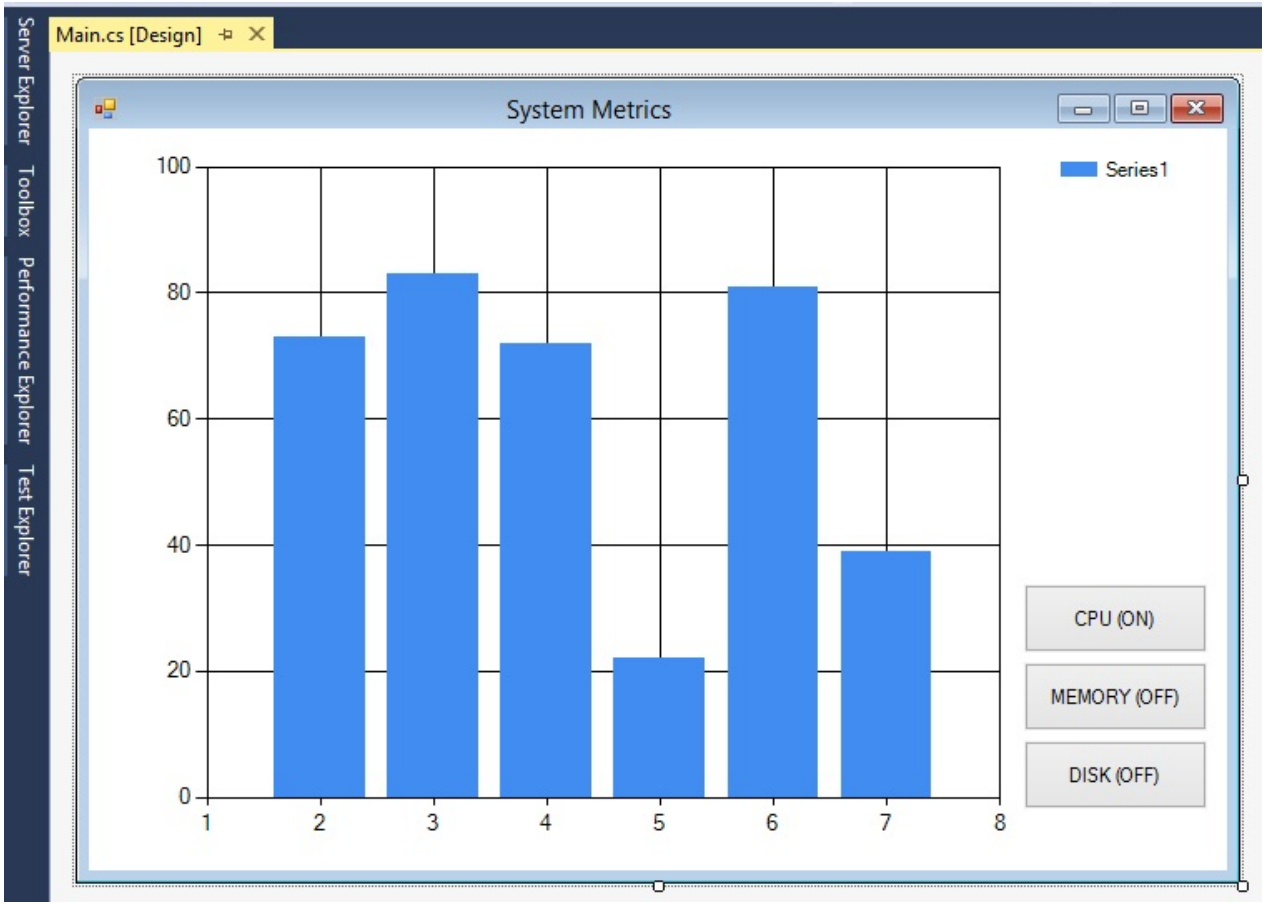
```
// Main.cs - Main
// DELETE THIS:
private void button1_Click(object sender, EventArgs e)
{
    var series = ChartDataHelper.RandomSeries("FakeSeries" + _seriesCounter.GetAndIncrement(
    _chartActor.Tell(new ChartingActor.AddSeries(series));
}
```

Step 2 - Add 3 New Buttons to `Main.cs`

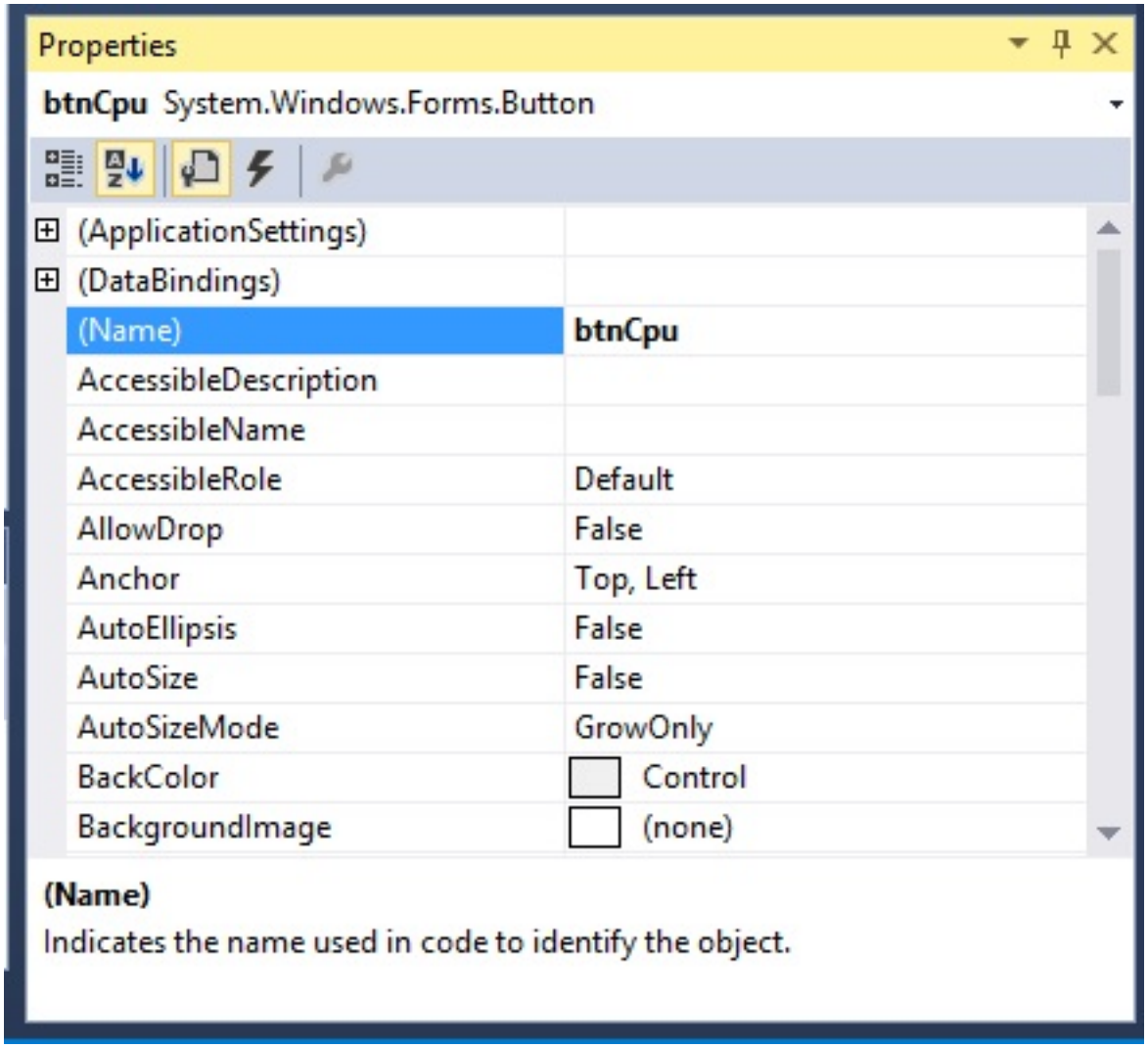
We're going to add three new buttons and click handlers for each:

- **CPU (ON)**
- **MEMORY (OFF)**
- **DISK (OFF)**

Your **[Design]** view in Visual Studio for `Main.cs` should look like this:



Make sure that you given a descriptive name to each of these buttons, because we're going to need to refer to them later. You can set a descriptive name for each using the **Properties** window in Visual Studio:



Here are the names we'll be using for each button when we refer to them later:

- **CPU (ON)** - `btnCpu`
- **MEMORY (OFF)** - `btnMemory`
- **DISK (OFF)** - `btnDisk`

Once you've renamed your buttons, *add click handlers for each button by double-clicking on the button* in the **[Design]** view.

```
// Main.cs - Main
private void btnCpu_Click(object sender, EventArgs e)
{

}

private void btnMemory_Click(object sender, EventArgs e)
{

}
```

```
private void btnDisk_Click(object sender, EventArgs e)
{

}
```

We'll fill these handlers in later.

Step 3 - Add Some New Message Types

We're going to add a few new actors to our project in a moment, but before we do that let's create a new file inside the `/Actors` folder in our project and define some new message types:

```
// Actors/ChartingMessages.cs

using Akka.Actor;

namespace ChartApp.Actors
{
    #region Reporting

    /// <summary>
    /// Signal used to indicate that it's time to sample all counters
    /// </summary>
    public class GatherMetrics { }

    /// <summary>
    /// Metric data at the time of sample
    /// </summary>
    public class Metric
    {
        public Metric(string series, float counterValue)
        {
            CounterValue = counterValue;
            Series = series;
        }

        public string Series { get; private set; }

        public float CounterValue { get; private set; }
    }

    #endregion

    #region Performance Counter Management

    /// <summary>
    /// All types of counters supported by this example
    /// </summary>
```

```
        public enum CounterType
        {
            Cpu,
            Memory,
            Disk
        }

        /// <summary>
        /// Enables a counter and begins publishing values to <see cref="Subscriber"/>.
        /// </summary>
        public class SubscribeCounter
        {
            public SubscribeCounter(CounterType counter, IActorRef subscriber)
            {
                Subscriber = subscriber;
                Counter = counter;
            }

            public CounterType Counter { get; private set; }

            public IActorRef Subscriber { get; private set; }
        }

        /// <summary>
        /// Unsubscribes <see cref="Subscriber"/> from receiving updates for a given counter
        /// </summary>
        public class UnsubscribeCounter
        {
            public UnsubscribeCounter(CounterType counter, IActorRef subscriber)
            {
                Subscriber = subscriber;
                Counter = counter;
            }

            public CounterType Counter { get; private set; }

            public IActorRef Subscriber { get; private set; }
        }

        #endregion
    }
```

Now we can start adding the actors who depend on these message definitions.

Step 4 - Create the `PerformanceCounterActor`

The `PerformanceCounterActor` is the actor who's going to publish `PerformanceCounter` values to the `ChartingActor` using Pub/Sub and the `Scheduler`.

Create a new file in the `/Actors` folder called `PerformanceCounterActor.cs` and type the

following:

```
// Actors/PerformanceCounterActor.cs

using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Threading;
using Akka.Actor;

namespace ChartApp.Actors
{
    /// <summary>
    /// Actor responsible for monitoring a specific <see cref="PerformanceCounter"/>
    /// </summary>
    public class PerformanceCounterActor : UntypedActor
    {
        private readonly string _seriesName;
        private readonly Func<PerformanceCounter> _performanceCounterGenerator;
        private PerformanceCounter _counter;

        private readonly HashSet<IActorRef> _subscriptions;
        private readonly ICancelable _cancelPublishing;

        public PerformanceCounterActor(string seriesName, Func<PerformanceCounter> performan
        {
            _seriesName = seriesName;
            _performanceCounterGenerator = performanceCounterGenerator;
            _subscriptions = new HashSet<IActorRef>();
            _cancelPublishing = new Cancelable(Context.System.Scheduler);
        }

        #region Actor lifecycle methods

        protected override void PreStart()
        {
            //create a new instance of the performance counter
            _counter = _performanceCounterGenerator();
            Context.System.Scheduler.ScheduleTellRepeatedly(TimeSpan.FromMilliseconds(250),
                new GatherMetrics(), Self, _cancelPublishing);
        }

        protected override void PostStop()
        {
            try
            {
                //terminate the scheduled task
                _cancelPublishing.Cancel(false);
                _counter.Dispose();
            }
            catch
            {
                //don't care about additional "ObjectDisposed" exceptions
            }
```

```
            finally
            {
                base.PostStop();
            }
        }

        #endregion

        protected override void OnReceive(object message)
        {
            if (message is GatherMetrics)
            {
                //publish latest counter value to all subscribers
                var metric = new Metric(_seriesName, _counter.NextValue());
                foreach(var sub in _subscriptions)
                    sub.Tell(metric);
            }
            else if (message is SubscribeCounter)
            {
                // add a subscription for this counter
                // (it's parent's job to filter by counter types)
                var sc = message as SubscribeCounter;
                _subscriptions.Add(sc.Subscriber);
            }
            else if (message is UnsubscribeCounter)
            {
                // remove a subscription from this counter
                var uc = message as UnsubscribeCounter;
                _subscriptions.Remove(uc.Subscriber);
            }
        }
    }
}
```

Before we move onto the next step, let's talk about what you just did...

Functional Programming for Reliability

Did you notice how, in the constructor of `PerformanceCounterActor`, we took a `Func<PerformanceCounter>` and NOT a `PerformanceCounter`? If you didn't, go back and look now. What gives?

This is a technique borrowed from functional programming. We use it whenever we have to inject an `IDisposable` object into the constructor of an actor. Why?

Well, we've got an actor that takes an `IDisposable` object as a parameter. So we're going to assume that this object will actually become `Disposed` at some point and will no longer

be available.

What happens when the `PerformanceCounterActor` needs to restart?

Every time the `PeformanceCounterActor` attempts to restart it will re-use its original constructor arguments, which includes reference types. If we re-use the same reference to the now-`Disposed` `PerformanceCounter`, the actor will crash repeatedly. Until its parent decides to just kill it altogether.

A better technique is to pass a factory function that `PerformanceCounterActor` can use to get a fresh instance of its `PeformanceCounter`. That's why we use a `Func<PerformanceCounter>` in the constructor, which gets invoked during the actor's `PreStart()` lifecycle method.

```
// create a new instance of the performance counter from factory that was passed in
_counter = _performanceCounterGenerator();
```

Because our `PeformanceCounter` is `IDisposable`, we also need to clean up the `PeformanceCounter` instance inside the `PostStop` lifecycle method of the actor.

We already know that we're going to get a fresh instance of that counter when the actor restarts, so we want to prevent resource leaks. This is how we do that:

```
// Actors/PerformanceCounterActor.cs
// prevent resource leaks by disposing of our current PeformanceCounter
protected override void PostStop()
{
    try
    {
        // terminate the scheduled task
        _cancelPublishing.Cancel(false);
        _counter.Dispose();
    }
    catch
    {
        // we don't care about additional "ObjectDisposed" exceptions
    }
    finally
    {
        base.PostStop();
    }
}
```

Pub / Sub Made Easy

The `PerformanceCounterActor` has pub / sub built into it by way of its handlers for `SubscribeCounter` and `UnsubscribeCounter` messages inside its `OnReceive` method:

```
// Actors/PerformanceCounterActor.cs
// ...
else if (message is SubscribeCounter)
{
    // add a subscription for this counter (it is up to the parent to filter by counter type
    var sc = message as SubscribeCounter;
    _subscriptions.Add(sc.Subscriber);
}
else if (message is UnsubscribeCounter)
{
    // remove a subscription from this counter
    var uc = message as UnsubscribeCounter;
    _subscriptions.Remove(uc.Subscriber);
}
```

In this lesson, `PerformanceCounterActor` only has one subscriber (`ChartingActor` , from inside `Main.cs`) but with a little re-architecting you could have these actors publishing their `PeformanceCounter` data to multiple recipients. Maybe that's a do-it-yourself exercise you can try later? ;)

How did we schedule publishing of `PeformanceCounter` data?

Inside the `PreStart` lifecycle method, we used the `Context` object to get access to the `Scheduler` , and then we had `PeformanceCounterActor` send itself a `GatherMetrics` method once every 250 milliseconds.

This causes `PeformanceCounterActor` to fetch data every 250ms and publish it to `ChartingActor` , giving us a live graph with a frame rate of 4 FPS.

```
// Actors/PerformanceCounterActor.cs
protected override void PreStart()
{
    // create a new instance of the performance counter
    _counter = _performanceCounterGenerator();
    Context.System.Scheduler.ScheduleTellRepeatedly(TimeSpan.FromMilliseconds(250), TimeSpan
        new GatherMetrics(), Self, _cancelPublishing);
```

```
}
```

Notice that inside the `PerformanceCounterActor`'s `PostStop` method, we invoke the `ICancelable` we created to cancel this recurring message:

```
 // terminate the scheduled task
_cancelPublishing.Cancel();
```

We do this for the same reason we `Dispose` the `PerformanceCounter` - to eliminate resource leaks and to prevent the `IScheduler` from sending recurring messages to dead or restarted actors.

Step 5 - Create the `PerformanceCounterCoordinatorActor`

The `PerformanceCounterCoordinatorActor` is the interface between the `ChartingActor` and all of the `PerformanceCounterActor` instances.

It has the following jobs:

- Lazily create all `PeformanceCounterActor` instances that are requested by the end-user;
- Provide the `PeformanceCounterActor` with a factory method (`Func<PerformanceCounter>`) for creating its counters;
- Manage all counter subscriptions for the `ChartingActor` ; and
- Tell the `ChartingActor` how to render each of the individual counter metrics (which colors and plot types to use for each `Series` that corresponds with a `PeformanceCounter` .)

Sounds complicated, right? Well, you'll be surprised when you see how small the code footprint is!

Create a new file in the `/Actors` folder called `PerformanceCounterCoordinatorActor.cs` and type the following:

```
// Actors/PerformanceCoordinatorActor.cs

using System;
```

```
using System.Collections.Generic;
using System.Diagnostics;
using System.Drawing;
using System.Windows.Forms.DataVisualization.Charting;
using Akka.Actor;

namespace ChartApp.Actors
{
    /// <summary>
    /// Actor responsible for translating UI calls into ActorSystem messages
    /// </summary>
    public class PerformanceCounterCoordinatorActor : ReceiveActor
    {
        #region Message types

        /// <summary>
        /// Subscribe the <see cref="ChartingActor"/> to updates for <see cref="Counter"/>.
        /// </summary>
        public class Watch
        {
            public Watch(CounterType counter)
            {
                Counter = counter;
            }

            public CounterType Counter { get; private set; }
        }

        /// <summary>
        /// Unsubscribe the <see cref="ChartingActor"/> to updates for <see cref="Counter"/>
        /// </summary>
        public class Unwatch
        {
            public Unwatch(CounterType counter)
            {
                Counter = counter;
            }

            public CounterType Counter { get; private set; }
        }

        #endregion

        /// <summary>
        /// Methods for generating new instances of all <see cref="PerformanceCounter"/>s we
        /// </summary>
        private static readonly Dictionary<CounterType, Func<PerformanceCounter>> CounterGen
            new Dictionary<CounterType, Func<PerformanceCounter>>()
        {
            {CounterType.Cpu, () => new PerformanceCounter("Processor", "% Processor Time",
            {CounterType.Memory, () => new PerformanceCounter("Memory", "% Committed Bytes I
            {CounterType.Disk, () => new PerformanceCounter("LogicalDisk", "% Disk Time", "_
        };

        /// <summary>
        /// Methods for creating new <see cref="Series"/> with distinct colors and names
```

```
        /// corresponding to each <see cref="PerformanceCounter"/>
        /// </summary>
        private static readonly Dictionary<CounterType, Func<Series>> CounterSeries =
            new Dictionary<CounterType, Func<Series>>()
        {
            {CounterType.Cpu, () =>
            new Series(CounterType.Cpu.ToString()){ ChartType = SeriesChartType.SplineArea,
             Color = Color.DarkGreen}},
            {CounterType.Memory, () =>
            new Series(CounterType.Memory.ToString()){ ChartType = SeriesChartType.FastLine,
            Color = Color.MediumBlue}},
            {CounterType.Disk, () =>
            new Series(CounterType.Disk.ToString()){ ChartType = SeriesChartType.SplineArea,
            Color = Color.DarkRed}},
        };

        private Dictionary<CounterType, IActorRef> _counterActors;

        private IActorRef _chartingActor;

        public PerformanceCounterCoordinatorActor(IActorRef chartingActor) :
            this(chartingActor, new Dictionary<CounterType, IActorRef>())
        {
        }

        public PerformanceCounterCoordinatorActor(IActorRef chartingActor, Dictionary<Counte
        {
            _chartingActor = chartingActor;
            _counterActors = counterActors;

            Receive<Watch>(watch =>
            {
                if (!_counterActors.ContainsKey(watch.Counter))
                {
                    // create a child actor to monitor this counter if one doesn't exist alr
                    var counterActor = Context.ActorOf(Props.Create(() =>
                        new PerformanceCounterActor(watch.Counter.ToString(), CounterGenerat

                    // add this counter actor to our index
                    _counterActors[watch.Counter] = counterActor;
                }

                // register this series with the ChartingActor
                _chartingActor.Tell(new ChartingActor.AddSeries(CounterSeries[watch.Counter]

                // tell the counter actor to begin publishing its statistics to the _chartin
                _counterActors[watch.Counter].Tell(new SubscribeCounter(watch.Counter, _char
            });

            Receive<Unwatch>(unwatch =>
            {
                if (!_counterActors.ContainsKey(unwatch.Counter))
                {
                    return; // noop
                }
```

```
                // unsubscribe the ChartingActor from receiving anymore updates
                _counterActors[unwatch.Counter].Tell(new UnsubscribeCounter(unwatch.Counter,

                // remove this series from the ChartingActor
                _chartingActor.Tell(new ChartingActor.RemoveSeries(unwatch.Counter.ToString(
            });
        }


    }
}
```

Okay, we're almost there. Just one more actor to go!

Step 6 - Create the `ButtonToggleActor`

You didn't think we were going to let you just fire off those buttons you created in Step 2 without adding some actors to manage them, did you? ;)

In this step, we're going to add a new type of actor that will run on the UI thread just like the `ChartingActor`.

The job of the `ButtonToggleActor` is to turn click events on the `Button` it manages into messages for the `PerformanceCounterCoordinatorActor`. The `ButtonToggleActor` also makes sure that the visual state of the `Button` accurately reflects the state of the subscription managed by the `PeformanceCounterCoordinatorActor` (e.g. ON/OFF).

Okay, create a new file in the `/Actors` folder called `ButtonToggleActor.cs` and type the following:

```
// Actors/ButtonToggleActor.cs

using System.Windows.Forms;
using Akka.Actor;

namespace ChartApp.Actors
{
    /// <summary>
    /// Actor responsible for managing button toggles
    /// </summary>
    public class ButtonToggleActor : UntypedActor
    {
        #region Message types
```

```
        /// <summary>
        /// Toggles this button on or off and sends an appropriate messages
        /// to the <see cref="PerformanceCounterCoordinatorActor"/>
        /// </summary>
        public class Toggle { }

        #endregion

        private readonly CounterType _myCounterType;
        private bool _isToggledOn;
        private readonly Button _myButton;
        private readonly IActorRef _coordinatorActor;

        public ButtonToggleActor(IActorRef coordinatorActor, Button myButton,
                CounterType myCounterType, bool isToggledOn = false)
        {
            _coordinatorActor = coordinatorActor;
            _myButton = myButton;
            _isToggledOn = isToggledOn;
            _myCounterType = myCounterType;
        }

        protected override void OnReceive(object message)
        {
            if (message is Toggle && _isToggledOn)
            {
                // toggle is currently on

                // stop watching this counter
                _coordinatorActor.Tell(new PerformanceCounterCoordinatorActor.Unwatch(_myCou

                FlipToggle();
            }
            else if (message is Toggle && !_isToggledOn)
            {
                // toggle is currently off

                // start watching this counter
                _coordinatorActor.Tell(new PerformanceCounterCoordinatorActor.Watch(_myCount

                FlipToggle();
            }
            else
            {
                Unhandled(message);
            }
        }

        private void FlipToggle()
        {
            // flip the toggle
            _isToggledOn = !_isToggledOn;

            // change the text of the button
            _myButton.Text = string.Format("{0} ({1})", _myCounterType.ToString().ToUpperInv
                _isToggledOn ? "ON" : "OFF");
```

```
        }
    }
}
```

Step 7 - Update the `ChartingActor`

Home stretch! We're almost there.

We need to integrate all of the new message types we defined in Step 3 into the `ChartingActor`. We also need to make some changes to the way we render the `Chart` since we're going to be making *live updates* to it continuously.

To start, add this code at the very top of the `ChartingActor` class:

```
// Actors/ChartingActor.cs

/// <summary>
/// Maximum number of points we will allow in a series
/// </summary>
public const int MaxPoints = 250;

/// <summary>
/// Incrementing counter we use to plot along the X-axis
/// </summary>
private int xPosCounter = 0;
```

Next, add a new message type that the `ChartingActor` is going to use. Add this inside the `Messages` region of `Actors/ChartingActor.cs`:

```
// Actors/ChartingActor.cs - inside the Messages region

/// <summary>
/// Remove an existing <see cref="Series"/> from the chart
/// </summary>
public class RemoveSeries
{
    public RemoveSeries(string seriesName)
    {
        SeriesName = seriesName;
    }

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

Add the following method to the bottom of the `ChartingActor` class (don't worry about the specifics, it's adding UI management code that isn't directly related to actors):

```
// Actors/ChartingActor.cs

private void SetChartBoundaries()
{
    double maxAxisX, maxAxisY, minAxisX, minAxisY = 0.0d;
    var allPoints = _seriesIndex.Values.SelectMany(series => series.Points).ToList();
    var yValues = allPoints.SelectMany(point => point.YValues).ToList();
    maxAxisX = xPosCounter;
    minAxisX = xPosCounter - MaxPoints;
    maxAxisY = yValues.Count > 0 ? Math.Ceiling(yValues.Max()) : 1.0d;
    minAxisY = yValues.Count > 0 ? Math.Floor(yValues.Min()) : 0.0d;
    if (allPoints.Count > 2)
    {
        var area = _chart.ChartAreas[0];
        area.AxisX.Minimum = minAxisX;
        area.AxisX.Maximum = maxAxisX;
        area.AxisY.Minimum = minAxisY;
        area.AxisY.Maximum = maxAxisY;
    }
}
```

> **NOTE**: the `SetChartBoundaries()` method is used to make sure that the boundary area of our chart gets updated as we remove old points from the beginning of the chart as time elapses.

Next, we're going to redefine all of our message handlers to use the new `SetChartBoundaries()` method.

Delete everything inside the previous `Individual Message Type Handlers` region, and then **replace it with the following**:

```
// Actors/ChartingActor.cs - inside the Individual Message Type Handlers region
private void HandleInitialize(InitializeChart ic)
{
    if (ic.InitialSeries != null)
    {
        // swap the two series out
        _seriesIndex = ic.InitialSeries;
    }

    // delete any existing series
    _chart.Series.Clear();
```

```
        // set the axes up
        var area = _chart.ChartAreas[0];
        area.AxisX.IntervalType = DateTimeIntervalType.Number;
        area.AxisY.IntervalType = DateTimeIntervalType.Number;

        SetChartBoundaries();

        // attempt to render the initial chart
        if (_seriesIndex.Any())
        {
            foreach (var series in _seriesIndex)
            {
                // force both the chart and the internal index to use the same names
                series.Value.Name = series.Key;
                _chart.Series.Add(series.Value);
            }
        }

        SetChartBoundaries();
    }

    private void HandleAddSeries(AddSeries series)
    {
        if(!string.IsNullOrEmpty(series.Series.Name) && !_seriesIndex.ContainsKey(series.Series.
        {
            _seriesIndex.Add(series.Series.Name, series.Series);
            _chart.Series.Add(series.Series);
            SetChartBoundaries();
        }
    }

    private void HandleRemoveSeries(RemoveSeries series)
    {
        if (!string.IsNullOrEmpty(series.SeriesName) && _seriesIndex.ContainsKey(series.SeriesNa
        {
            var seriesToRemove = _seriesIndex[series.SeriesName];
            _seriesIndex.Remove(series.SeriesName);
            _chart.Series.Remove(seriesToRemove);
            SetChartBoundaries();
        }
    }

    private void HandleMetrics(Metric metric)
    {
        if (!string.IsNullOrEmpty(metric.Series) && _seriesIndex.ContainsKey(metric.Series))
        {
            var series = _seriesIndex[metric.Series];
            series.Points.AddXY(xPosCounter++, metric.CounterValue);
            while(series.Points.Count > MaxPoints) series.Points.RemoveAt(0);
            SetChartBoundaries();
        }
    }
```

And finally, add these `Receive<T>` handlers to the constructor for `ChartingActor`:

```
// Actors/ChartingActor.cs - add these below the original Receive<T> handlers in the constru
Receive<RemoveSeries>(removeSeries => HandleRemoveSeries(removeSeries));
Receive<Metric>(metric => HandleMetrics(metric));
```

Step 8 - Replace the `Main_Load` Handler in `Main.cs`

Now that we have real data we want to plot in real-time, we need to replace the original `Main_Load` event handler, which supplied fake data to our `ChartActor` with a real one that sets us up for live charting.

Add the following declarations to the top of the `Main` class inside `Main.cs`:

```
// Main.cs - at top of Main class
private IActorRef _coordinatorActor;
private Dictionary<CounterType, IActorRef> _toggleActors = new Dictionary<CounterType, IActo
```

Then, replace the `Main_Load` event handler in the `Init` region so that it matches this:

```
// Main.cs - replace Main_Load event handler in the Init region
private void Main_Load(object sender, EventArgs e)
{
    _chartActor = Program.ChartActors.ActorOf(Props.Create(() => new ChartingActor(sysChart)
    _chartActor.Tell(new ChartingActor.InitializeChart(null)); //no initial series

    _coordinatorActor = Program.ChartActors.ActorOf(Props.Create(() =>
            new PerformanceCounterCoordinatorActor(_chartActor)), "counters");

    // CPU button toggle actor
    _toggleActors[CounterType.Cpu] = Program.ChartActors.ActorOf(
        Props.Create(() => new ButtonToggleActor(_coordinatorActor, btnCpu, CounterType.Cpu,
            .WithDispatcher("akka.actor.synchronized-dispatcher"));

    // MEMORY button toggle actor
    _toggleActors[CounterType.Memory] = Program.ChartActors.ActorOf(
       Props.Create(() => new ButtonToggleActor(_coordinatorActor, btnMemory, CounterType.Me
           .WithDispatcher("akka.actor.synchronized-dispatcher"));

    // DISK button toggle actor
    _toggleActors[CounterType.Disk] = Program.ChartActors.ActorOf(
       Props.Create(() => new ButtonToggleActor(_coordinatorActor, btnDisk, CounterType.Disk
```

```
            .WithDispatcher("akka.actor.synchronized-dispatcher"));

        // Set the CPU toggle to ON so we start getting some data
        _toggleActors[CounterType.Cpu].Tell(new ButtonToggleActor.Toggle());
    }
```

Wait a minute, what's this `WithDispatcher` nonsense?!

`Props` has a built-in fluent interface which allows you to configure your actor deployments programmatically, and in this instance we decided to use the `Props.WithDispatcher` method to guarantee that each of the `ButtonToggleActor` instances run on the UI thread.

As we saw in Lesson 2.1, you can also configure the `Dispatcher` for an actor via the HOCON config. So if an actor has a `Dispatcher` set in HOCON, *and* one declared programmatically via the `Props` fluent interface, which wins?

In case of a conflict, `Config` *wins and* `Props` *loses .* Any conflicting settings declared by the `Props` fluent interface will always be overriden by what was declared in configuration.

Step 9 - Have Button Handlers Send `Toggle` Messages to Corresponding `ButtonToggleActor`

THE LAST STEP. We promise :) Thanks for hanging in there.

Finally, we need to wire up the button handlers we created in Step 3.

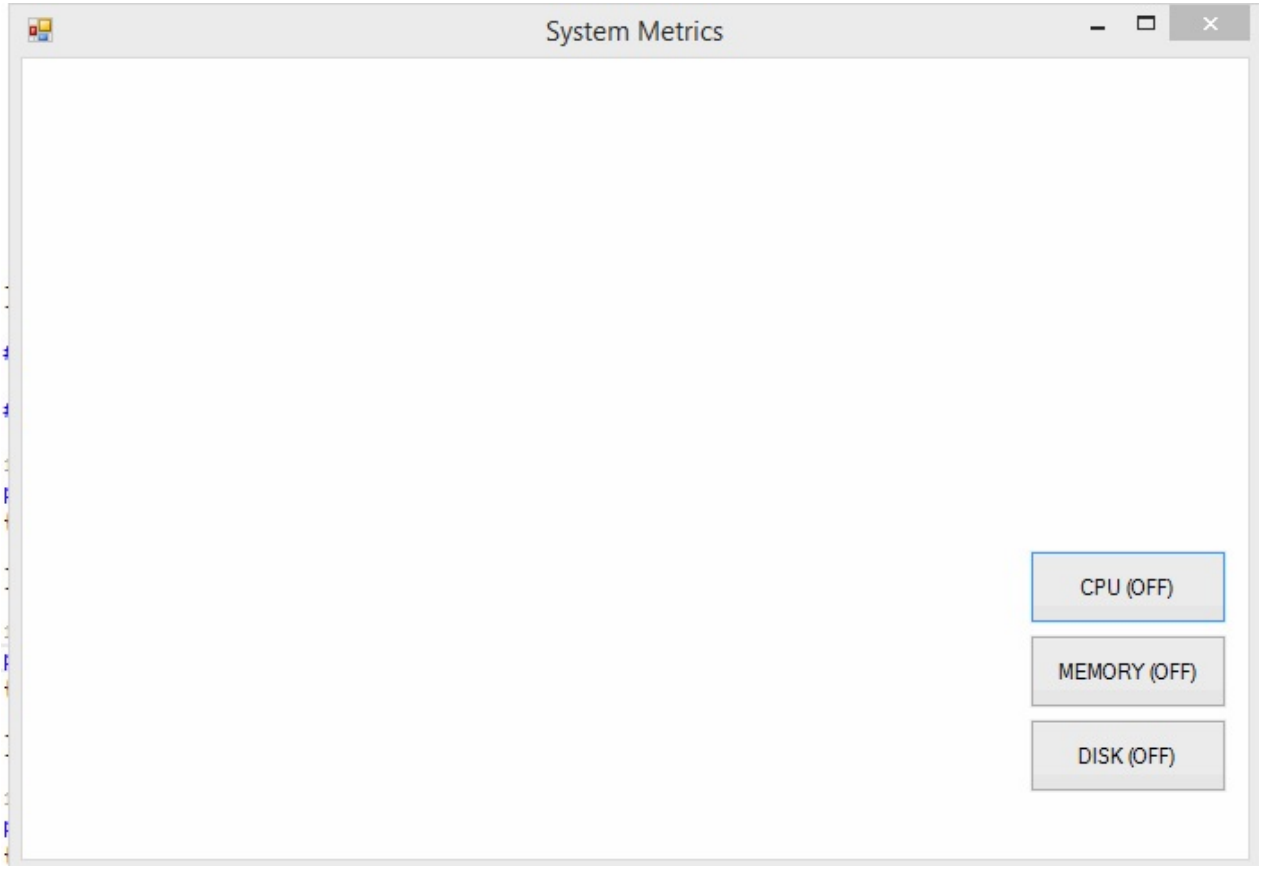
Wire up your button handlers in `Main.cs`. They should look like this:

```
// Main.cs - wiring up the button handlers added in step 3
private void btnCpu_Click(object sender, EventArgs e)
{
    _toggleActors[CounterType.Cpu].Tell(new ButtonToggleActor.Toggle());
}

private void btnMemory_Click(object sender, EventArgs e)
{
    _toggleActors[CounterType.Memory].Tell(new ButtonToggleActor.Toggle());
}

private void btnDisk_Click(object sender, EventArgs e)
{
```

```
        _toggleActors[CounterType.Disk].Tell(new ButtonToggleActor.Toggle());
    }
```

Once you're done

Build and run `SystemCharting.sln` and you should see the following:



Compare your code to the code in the /Completed/ folder to compare your final output to what the instructors produced.

Great job!

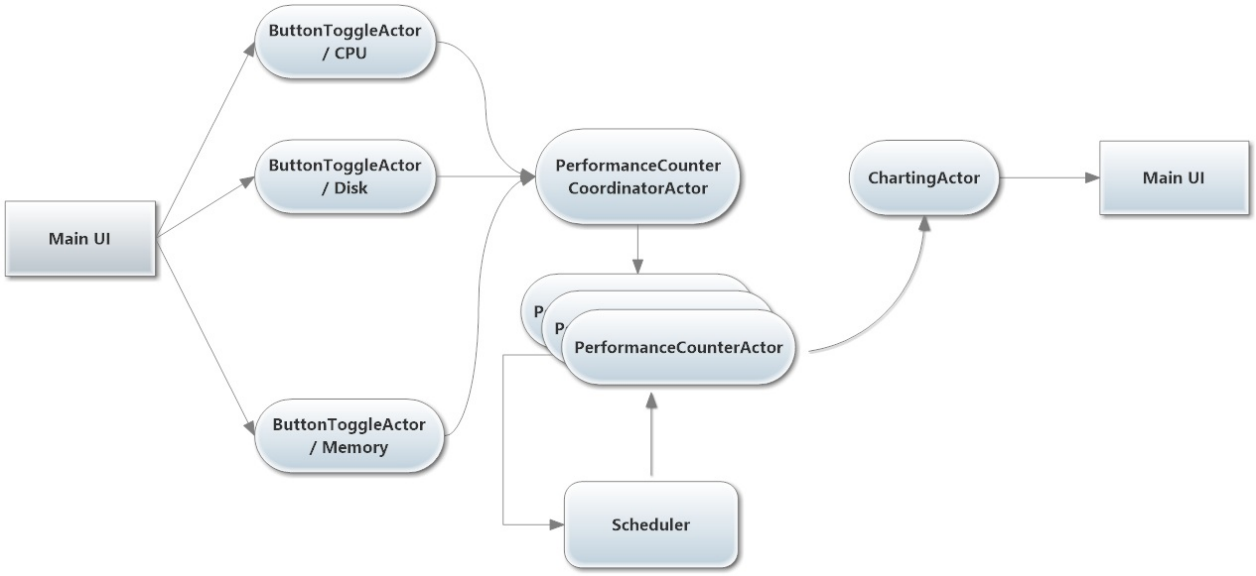
Wow. That was *a lot* of code. Great job and thanks for sticking it out! We now have a fully functioning Resource Monitor app, implemented with actors.

Every other lesson builds on this one, so before continuing, please make sure your code matches the output of the /Completed/ folder.

At this point. you should understand how the `Scheduler` works and how you can use it alongside patterns like Pub-sub to make very reactive systems with actors that have a

comparatively small code footprint.

Here is a high-level overview of our working system at this point:



Let's move onto Lesson 4 - Switching Actor Behavior at Run-time with `Become` and `Unbecome`.

Any questions?

Don't be afraid to ask questions :).

Come ask any questions you have, big or small, in this ongoing Bootcamp chat with the Petabridge & Akka.NET teams.

Problems with the code?

If there is a problem with the code running, or something else that needs to be fixed in this lesson, please create an issue and we'll get right on it. This will benefit everyone going through Bootcamp.

Lesson 2.4: Switching Actor Behavior at Run-time with `BecomeStacked` and `UnbecomeStacked`

In this lesson we're going to learn about one of the really cool things actors can do: change their behavior at run-time!

Key Concepts / Background

Let's start with a real-world scenario in which you'd want the ability to change an actor's behavior.

Real-World Scenario: Authentication

Imagine you're building a simple chat system using Akka.NET actors, and here's what your `UserActor` looks like - this is the actor that is responsible for all communication to and from a specific human user.

```
public class UserActor : ReceiveActor {
    private readonly string _userId;
    private readonly string _chatRoomId;

    public UserActor(string userId, string chatRoomId) {
        _userId = userId;
        _chatRoomId = chatRoomId;
        Receive<IncomingMessage>(inc => inc.ChatRoomId == _chatRoomId,
            inc => {
                // print message for user
            });
        Receive<OutgoingMessage>(inc => inc.ChatRoomId == _chatRoomId,
            inc => {
                // send message to chatroom
            });
    }
}
```

So we have basic chat working - yay! But… right now there's nothing to guarantee that this user is who they say they are. This system needs some authentication.

How could we rewrite this actor to handle these same types of chat messages differently when:

- The user is **authenticating**
- The user is **authenticated**, or
- The user **couldn't authenticate**?

Simple: we can use switchable actor behaviors to do this!

What is switchable behavior?

One of the core attributes of an actor in the Actor Model is that an actor can change its behavior between messages that it processes.

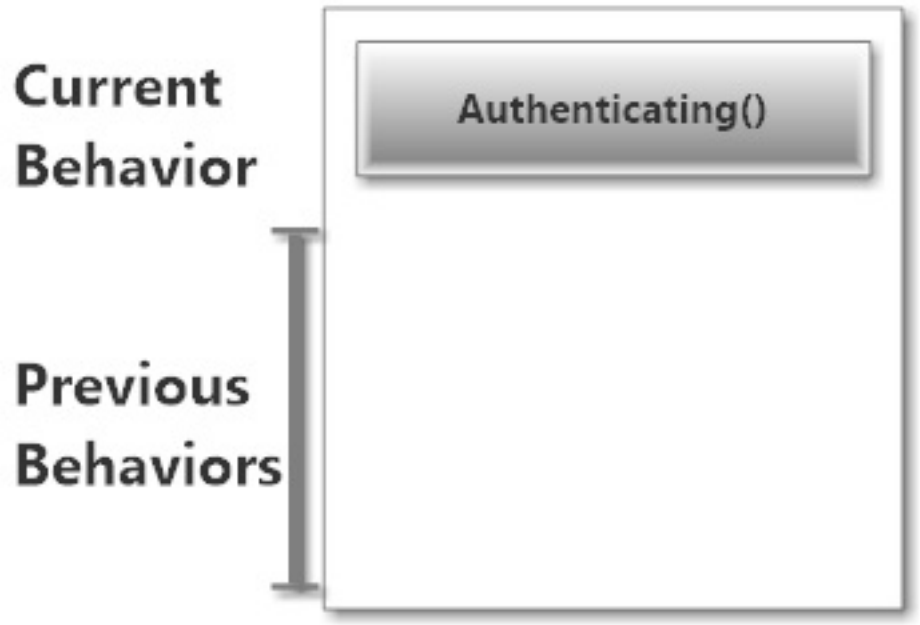
This capability allows you to do all sorts of cool stuff, like build Finite State Machines or change how your actors handle messages based on other messages they've received.

Switchable behavior is one of the most powerful and fundamental capabilities of any true actor system. It's one of the key features enabling actor reusability, and helping you to do a massive amount of work with a very small code footprint.

How does switchable behavior work?

The Behavior Stack

Akka.NET actors have the concept of a "behavior stack". Whichever method sits at the top of the behavior stack defines the actor's current behavior. Currently, that behavior is `Authenticating()`:



Use `Become` and `BecomeStacked` to adopt new behavior

Whenever we call `BecomeStacked`, we tell the `ReceiveActor` to push a new behavior onto the stack. This new behavior dictates which `Receive` methods will be used to process any messages delivered to an actor.

Here's what happens to the behavior stack when our example actor becomes `Authenticated` via `BecomeStacked`:

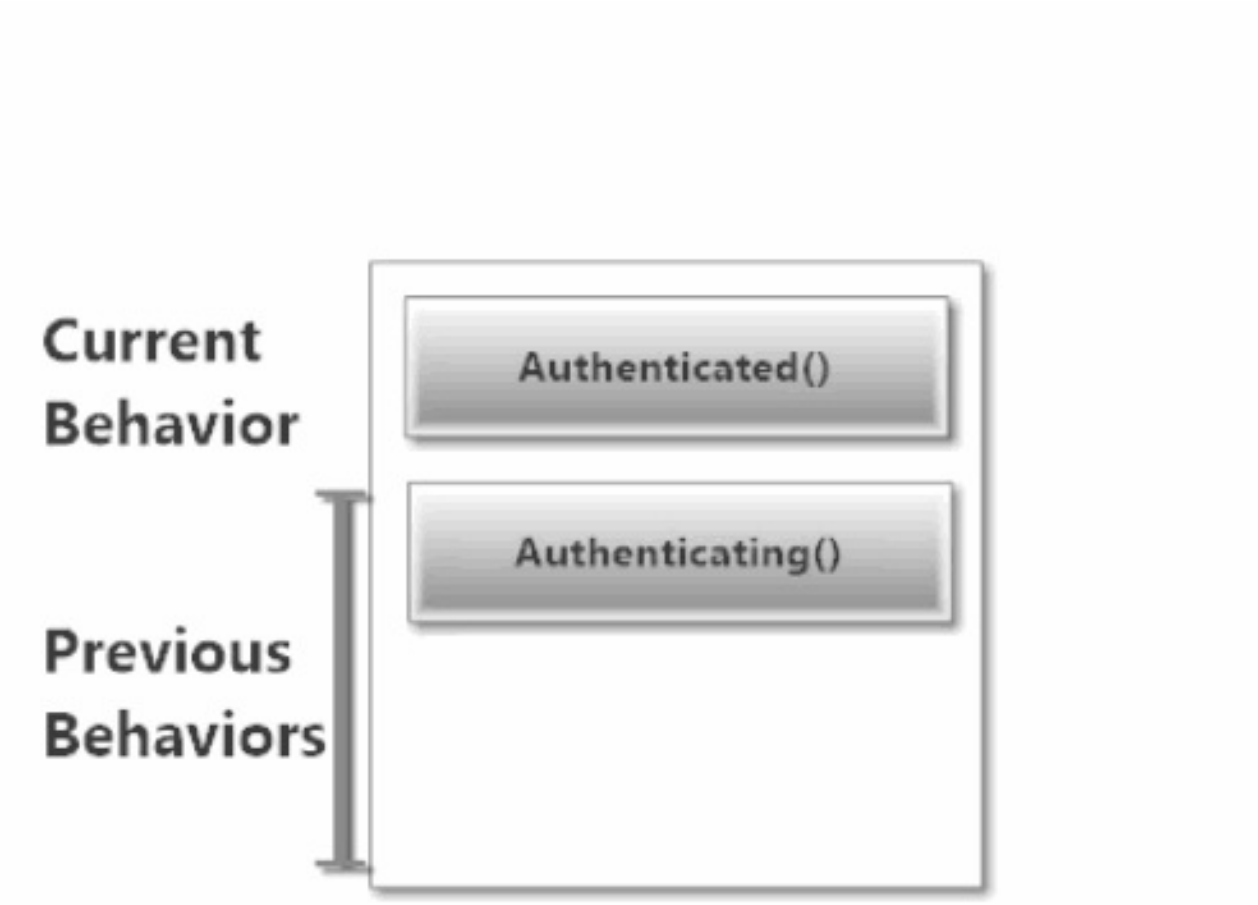
> NOTE: `Become` will delete the old behavior off of the stack - so the stack will never have more than one behavior in it at a time.
>
> Use `BecomeStacked` if you want to push behavior onto the stack, and `UnbecomeStacked` if you want to revert to a previous behavior. Most users only ever need to use `Become`.

Use `UnbecomeStacked` to revert to old behavior

To make an actor revert to the previous behavior in the behavior stack, all we have to do is call `UnbecomeStacked`.

Whenever we call `UnbecomeStacked`, we pop our current behavior off of the stack and replace it with the previous behavior from before (again, this new behavior will dictate which `Receive` methods are used to handle incoming messages).

Here's what happens to the behavior stack when our example actor `UnbecomeStacked`s:



What is the API to change behaviors?

The API to change behaviors is very simple:

- `Become` - Replaces the current receive loop with the specified one. Eliminates the behavior stack.
- `BecomeStacked` - Adds the specified method to the top of the behavior stack, while maintaining the previous ones below it;
- `UnbecomeStacked` - Reverts to the previous receive method from the stack (only works with `BecomeStacked`).

The difference is that `BecomeStacked` preserves the old behavior, so you can just call `UnbecomeStacked` to go back to the previous behavior. The preference of one over the other depends on your needs. You can call `BecomeStacked` as many times as you need, and you can call `UnbecomeStacked` as many times as you called `BecomeStacked`. Additional calls to `UnbecomeStacked` won't do anything if the current behavior is the only behavior in the stack.

Isn't it problematic for actors to change behaviors?

No, actually it's safe and is a feature that gives your `ActorSystem` a ton of flexibility and code reuse.

Here are some common questions about switchable behavior:

When is the new behavior applied?

We can safely switch actor message-processing behavior because Akka.NET actors only process one message at a time. The new message processing behavior won't be applied until the next message arrives.

Isn't it bad that `Become` blows away the behavior stack?

No, not really. This is the way it's most commonly used, by far. Explicitly switching from one behavior to another is the most common approach used for switching behavior. Simple, explicit switches also make it much easier to read and reason about your code.

If you find you actually need to take advantage of the behavior stack—and a simple, explicit `Become(YourNewBehavior)` won't work for the situation—the behavior stack is available to you.

In this lesson, we use `BecomeStacked` and `UnbecomeStacked` to demonstrate them. Usually we just use `Become`.

How deep can the behavior stack go?

The stack can go *really* deep, but it's not unlimited.

Also, each time your actor restarts, the behavior stack is cleared and the actor starts from the initial behavior you've coded.

What happens if you call `UnbecomeStacked` and with nothing left in the behavior stack?

Nothing - `UnbecomeStacked` is a safe method and won't do anything if the current behavior is the only behavior in the stack.

Back to the real-world example

Okay, now that you understand switchable behavior, let's return to our real-world scenario and see how it is used. Recall that we need to add authentication to our chat system actor.

So, how could we rewrite this actor to handle chat messages differently when:

- The user is **authenticating**
- The user is **authenticated**, or
- The user **couldn't authenticate**?

Here's one way we can implement switchable message behavior in our `UserActor` to handle basic authentication:

```
public class UserActor : ReceiveActor {
    private readonly string _userId;
    private readonly string _chatRoomId;

    public UserActor(string userId, string chatRoomId) {
        _userId = userId;
        _chatRoomId = chatRoomId;

        // start with the Authenticating behavior
        Authenticating();
    }

    protected override void PreStart() {
        // start the authentication process for this user
        Context.ActorSelection("/user/authenticator/")
            .Tell(new AuthenticatePlease(_userId));
    }

    private void Authenticating() {
        Receive<AuthenticationSuccess>(auth => {
            Become(Authenticated); //switch behavior to Authenticated
        });
        Receive<AuthenticationFailure>(auth => {
            Become(Unauthenticated); //switch behavior to Unauthenticated
        });
        Receive<IncomingMessage>(inc => inc.ChatRoomId == _chatRoomId,
            inc => {
                // can't accept message yet - not auth'd
            });
        Receive<OutgoingMessage>(inc => inc.ChatRoomId == _chatRoomId,
            inc => {
                // can't send message yet - not auth'd
            });
    }

    private void Unauthenticated() {
```

```
            //switch to Authenticating
            Receive<RetryAuthentication>(retry => Become(Authenticating));
            Receive<IncomingMessage>(inc => inc.ChatRoomId == _chatRoomId,
                inc => {
                    // have to reject message - auth failed
                });
            Receive<OutgoingMessage>(inc => inc.ChatRoomId == _chatRoomId,
                inc => {
                    // have to reject message - auth failed
                });
        }

        private void Authenticated() {
            Receive<IncomingMessage>(inc => inc.ChatRoomId == _chatRoomId,
                inc => {
                    // print message for user
                });
            Receive<OutgoingMessage>(inc => inc.ChatRoomId == _chatRoomId,
                inc => {
                    // send message to chatroom
                });
        }
    }
```

Whoa! What's all this stuff? Let's review it.

First, we took the `Receive<T>` handlers defined on our `ReceiveActor` and moved them into three separate methods. Each of these methods represents a state that will control how the actor processes messages:

- `Authenticating()` : this behavior is used to process messages when the user is attempting to authenticate (initial behavior).
- `Authenticated()` : this behavior is used to process messages when the authentication operation is successful; and,
- `Unauthenticated()` : this behavior is used to process messages when the authentication operation fails.

We called `Authenticating()` from the constructor, so our actor began in the `Authenticating()` state.

This means that only the `Receive<T>` handlers defined in the `Authenticating()` method will be used to process messages (initially).

However, if we receive a message of type `AuthenticationSuccess` or `AuthenticationFailure`, we use the `Become` method (docs) to switch behaviors to either `Authenticated` or

`Unauthenticated`, respectively.

Can I switch behaviors in an `UntypedActor`?

Yes, but the syntax is a little different inside an `UntypedActor`. To switch behaviors in an `UntypedActor`, you have to access `BecomeStacked` and `UnbecomeStacked` via the `ActorContext`, instead of calling them directly.

These are the API calls inside an `UntypedActor`:

- `Context.Become(Receive rec)` - changes behavior without preservering the previous behavior on the stack;
- `Context.BecomeStacked(Receive rec)` - pushes a new behavior on the stack or
- `Context.UnbecomeStacked()` - pops the current behavior and switches to the previous (if applicable.)

The first argument to `Context.Become` is a `Receive` delegate, which is really any method with the following signature:

```
void MethodName(object someParameterName);
```

This delegate is just used to represent another method in the actor that receives a message and represents the new behavior state.

Here's an example (`OtherBehavior` is the `Receive` delegate):

```
public class MyActor : UntypedActor {
    protected override void OnReceive(object message) {
        if(message is SwitchMe) {
            // preserve the previous behavior on the stack
            Context.BecomeStacked(OtherBehavior);
        }
    }

    // OtherBehavior is a Receive delegate
    private void OtherBehavior(object message) {
        if(message is SwitchMeBack) {
            // switch back to previous behavior on the stack
            Context.UnbecomeStacked();
        }
    }
}
```

Aside from those syntactical differences, behavior switching works exactly the same way across both `UntypedActor` and `ReceiveActor`.

Now, let's put behavior switching to work for us!

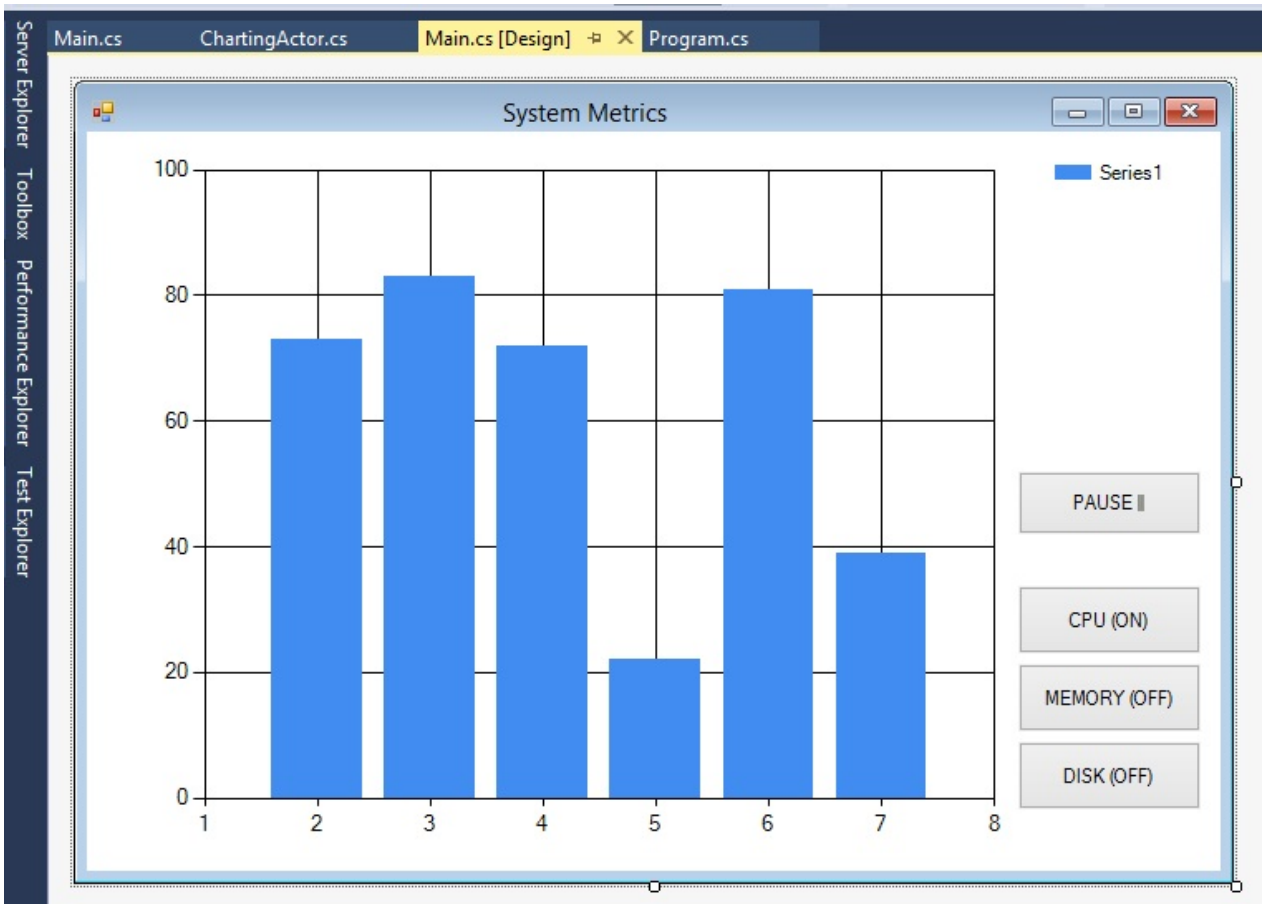
Exercise

In this lesson we're going to add the ability to pause and resume live updates to the `ChartingActor` via switchable actor behaviors.

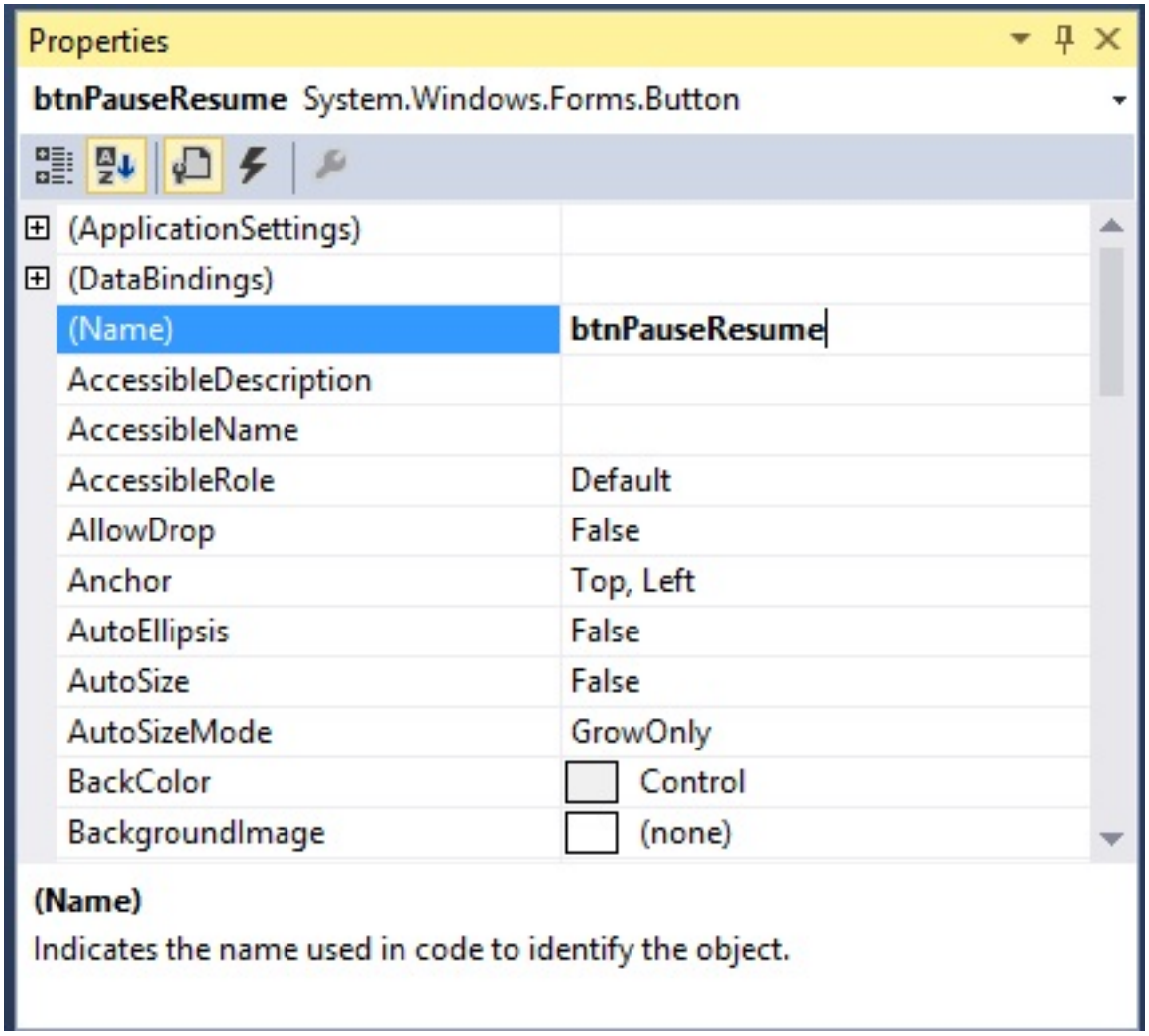
Phase 1 - Add a New `Pause / Resume` Button to `Main.cs`

This is the last button you'll have to add, we promise.

Go to the **[Design]** view of `Main.cs` and add a new button with the following text: `PAUSE ||`



Got to the **Properties** window in Visual Studio and change the name of this button to `btnPauseResume`.



Double click on the `btnPauseResume` to add a click handler to `Main.cs`.

```
private void btnPauseResume_Click(object sender, EventArgs e)
{

}
```

We'll fill this click handler in shortly.

Phase 2 - Add Switchable Behavior to `ChartingActor`

We're going to add some dynamic behavior to the `ChartingActor` - but first we need to do a little cleanup.

First, add a `using` reference for the Windows Forms namespace at the top of `Actors/ChartingActor.cs`.

```
// Actors/ChartingActor.cs

using System.Windows.Forms;
```

Next we need to declare a new message type inside the `Messages` region of `ChartingActor`.

```
// Actors/ChartingActor.cs - add inside the Messages region
/// <summary>
/// Toggles the pausing between charts
/// </summary>
public class TogglePause { }
```

Next, add the following field declaration just above the `ChartingActor` constructor declarations:

```
// Actors/ChartingActor.cs - just above ChartingActor's constructors

private readonly Button _pauseButton;
```

Move all of the `Receive<T>` declarations from `ChartingActor`'s main constructor into a new method called `Charting()`.

```
// Actors/ChartingActor.cs - just after ChartingActor's constructors
private void Charting()
{
    Receive<InitializeChart>(ic => HandleInitialize(ic));
    Receive<AddSeries>(addSeries => HandleAddSeries(addSeries));
    Receive<RemoveSeries>(removeSeries => HandleRemoveSeries(removeSeries));
    Receive<Metric>(metric => HandleMetrics(metric));

    //new receive handler for the TogglePause message type
    Receive<TogglePause>(pause =>
    {
        SetPauseButtonText(true);
        BecomeStacked(Paused);
    });
}
```

Add a new method called `HandleMetricsPaused` to the `ChartingActor`'s `Individual Message Type Handlers` region.

```
// Actors/ChartingActor.cs - inside Individual Message Type Handlers region
private void HandleMetricsPaused(Metric metric)
{
    if (!string.IsNullOrEmpty(metric.Series) && _seriesIndex.ContainsKey(metric.Series))
    {
        var series = _seriesIndex[metric.Series];
        series.Points.AddXY(xPosCounter++, 0.0d); //set the Y value to zero when we're pause
        while (series.Points.Count > MaxPoints) series.Points.RemoveAt(0);
        SetChartBoundaries();
    }
}
```

Define a new method called `SetPauseButtonText` at the *very* bottom of the `ChartingActor` class:

```
// Actors/ChartingActor.cs - add to the very bottom of the ChartingActor class
private void SetPauseButtonText(bool paused)
    {
        _pauseButton.Text = string.Format("{0}", !paused ? "PAUSE ||" : "RESUME ->");
    }
```

Add a new method called `Paused` just after the `Charting` method inside `ChartingActor`:

```
// Actors/ChartingActor.cs - just after the Charting method
private void Paused()
{
    Receive<Metric>(metric => HandleMetricsPaused(metric));
    Receive<TogglePause>(pause =>
    {
        SetPauseButtonText(false);
        UnbecomeStacked();
    });
}
```

And finally, let's **replace both of `ChartingActor`'s constructors**:

```
public ChartingActor(Chart chart, Button pauseButton) : this(chart, new Dictionary<string, S
```

```
{
}

public ChartingActor(Chart chart, Dictionary<string, Series> seriesIndex, Button pauseButton
{
    _chart = chart;
    _seriesIndex = seriesIndex;
    _pauseButton = pauseButton;
    Charting();
}
```

Phase 3 - Update the `Main_Load` and `Pause / Resume` Click Handler in Main.cs

Since we changed the constructor arguments for `ChartingActor` in Phase 2, we need to fix this inside our `Main_Load` event handler.

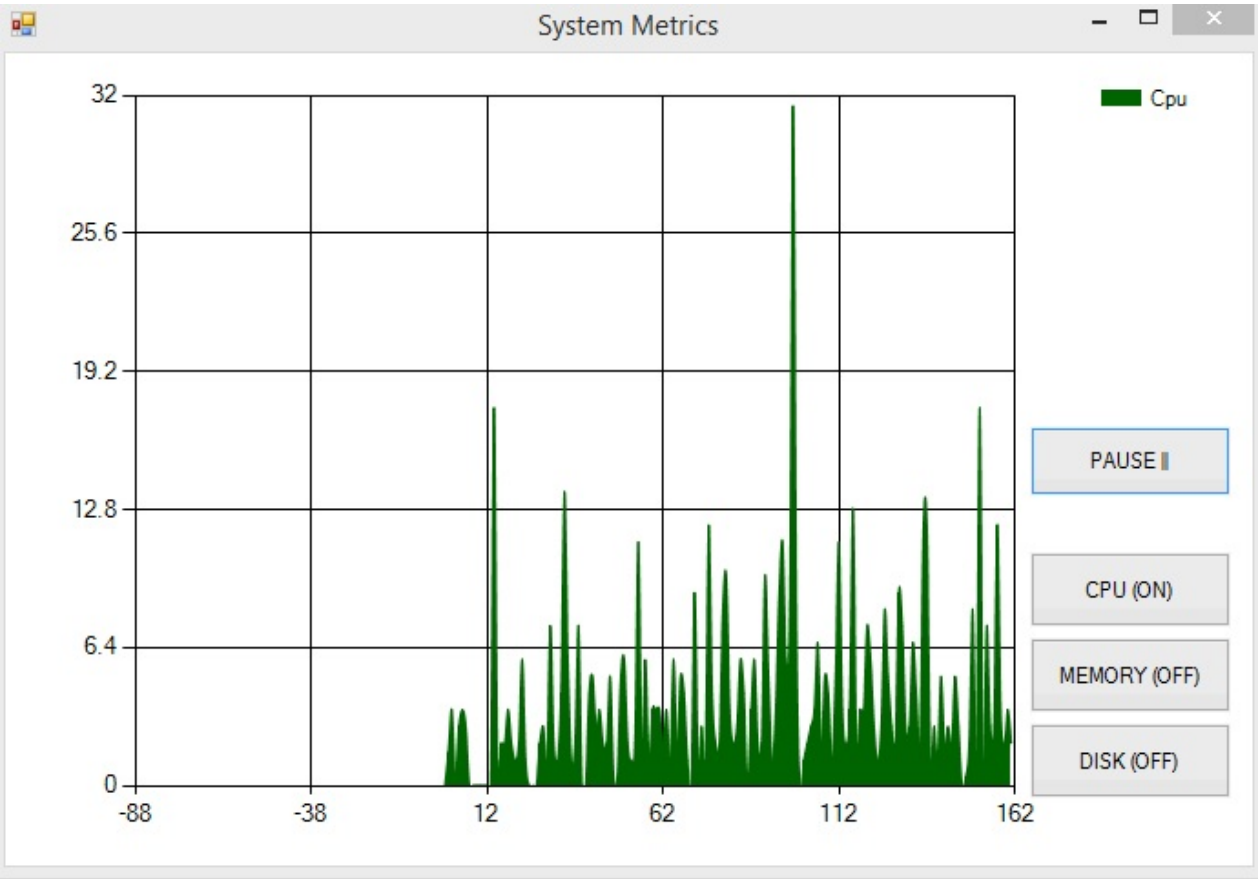
```
//Main.cs - Main_Load event handler
_chartActor = Program.ChartActors.ActorOf(Props.Create(() => new ChartingActor(sysChart, btn
```

And finally, we need to update our `btnPauseResume` click event handler to have it tell the `ChartingActor` to pause or resume live updates:

```
//Main.cs - btnPauseResume click handler
private void btnPauseResume_Click(object sender, EventArgs e)
{
    _chartActor.Tell(new ChartingActor.TogglePause());
}
```

Once you're done

Build and run `SystemCharting.sln` and you should see the following:

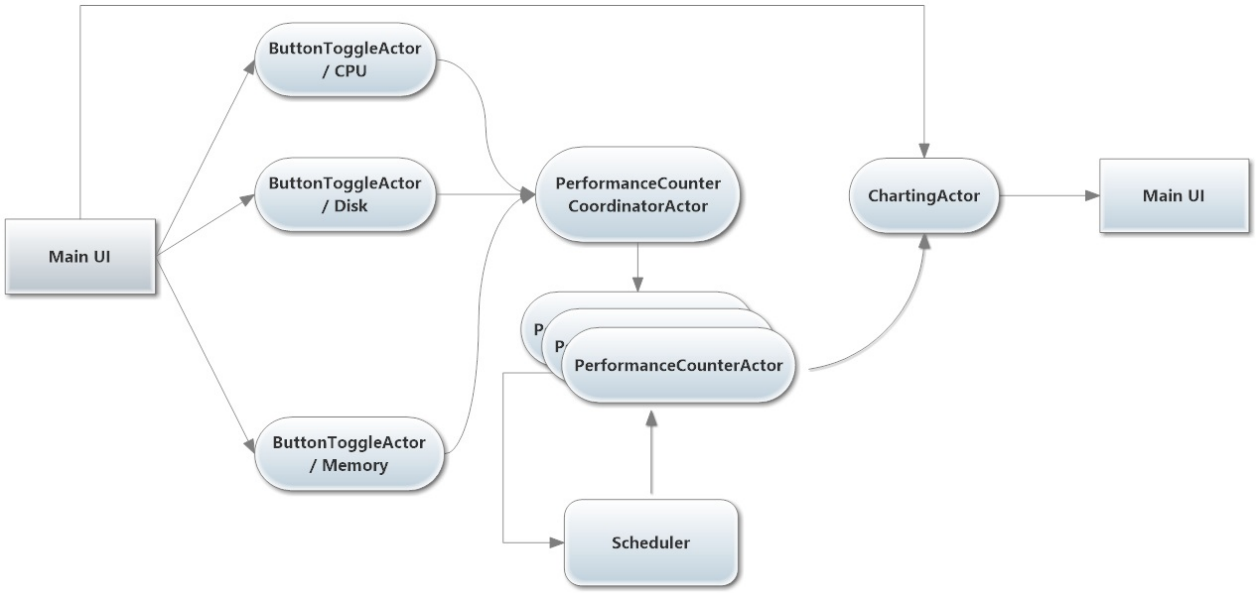


Compare your code to the code in the /Completed/ folder to compare your final output to what the instructors produced.

Great job!

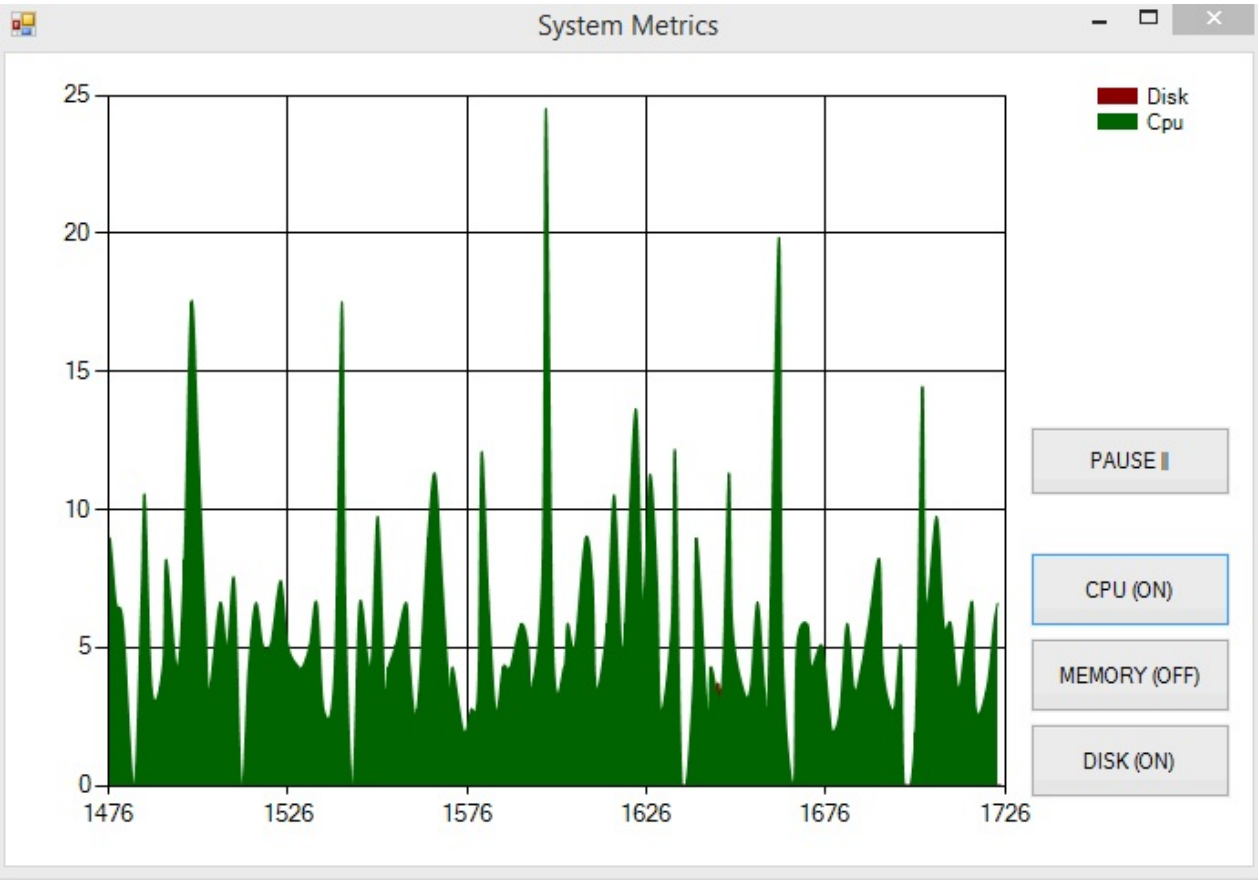
YEAAAAAAAAAAAAH! We have a live updating chart that we can pause over time!

Here is a high-level overview of our working system at this point:



But wait a minute!

What happens if I toggle a chart on or off when the `ChartingActor` is in a paused state?



DOH!!!!!! It doesn't work!

This is exactly the problem we're going to solve in the next lesson, using a message

`Stash` to defer processing of messages until we're ready.

Let's move onto Lesson 5 - Using `Stash` to Defer Processing of Messages.

Any questions?

Don't be afraid to ask questions :).

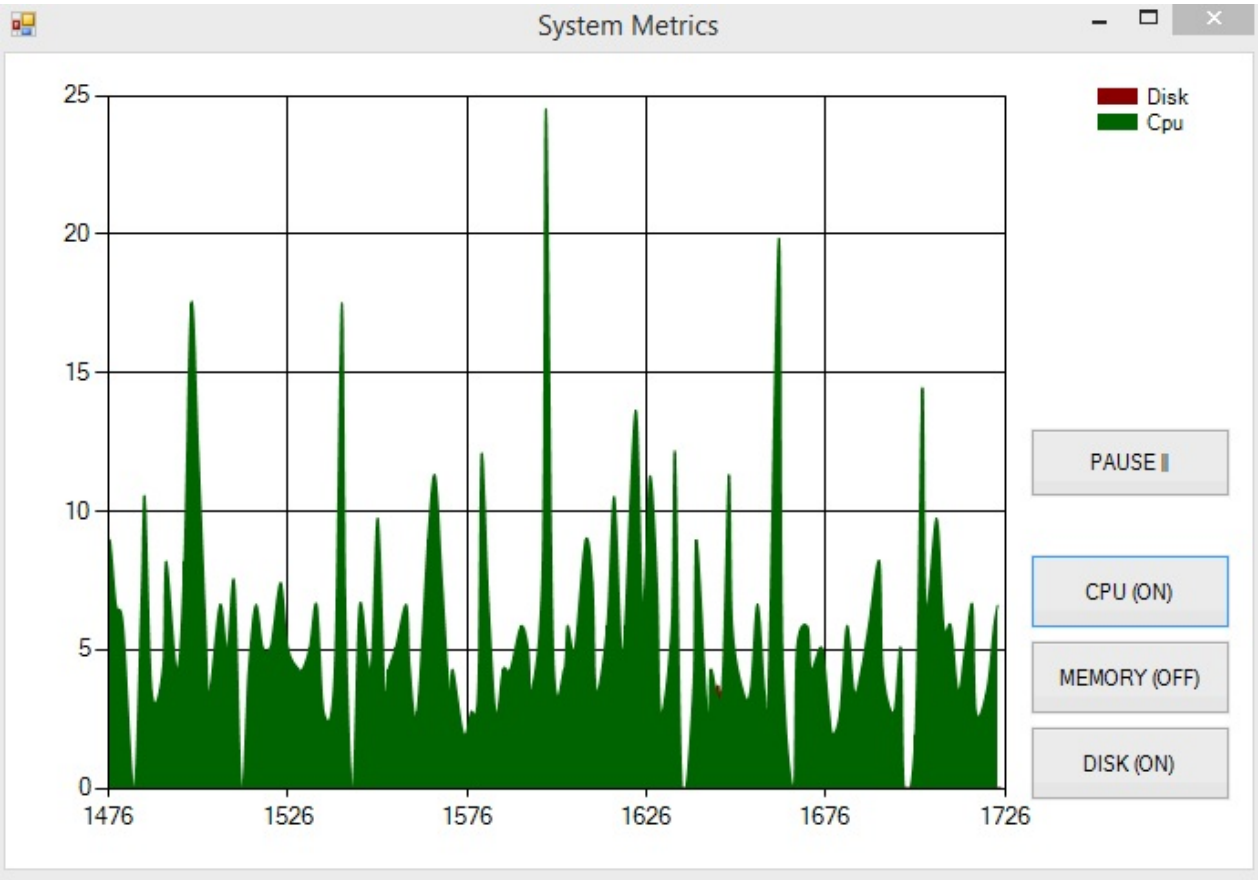
Come ask any questions you have, big or small, in this ongoing Bootcamp chat with the Petabridge & Akka.NET teams.

Problems with the code?

If there is a problem with the code running, or something else that needs to be fixed in this lesson, please create an issue and we'll get right on it. This will benefit everyone going through Bootcamp.

Lesson 2.5: Using `Stash` to Defer Processing of Messages

At the end of Lesson 4 we discovered a significant bug in how we implemented the **Pause / Resume** functionality on live charts, as you can see below:



The bug is that when our `ChartingActor` changes its behavior to `Paused`, it no longer processes the `AddSeries` and `RemoveSeries` messages generated whenever a toggle button is pressed for a particular performance counter.

In it's current form, it doesn't take much for the visual state of the buttons to get completely out of sync with the live chart. All you have to do is press a toggle button when the graph is paused and it's immediately out of sync.

So, how can we fix this?

The answer is to defer processing of `AddSeries` and `RemoveSeries` messages until the `ChartingActor` is back in its `Charting` behavior, at which time it can actually do something

with those messages.

The mechanism for this is the `Stash`.

Key Concepts / Background

One of the side effects of switchable behavior for actors is that some behaviors may not be able to process specific types of messages. For instance, let's consider the authentication example we used for behavior-switching in Lesson 4.

What is the `Stash`?

The `Stash` is a stack-like data structure implemented in your actor to defer messages for later processing.

How to add a `Stash` to your actor

To add a `Stash` to an actor, all we need to do is decorate it with the `IWithBoundedStash` or `IWithUnboundedStash` interface, like so:

```
public class UserActor : ReceiveActor, IWithUnboundedStash {
    private readonly string _userId;
    private readonly string _chatRoomId;

    // added along with the IWithUnboundedStash interface
    public IStash Stash {get;set;}

    // constructors, behaviors, etc...
}
```

When do we want a `BoundedStash` vs. an `UnboundedStash`

99% of the time you are going to want to use `UnboundedStash` - which allows your `Stash` to accept an unlimited number of messages. A `BoundedStash` should only be used when you want to set a maximum number of messages that can be stashed at any given time - your actor will crash whenever your `Stash` exceeds the limit of your `BoundedStash`.

Do we have to initialize the `Stash`?

But wait a minute, there's a new `Stash` property on the `UserActor` that includes a public getter and setter - does this mean that we have to initialize `Stash` ourselves? **NO!**

The `Stash` property is automatically populated by an Akka.NET feature known as the "Actor Construction Pipeline," which gets used every time an actor is created locally (the details are beyond the scope of this lesson.)

When the `ActorSystem` sees a `IWithBoundedStash` interface on an actor, it knows to automatically populate a `BoundedStash` inside its `Stash` property. Likewise, if it sees a `IWithUnboundedStash` interface, it knows to populate a `UnboundedStash` in that property instead.

How to use the `Stash`

Now that we've added a `Stash` to `UserActor`, how do we actually use it to store messages for later processing, and release previously stored messages to be processed?

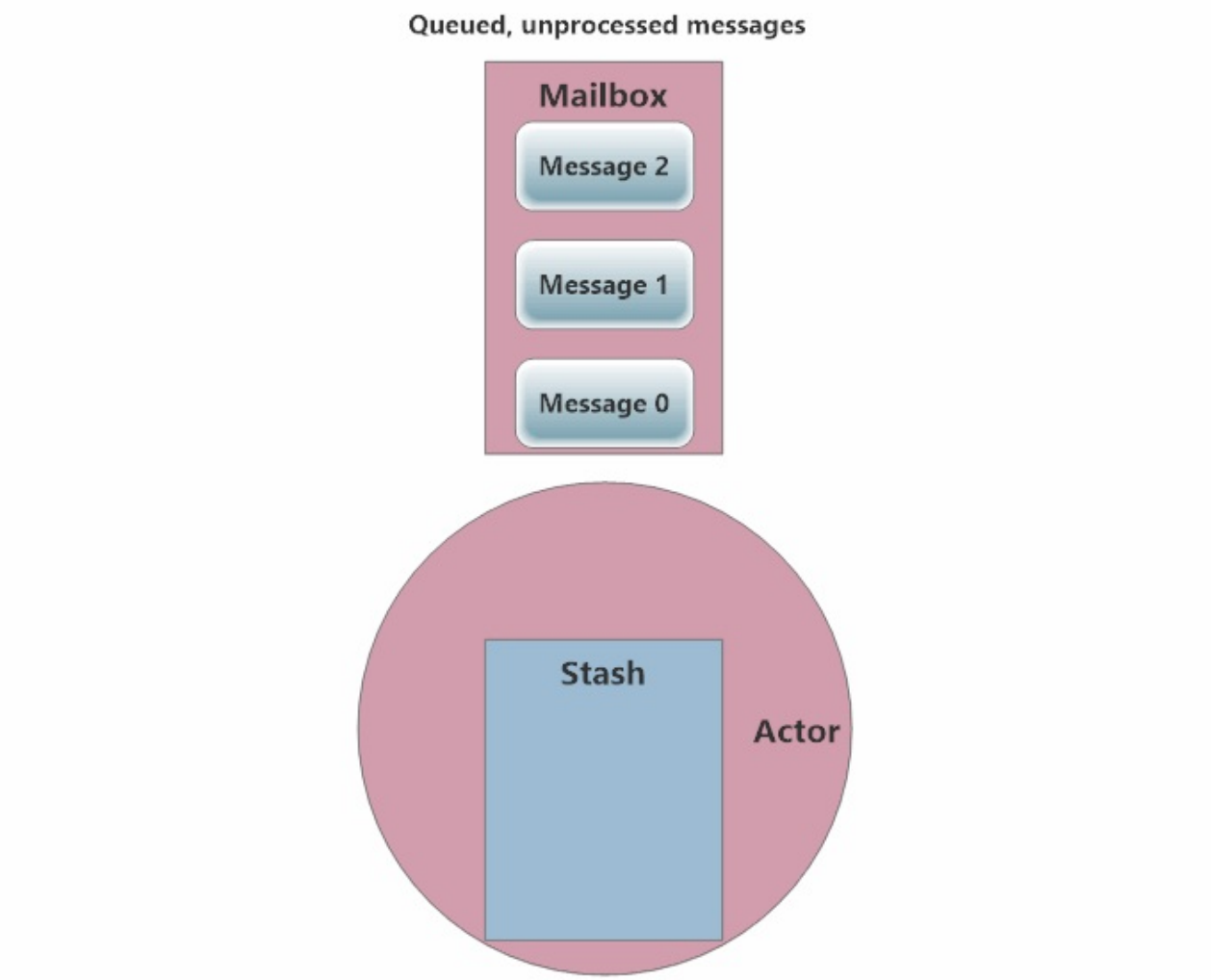
Stashing Messages

Inside your actor's `OnReceive` or `Receive<T>` handler, you can call `Stash.Stash()` to put the current message at the top of the `Stash`.

You only need to stash messages that you don't want to process now - in the below visualization, our actor happily processes Message 1 but stashes messages 2 and 0.

Note: calling `Stash()` automatically stashes the current message, so you don't pass the message to the `Stash.Stash()` call.

This is what that the full sequence of stashing a message looks like:



Great! Now that we know how to `Stash` a message for later processing, how do we get messages back out of the `Stash`?

Unstashing a Single Message

We call `Stash.Unstash()` to pop off the message at the top of the `Stash`.

When you call `Stash.Unstash()`, the `Stash` will place this message *at the front of the actor's mailbox, ahead of other queued user messages*.

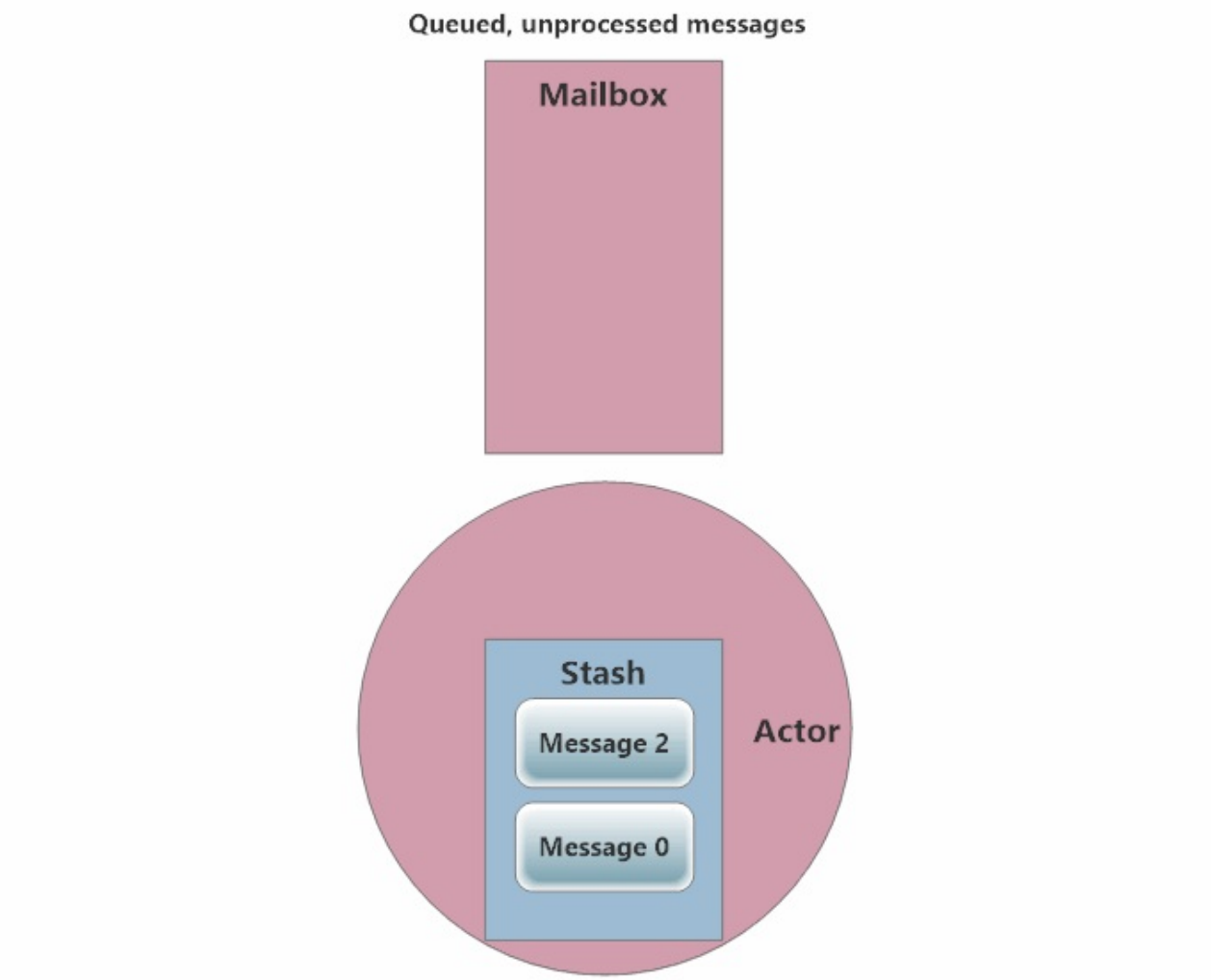
The VIP line

Inside the mailbox, it's as if there are two separate queues for `user` messages to be processed by the actor: there's the normal message queue, and then there's the VIP line.

This VIP line is reserved for messages coming from the `Stash`, and any messages in the VIP line will jump ahead of messages in the normal queue and get processed by the

actor first. (On that note, there's also a "super VIP" line for `system` message, which cuts ahead of all `user` messages. But that's out of the scope of this lesson.)

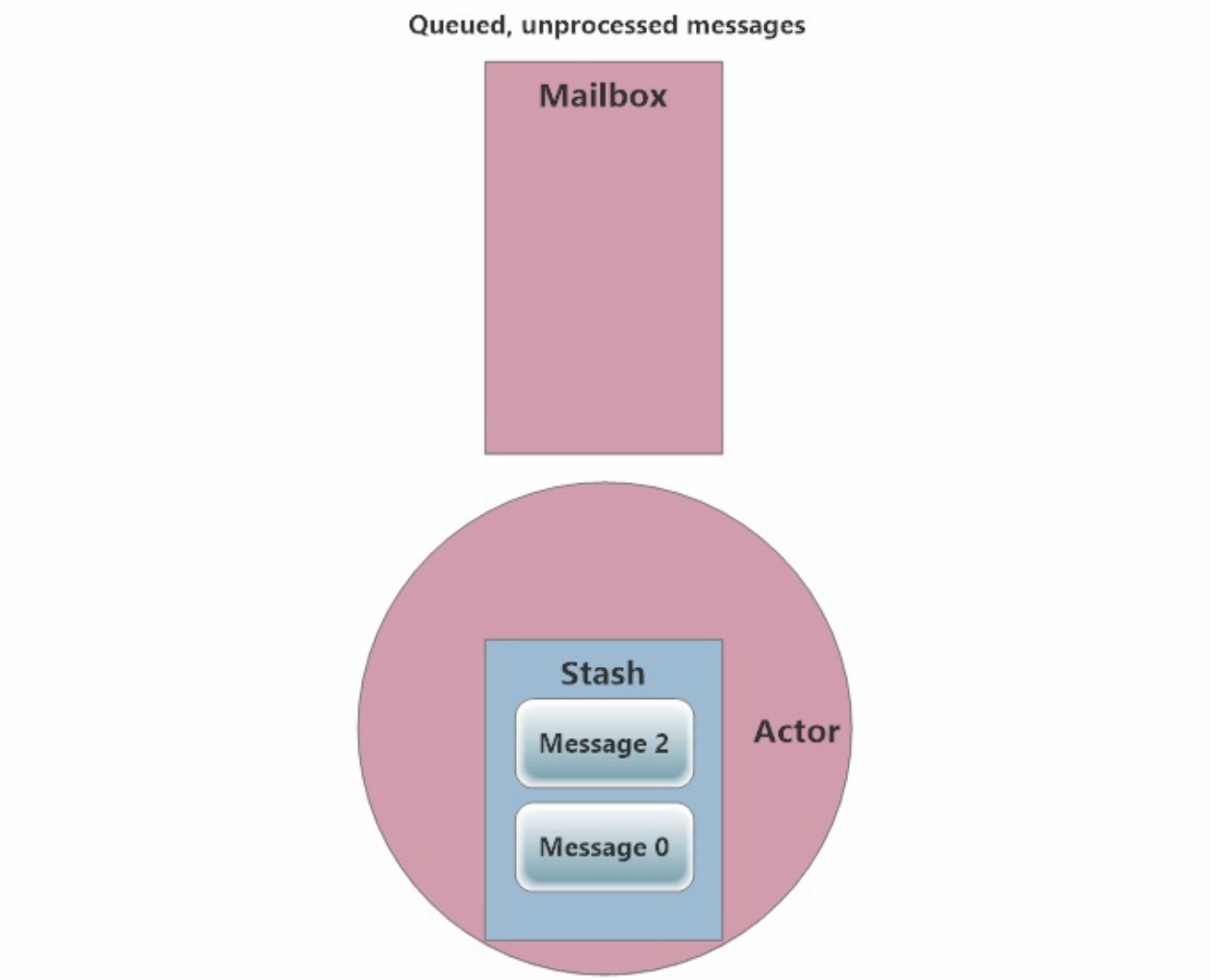
This is what the sequence of unstashing a message looks like:



Unstashing the Entire Stash at Once

If we need to unstash *everything* in our actor's `Stash` all at once, we can use the `Stash.UnstashAll()` method to push the entire contents of the `Stash` into the front of the mailbox.

Here's what calling `Stash.UnstashAll()` looks like:



Do messages stay in their original order when they come out of the `Stash`?

It depends on how you take them out of the `Stash`.

`Stash.UnstashAll()` preserves FIFO message order

When you make a call to `Stash.UnstashAll()`, the `Stash` will ensure that the original FIFO order of the messages in the `Stash` is preserved when they're appended to the front of your actor's mailbox. (As shown in the `Stash.UnstashAll()` animation.)

`Stash.Unstash()` can change the message order

If you call `Stash.Unstash()` repeatedly, you can change the original FIFO order of the messages.

Remember that VIP line inside the mailbox, where the `Stash` puts messages when they

are unstashed?

Well, when you `Unstash()` a ***single*** message, it goes to the back of that VIP line. It's still ahead of normal `user` messages, but it is behind any other messages that were previously unstashed and are ahead of it in the VIP line.

There is a lot more that goes into *why* this can happen, but it's well beyond the scope of this lesson.

Does a `Stash`ed message lose any data?

Absolutely not. When you `Stash` a message, you're technically stashing the message AND the message `Envelope`, which contains all the metadata for the message (its `Sender`, etc).

What Happens to the Messages in an Actor's `Stash` During Restarts?

An excellent question! The `Stash` is part of your actor's ephemeral state. In the case of a restart, the stash will be destroyed and garbage collected. This is the opposite of the actor's mailbox, which persists its messages across restarts.

However, you can preserve the contents of your `Stash` during restarts by calling `Stash.UnstashAll()` inside your actor's `PreRestart` lifecycle method. This will move all the stashed messages into the actor mailbox, which persists through the restart:

```
// move stashed messages to the mailbox so they persist through restart
protected override void PreRestart(Exception reason, object message){
    Stash.UnstashAll();
}
```

Real-World Scenario: Authentication with Buffering of Messages

Now that you know what the `Stash` is and how it works, let's revisit the `UserActor` from our chat room example and solve the problem with throwing away messages before the user was `Authenticated`.

This is the UserActor we designed in the Concepts area of lesson 4, with behavior switching for different states of authentication:

```
public class UserActor : ReceiveActor {
    private readonly string _userId;
    private readonly string _chatRoomId;

    public UserActor(string userId, string chatRoomId) {
        _userId = userId;
        _chatRoomId = chatRoomId;

        // start with the Authenticating behavior
        Authenticating();
    }

    protected override void PreStart() {
        // start the authentication process for this user
        Context.ActorSelection("/user/authenticator/")
            .Tell(new AuthenticatePlease(_userId));
    }

    private void Authenticating() {
        Receive<AuthenticationSuccess>(auth => {
            Become(Authenticated); //switch behavior to Authenticated
        });
        Receive<AuthenticationFailure>(auth => {
            Become(Unauthenticated); //switch behavior to Unauthenticated
        });
        Receive<IncomingMessage>(inc => inc.ChatRoomId == _chatRoomId,
            inc => {
                // can't accept message yet - not auth'd
            });
        Receive<OutgoingMessage>(inc => inc.ChatRoomId == _chatRoomId,
            inc => {
                // can't send message yet - not auth'd
            });
    }

    private void Unauthenticated() {
        // switch to Authenticating
        Receive<RetryAuthentication>(retry => Become(Authenticating));
        Receive<IncomingMessage>(inc => inc.ChatRoomId == _chatRoomId,
            inc => {
                // have to reject message - auth failed
            });
        Receive<OutgoingMessage>(inc => inc.ChatRoomId == _chatRoomId,
            inc => {
                // have to reject message - auth failed
            });
    }

    private void Authenticated() {
        Receive<IncomingMessage>(inc => inc.ChatRoomId == _chatRoomId,
```

```
            inc => {
                // print message for user
            });
        Receive<OutgoingMessage>(inc => inc.ChatRoomId == _chatRoomId,
            inc => {
                // send message to chatroom
            });
    }
}
```

When we first saw that chat room `UserActor` example in lesson 4, we were focused on switching behaviors to enable authentication in the first place. But we ignored a major problem with the `UserActor`: during the `Authenticating` phase, we simply throw away any attempted `OutgoingMessage` and `IncomingMessage` instances.

We're losing messages to/from the user for no good reason, because we didn't know how to delay message processing. **Yuck!** Let's fix it.

The right way to deal these messages is to temporarily store them until the `UserActor` enters either the `Authenticated` or `Unauthenticated` state. At that time, the `UserActor` will be able to make an intelligent decision about what to do with messages to/from the user.

This is what it looks like once we update the `Authenticating` behavior of our `UserActor` to delay processing messages until it knows whether or not the user is authenticated:

```
public class UserActor : ReceiveActor, IWithUnboundedStash {
    // constructors, fields, etc...

    private void Authenticating() {
        Receive<AuthenticationSuccess>(auth => {
            Become(Authenticated); // switch behavior to Authenticated
            Stash.UnstashAll(); // move all stashed messages to the mailbox for processing i
        });
        Receive<AuthenticationFailure>(auth => {
            Become(Unauthenticated); // switch behavior to Unauthenticated
            Stash.UnstashAll(); // move all stashed messages to the mailbox for processing i
        });
        Receive<IncomingMessage>(inc => inc.ChatRoomId == _chatRoomId,
            inc => {
                // save this message for later
                Stash.Stash();
            });
        Receive<OutgoingMessage>(inc => inc.ChatRoomId == _chatRoomId,
            inc => {
                // save this message for later
                Stash.Stash();
            });
```

```
        }

        // other UserActor behaviors...
    }
```

Now any messages the `UserActor` receives while it's `Authenticating` will be available for processing when it switches behavior to `Authenticated` or `Unauthenticated`.

Excellent! Now that you understand the `Stash`, let's put it to work to fix our system graphs.

Exercise

In this section, we're going to use an `UnboundedStash` to fix the **Pause / Resume** bug inside the `ChartingActor` that we noticed at the end of Lesson 4.

Phase 1 - Have the `ChartingActor` Implement the `WithUnboundedStash` Interface

Inside `Actors/ChartingActor.cs`, update the `ChartingActor` class declaration and have it implement the `WithUnboundedStash` interface:

```
// Actors/ChartingActor.cs
public class ChartingActor : ReceiveActor, IWithUnboundedStash
```

Also, implement the interface and have it add the following property somewhere inside `ChartingActor`:

```
// Actors/ChartingActor.cs - inside ChartingActor class definition
public IStash Stash { get; set; }
```

Phase 2 - Add `Stash` Method Calls to `Receive<T>` Handlers Inside `Paused()` Behavior

Go to the `Paused()` method declared inside `ChartingActor`.

Update it to `Stash()` the `AddSeries` and `RemoveSeries` messages:

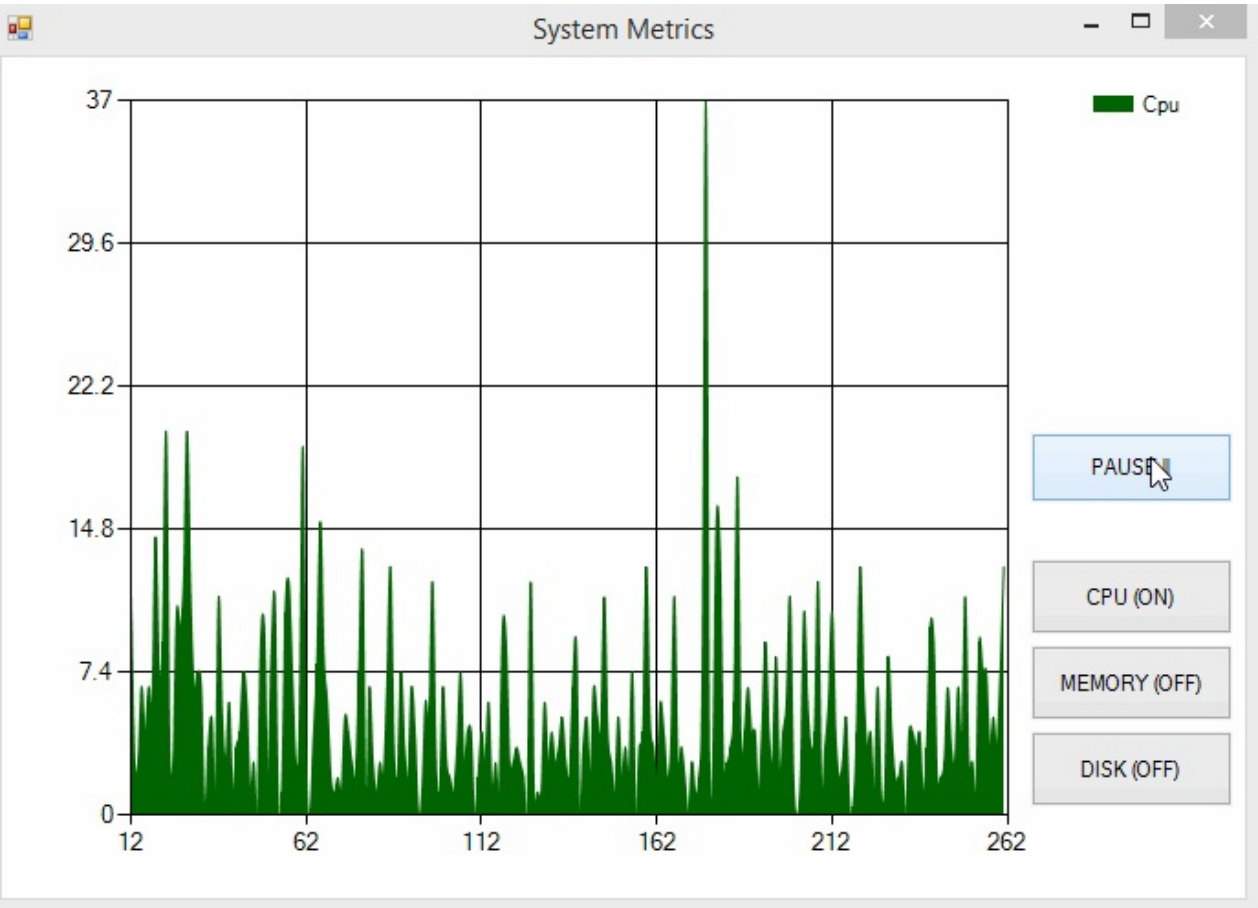
```
// Actors/ChartingActor.cs - inside ChartingActor class definition
private void Paused()
{
    // while paused, we need to stash AddSeries & RemoveSeries messages
    Receive<AddSeries>(addSeries => Stash.Stash());
    Receive<RemoveSeries>(removeSeries => Stash.Stash());
    Receive<Metric>(metric => HandleMetricsPaused(metric));
    Receive<TogglePause>(pause =>
    {
        SetPauseButtonText(false);
        UnbecomeStacked();

        // ChartingActor is leaving the Paused state, put messages back
        // into mailbox for processing under new behavior
        Stash.UnstashAll();
    });
}
```

That's it! The `ChartingActor` will now save any `AddSeries` or `RemoveSeries` messages and will replay them in the order they were received as soon as it switches from the `Paused()` state to the `Charting()` state.

The bug should now be fixed!

Once you're done

Build and run `SystemCharting.sln` and you should see the following:



Compare your code to the code in the /Completed/ folder to compare your final output to what the instructors produced.

Great job!

Wohoo! You did it! Unit 2 is complete! Now go enjoy a well-deserved break, and gear up for Unit 3!

Ready for more? Start Unit 3 now.

Any questions?

Don't be afraid to ask questions :).

Come ask any questions you have, big or small, in this ongoing Bootcamp chat with the Petabridge & Akka.NET teams.

Problems with the code?

If there is a problem with the code running, or something else that needs to be fixed in this lesson, please create an issue and we'll get right on it. This will benefit everyone going through Bootcamp.

Akka.NET Bootcamp - Unit 3: Advanced Akka.NET

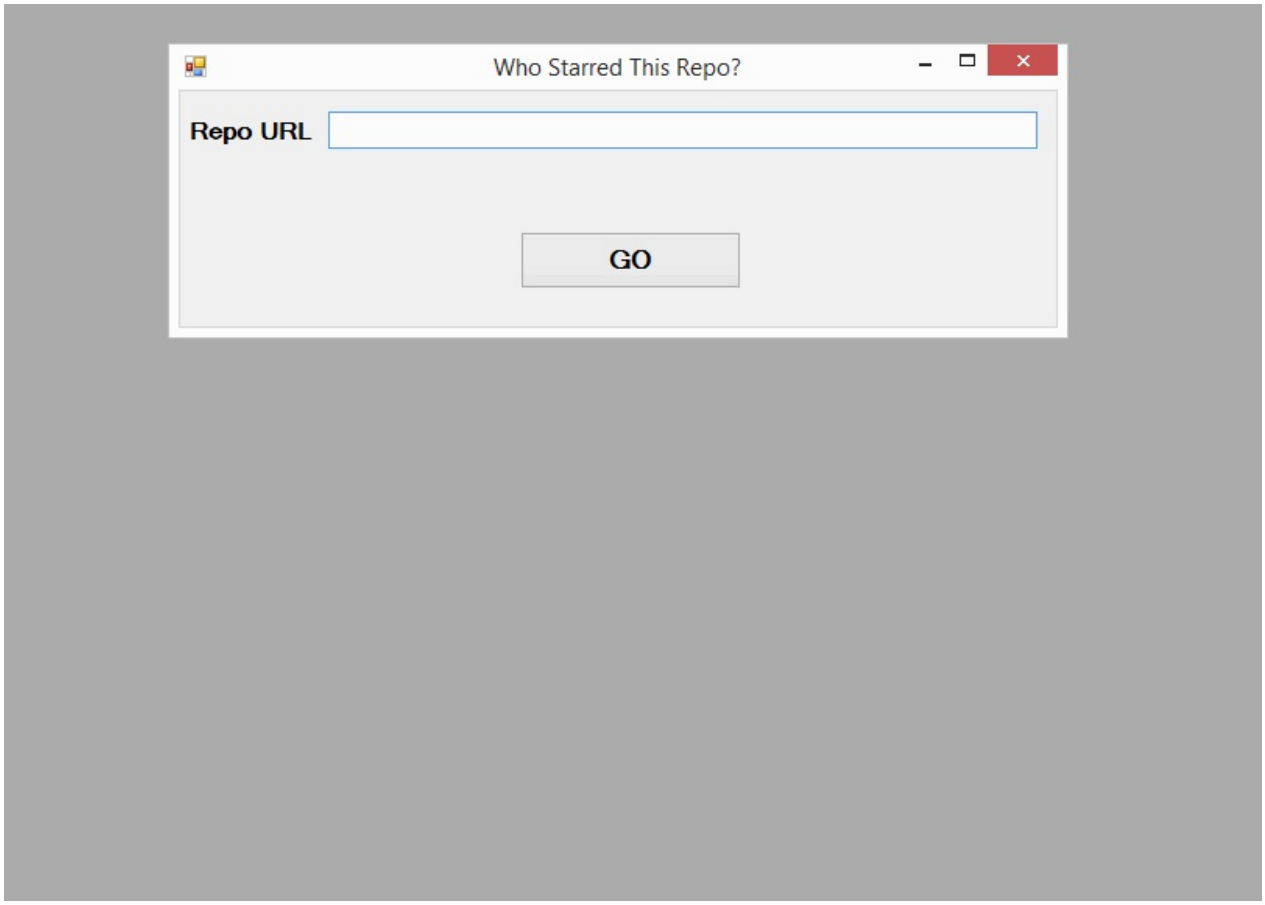
In **Unit 1**, we learned some of the fundamentals of Akka.NET and the actor model.

In **Unit 2** we learned some of the more sophisticated concepts behind Akka.NET, such as pattern matching, basic Akka.NET configuration, scheduled messages, and more!

In Unit 3, we're going to learn how to leverage the Task Parallel Library (TPL) and Akka.NET routers to scale out actor systems for massive performance boosts via parallelism.

Concepts you'll learn

Over the course of Unit 3, you're going to build a sophisticated GitHub scraper that can simultaneously retrieve data from multiple GitHub repos at once.



This system will also be able to fetch information about the GitHubbers who have participated in those repos (e.g. starred or forked). By the end, we'll have a nicely scalable system for retrieving data from the GitHub API, capable of coordinating a huge amount of data retrieval in parallel (up to the allowed rate limit of the API, of course)!

In Unit 3 you will learn:

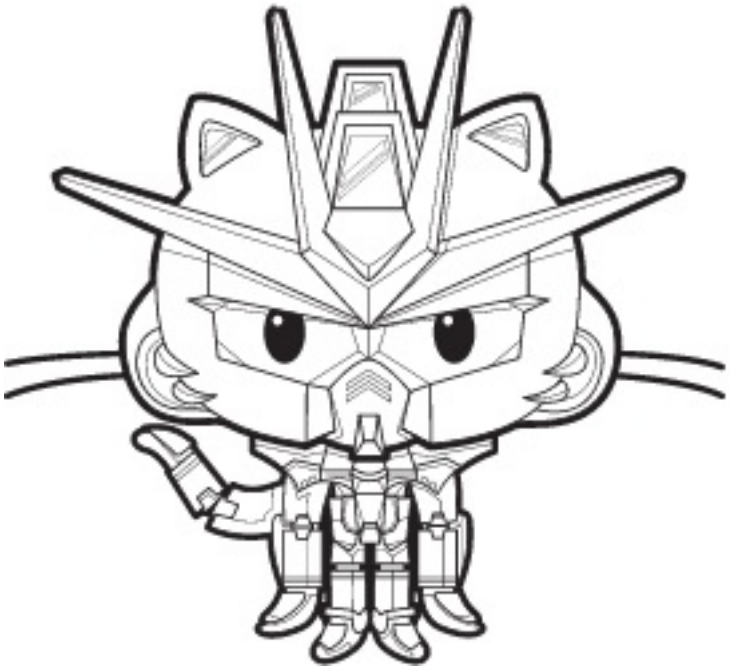
1. How to use `Group` routers to divide work among your actors
2. How to use `Pool` routers to automatically create and manage pools of actors
3. How to use HOCON to configure your routers
4. How to use `Ask` to wait inline for actors to respond to your messages
5. How to perform work asynchronously inside your actors using `PipeTo`

Teaming up with Octokit, the official GitHub SDK for .NET

In this lesson we'll also be introducing you to Octokit, the official GitHub SDK for .NET (and other languages!)

> **OCTOKIT NOTE:** If you're working behind a proxy server and have issues connecting to the Github API with Octokit, try adding this to your config file and

> see if it fixes the issue:
>
> `<system.net><defaultProxy useDefaultCredentials="true" /></system.net>`



If you have any questions about Octokit or want to learn more about it, make sure you check out Octokit.NET on GitHub!

Table of Contents

Things you'll need

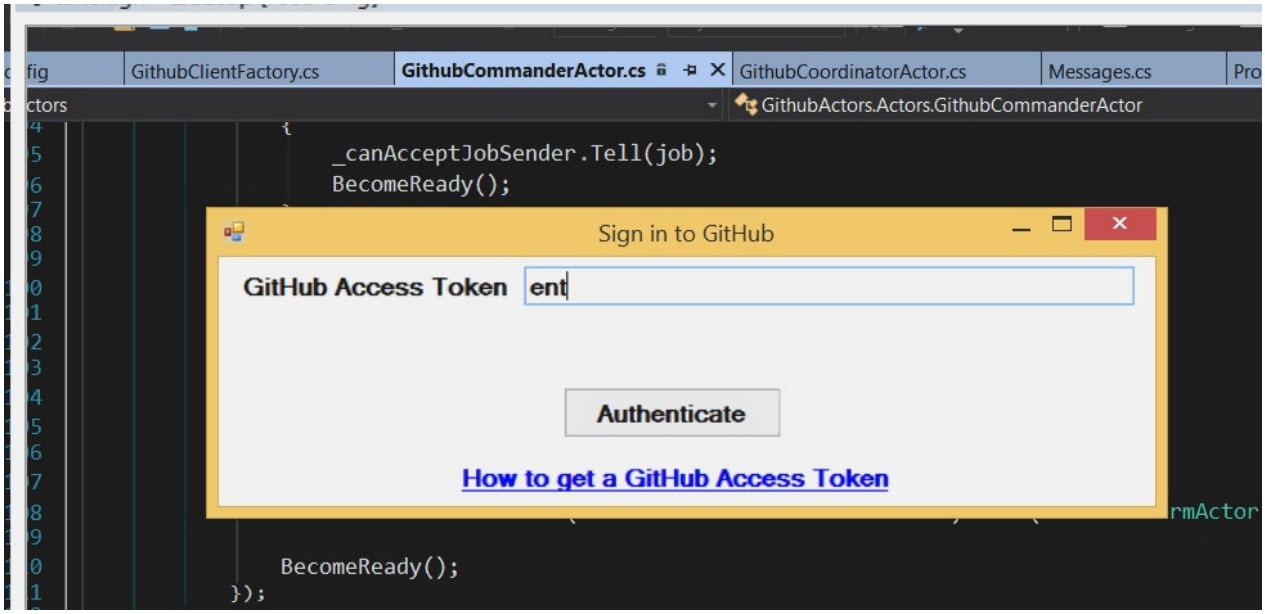
You will need to make a GitHub OAuth access token for the API.

This will be a throwaway token in your account only used for this app. Follow the instructions here and write down your OAuth token.

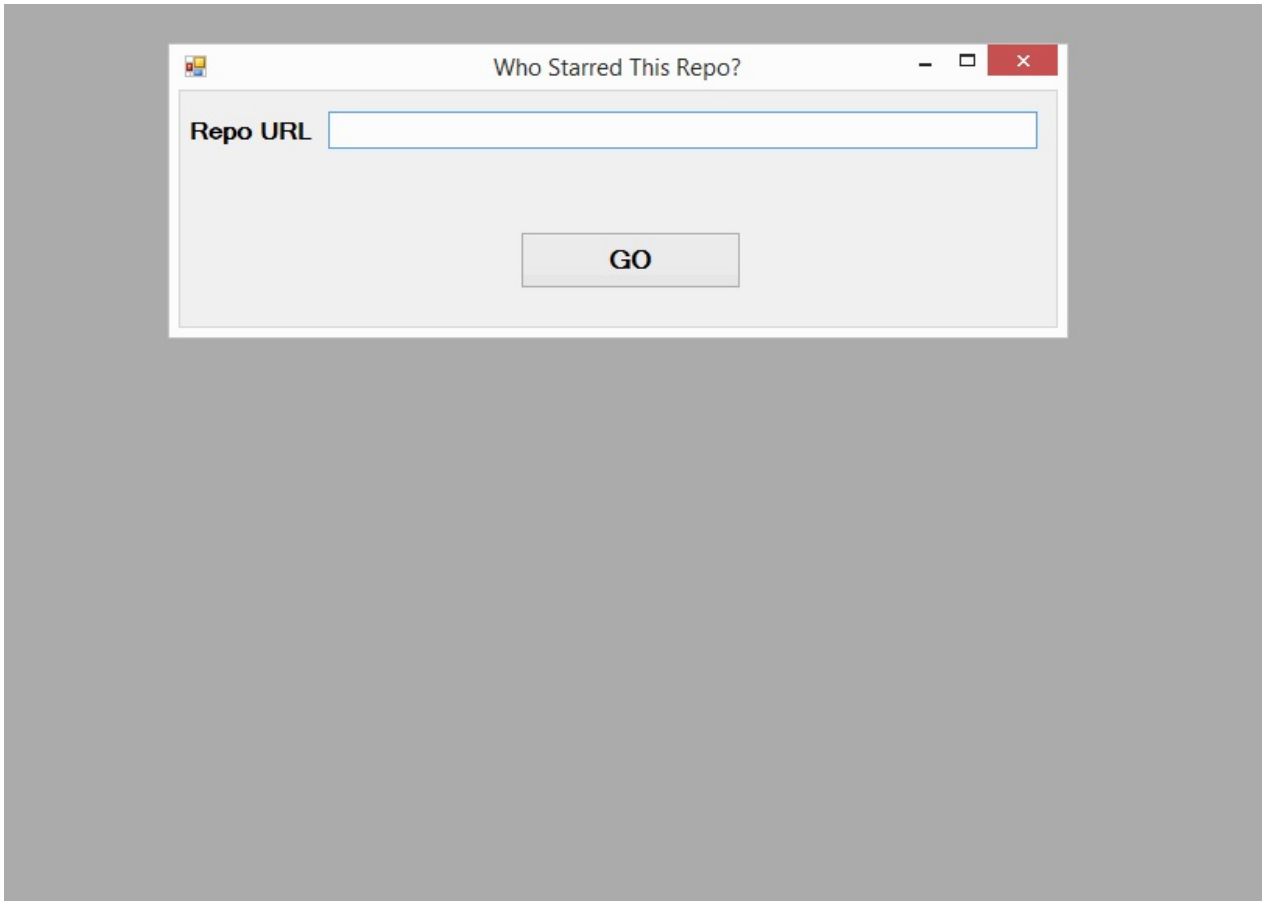
When you run the samples and are actually pulling data from GitHub, you will get two popup windows:

1. The first popup window will ask you for your GitHub token. This is the read-only access token that you just created.
2. The second window is where you'll enter the URL of the actual repo that you want to fetch info for.

This is where you enter your access token when you run the code:



You will then get a second window, where you enter the URL of the repo you want to inspect:



Get Started

To get started, go to the /DoThis/ folder and open `GithubActors.sln`.

And then go to Lesson 1.

Lesson 3.1: Using `Group` routers to divide work among your actors

Welcome to Unit 3! As you know, this unit focuses on making our actor systems more scalable, resilient, and parallelized.

Over the course of Unit 3, you're going to build a sophisticated GitHub scraper that can simultaneously retrieve data from multiple GitHub repos at once. This system will also be able to fetch information about the GitHubbers who have participated in those repos (e.g. starred or forked). By the end, we'll have a nicely scalable system for retrieving data from the GitHub API, capable of coordinating a huge amount of data retrieval in parallel!

> ***Heads up: This lesson is the most critical (and longest) of all the lessons in Unit 3. Grab some coffee and get comfortable!***

The most important new concept we need to learn is `Router`s (docs). Let's get going.

Key Concepts / Background

`Router`s

What is a `Router`?

A `Router` is a special kind of actor that acts as a messaging hub to a group of other actors. The purpose of a `Router` is to divide and balance work (represented as message streams) across some group of other actors, which will actually do the work.

What's special about a `Router`?

A `Router` actually *is* an actor, but with one critical difference: it can process more than one message at a time.

Wait, what? I thought all actors only processed one message at a time?! That is true for 99.999% of cases. But remember: *The purpose of a `Router` is to route messages*

onward, NOT to process them.

Since a `Router` does not need to actually process messages and take any significant action on the message—it just has to forward the message on to another actor—it can break the "one message at a time" rule and everything is fine. In fact, it's better that way. Why?

What does a `Router` give me?

Fundamentally, what a `Router` gives you is massive message throughput and scalability.

`Router`s are the critical component that breaks down large data streams into easily managed ones. `Router`s allow you to divide up a huge amount of work across a group of actors and scale up processing easily.

On the surface, `Router`s look like normal actors, but they are actually implemented differently. Routers are designed to be extremely efficient at one thing: receiving messages and quickly passing them on to routees.

A normal actor *can* be used for routing messages, but a normal actor's single-threaded processing can become a bottleneck. Routers achieve much higher throughput by changing the usual message-processing pipeline to allow concurrent message routing. This is achieved by embedding the routing logic of a `Router` directly in their `ActorCell` itself (the mesh between the actor's mailbox and the actor,) rather than in the receive loop / message-handling code of the router actor. This way, messages sent to a router's `IActorRef` are immediately routed to the routee, bypassing the single-threaded message-handling code of the router's actor entirely.

Fortunately, *all of this complexity is invisible to consumers of the routing API*.

Okay, what kinds of `Router`s are there?

There are two types of routers: Pool routers, and Group routers.

Pool Routers

A "Pool" router is a `Router` that creates and manages its worker actors ("routees"). You provide a `NrOfInstances` to the router and the router will handle routee creation (and

supervision) by itself.

> **We cover Pool routers in depth in lesson 2**.

Group Routers

A group router is a router that does not create/manage its routees. It only forwards them messages. You specify the routees when creating the group router by passing the router the `ActorPath`s for each routee.

In this lesson, we'll be working with Group routers.

How do I configure a `Router`?

You can configure a router directly in your code (dubbed "procedural" or "programatic" configuration), or you can configure the router using HOCON and `App.config`.

We'll use procedural configuration in this lesson and the next, but go in depth on using HOCON to configure routers in Lesson 3.

When a routee gets a message, is the `Sender` the `Router`, or the actor that sent the message to the `Router`?

Great question. Glad you asked!

The `Sender` of a message delivered to a routee is the actor who sent the message to the router. The router just forwards the message. A router is effectively transparent, and only works in one direction: to its routees.

So, if the routee replies to the message, that response will go to the original actor, bypassing the router altogether. When replying this way, the routee can also specify the router as the `Sender` for its response, so that any follow-up reply from the original actor flows back through the router.

How does a `Router` know where to forward a message?

A `Router` decides how to distribute messages to its routees based on the `RoutingStrategy` you assign it.

Routing Strategies

A `RoutingStrategy` encapsulates the logic used by a router to distribute messages to its routees.

There are quite a few types of `RoutingStrategy` available to you out of the box. But, we divide them into two general categories: *special-case strategies*, and *load balancing strategies*.

You can also define your own `RoutingStrategy`, but that's quite complex and beyond the scope of this course.

Let's go over these message routing strategies.

Special-case `RoutingStrategy`s

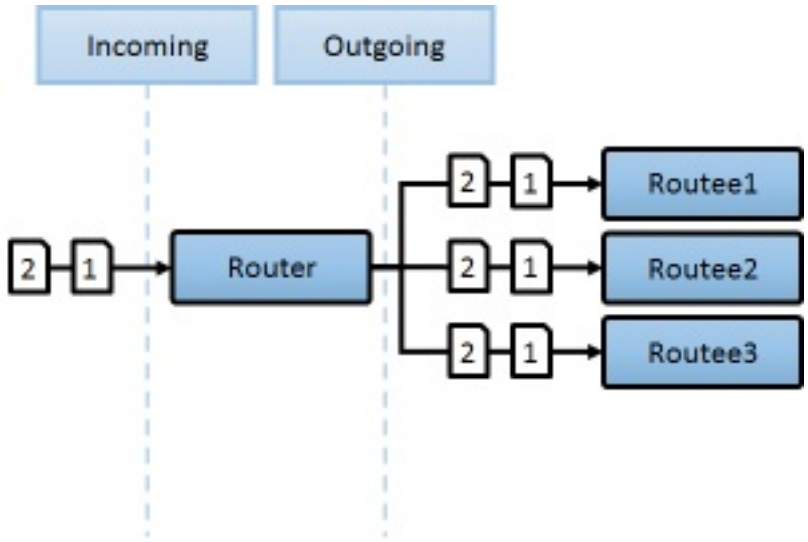
Each of these is a `RoutingStrategy` with a specialized purpose.

1. `Broadcast`
2. `Random`
3. `ConsistentHash`

`Broadcast`

Under this `RoutingStrategy`, the router will forward any message it receives to ***ALL*** of its routees.

Here's what the `Broadcast` `RoutingStrategy` looks like:



`Random`

Under the `Random` `RoutingStrategy`, each time the `Router` receives a message, it will pick a routee at random to forward the message to.

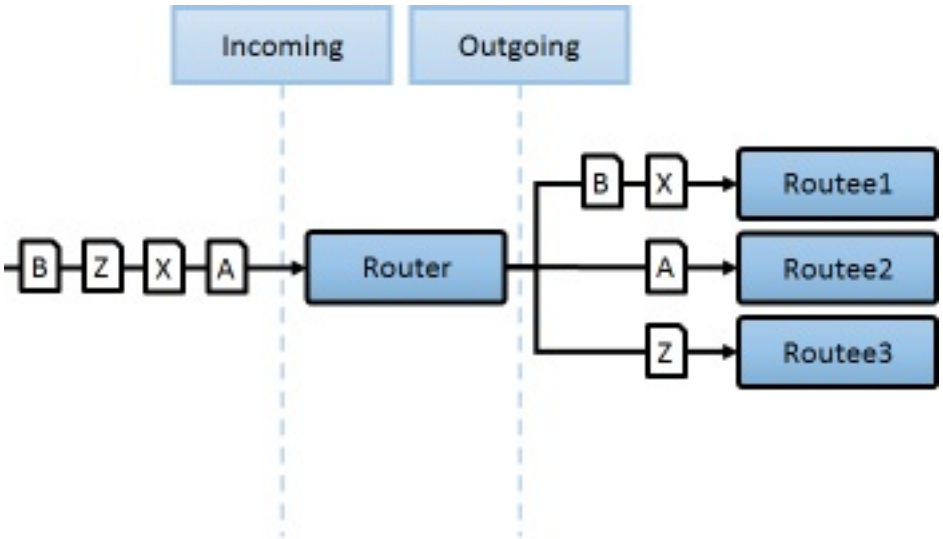
The router will repeat this process each time it receives a message, so messages 1, 2, 3...N will each be sent to a randomly selected actor.

`ConsistentHash`

With the `ConsistentHash` `RoutingStrategy`, the `Router` uses consistent hashing to select the routee based on the data of the message that was sent.

This has a specific use case where you want the routing of the message to depend on the message itself. For example, imagine that you had a system that needed to process events on a per-user level. To do this, your analytics events could flow to a `ConsistentHashRouter` that then forwards the event message on to a per-user `UserEventTrackingActor`, which will track all the events for that given user. The hash key would probably be the user UUID contained inside each of the user-generated events.

Here's what the `ConsistentHash` `RoutingStrategy` looks like:



Load balancing `RoutingStrategy`s

These routing strategies are all different approaches to the same goal: *attempting to evenly distribute work across the group*.

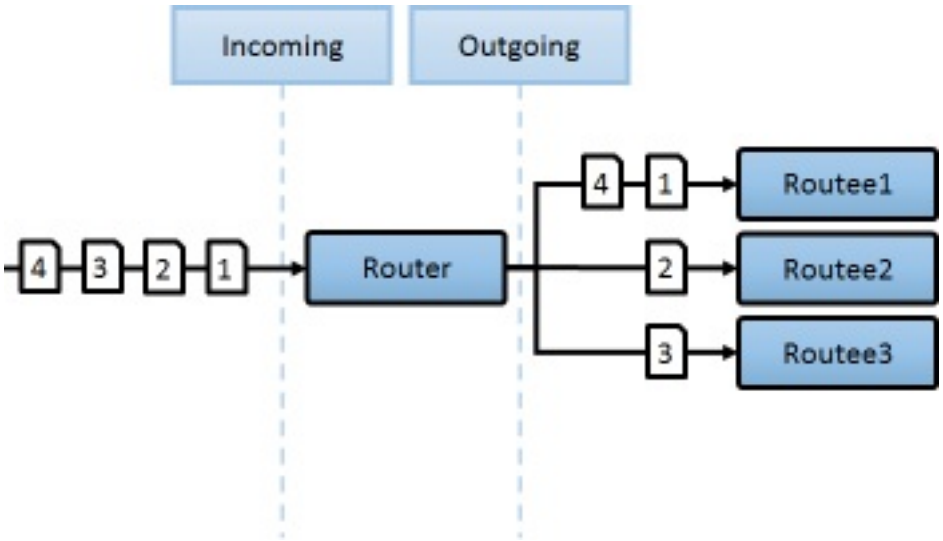
1. `RoundRobin`
2. `SmallestMailbox`
3. `TailChopping`
4. `Resizable`
5. `ScatterGatherFirstCompleted`

Let's review them quickly.

`RoundRobin`

Under this `RoutingStrategy`, the router will use round-robin to select a routee. For concurrent calls, `RoundRobin` is just a best effort—meaning that the router makes no complex effort to determine which routee is best suited for a message, it just sends it to the next in the round robin pattern and advances.

Here's what the `RoundRobin` `RoutingStrategy` looks like:



In practice, a `RoundRobin` strategy is going to be fine for most situations and should be your go-to load balancing `RoutingStrategy`.

However, if you find yourself with a router under *massive* load (e.g. 10s of millions of messages per second) or needing a unique pattern, then you may want to look at the more esoteric load balancing strategies that follow.

`TailChopping`

The TailChoppingRouter will first send the message to one, randomly picked, routee and then after a small delay to to a second routee (picked randomly from the remaining

routees) and so on. It waits for first reply it gets back and forwards it back to original sender. Other replies are discarded.

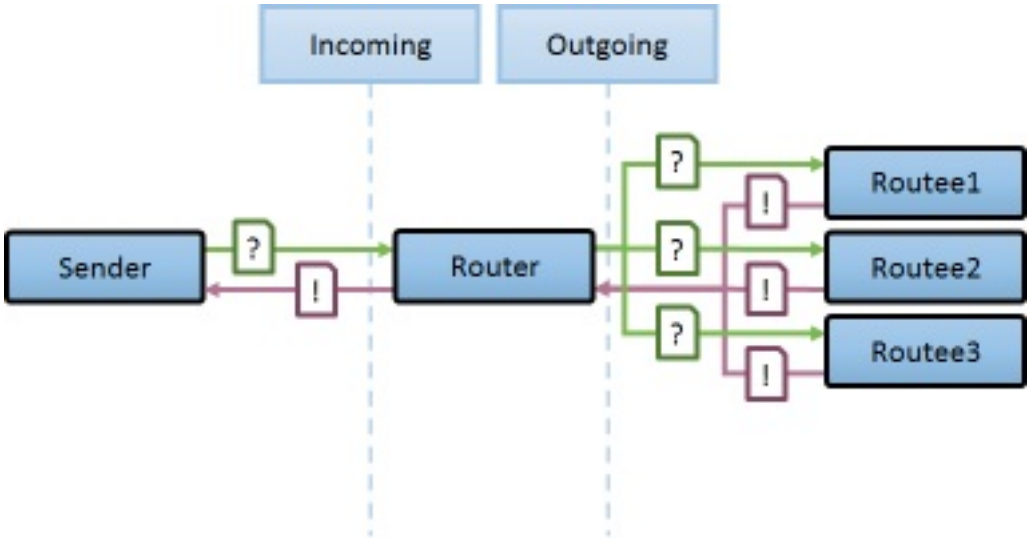
The goal of this router is to decrease latency by performing redundant queries to multiple routees, assuming that one of the other actors may still be faster to respond than the initial one.

`ScatterGatherFirstCompleted`

The `ScatterGatherFirstCompletedRouter` will send the message on to all its routees. It then waits for first reply it gets back. This result will be sent back to original sender. Other replies are discarded.

`ScatterGatherFirstCompletedRouter` expects at least one reply within a configured duration, otherwise it will reply to its `Sender` with an exception.

Here's what `ScatterGatherFirstCompleted` looks like:



`SmallestMailbox`

This `RoutingStrategy` only works with Pool routers, so we'll cover it in the next lesson.

`ResizableRouter`

This `RoutingStrategy` only works with Pool routers, so we'll cover it in the next lesson.

Special `Router` messages

Regardless of its `RoutingStrategy`, there are a few special messages that you can send to a `Router` to cause certain behaviors.

`Broadcast`

Sending a `Broadcast` message to a non-`Broadcast` router makes the router act like a `BroadcastRouter` for that single message. After the message is processed, the router will return to its normal `RoutingStrategy`.

When would you use this? It doesn't come up very often, but one use case we can think of is if a group of routees all needed to take some action in response to a global-level event.

For example, perhaps you have a group of actors that all must be alerted if a critical system goes down. In this case, you could send their router a `Broadcast` message and all the routees would be alerted.

```
// tell a router to broadcast message to all its routees
// regardless of what type of router it actually is
router.Tell(new Broadcast("Shields failing, Captain!"));
```

`GetRoutees`

The `GetRoutees` message type tells a router to return a list of its routees. This is most commonly used for debugging, but in our example we'll also use it to track how many jobs are open.

```
// get a list of a routers routees
router.Tell(new GetRoutees());
```

`PoisonPill`

A `PoisonPill` message has special handling for all actors, including for routers. When any actor receives a `PoisonPill` message, that actor will be stopped immediately.

For a group router, this only stops the router and does not stop its routees. For a pool router, it will also have the effect of shutting down its child routees as they are supervised children of the router.

Sending a `PoisonPill` will terminate any actor immediately:

```
// kill router actor
router.Tell(PoisonPill.Instance);
```

Great! Now that you know what the different kinds of routers are, and how to use them, let's wrap up by covering how group routers and their routees recover from failures.

How does supervision work with Group routers?

A group router does not supervise its routees.

Recall that group routers do not create their routees, but instead are passed the `ActorPath`s of their routees. This means that those routees exist somewhere else in the hierarchy, and are managed by whatever other parent actors created them.

Practically, this means that a group router usually won't know that its routees have died. A group router will attempt to `DeathWatch` its routees, but it doesn't always succeed in subscribing. Much of this is due to the fact that `ActorPath`s can have wildcards.

Isn't it bad that group routers usually don't know their routees have died?

Yes, it is bad.

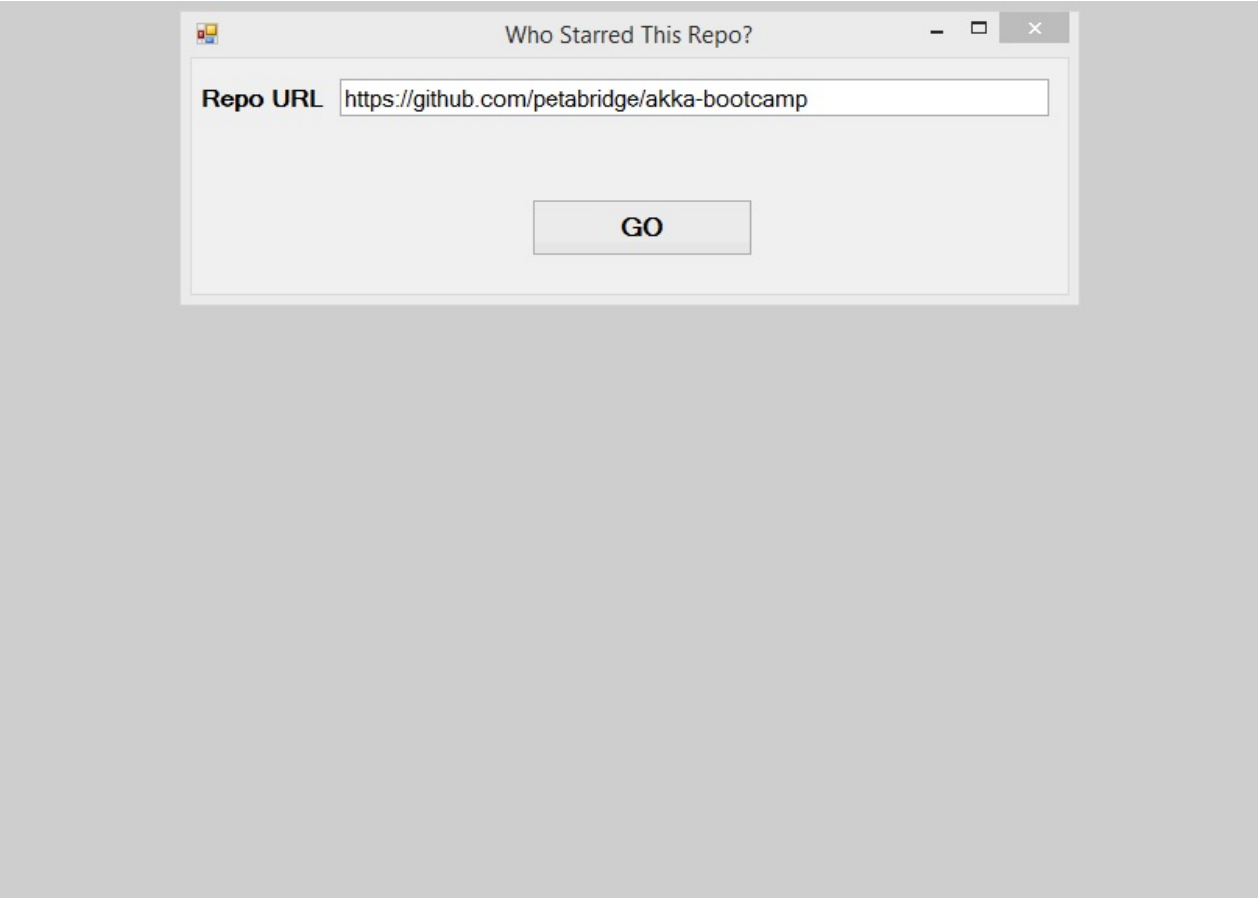
This is one of the key reasons that, in general, *we recommend using pool routers instead of group routers*.

In this lesson, we're only using group routers, but going forward we'll be using pool routers almost exclusively.

Phew! That was a LOT of new information. Now let's put it to use and make something!

Exercise

If you build and run `GithubActors.sln`, you'll notice that we can only process one GitHub repository at a time right now:



The current state of our actor hierarchy for processing GitHub repositories currently looks like this:

We're going to modify the `GithubCommanderActor` to use a `BroadcastGroup` router so we can run multiple jobs in parallel by the end of this lesson!

Phase 1 - Add `IWithUnboundedStash` to the `GithubCommanderActor`

Open `Actors\GithubCommanderActor.cs` and make the following changes to the actor declaration:

```
// Actors\GithubCommanderActor.cs
public class GithubCommanderActor : ReceiveActor, IWithUnboundedStash
```

And then add the `Stash` property to `GithubCommanderActor` somewhere

```
// inside GithubCommanderActor class definition
public IStash Stash { get; set; }
```

Phase 2 - Add switchable behaviors to the `GithubCommanderActor`

Replace the `GithubCommanderActor` 's constructor and current `Receive<T>` handlers with the following:

```
/* replace the current constructor and Receive<T> handlers
 * in Actors/GithubCommanderActor.cs with the following
 */
private int pendingJobReplies;

public GithubCommanderActor()
{
   Ready();
}

private void Ready()
{
    Receive<CanAcceptJob>(job =>
    {
        _coordinator.Tell(job);

        BecomeAsking();
    });
}

private void BecomeAsking()
{
    _canAcceptJobSender = Sender;
    pendingJobReplies = 3; //the number of routees
    Become(Asking);
}

private void Asking()
{
    // stash any subsequent requests
    Receive<CanAcceptJob>(job => Stash.Stash());

    Receive<UnableToAcceptJob>(job =>
    {
        pendingJobReplies--;
        if (pendingJobReplies == 0)
        {
            _canAcceptJobSender.Tell(job);
            BecomeReady();
        }
    });

    Receive<AbleToAcceptJob>(job =>
    {
        _canAcceptJobSender.Tell(job);
```

```
            // start processing messages
            Sender.Tell(new GithubCoordinatorActor.BeginJob(job.Repo));

            // launch the new window to view results of the processing
            Context.ActorSelection(ActorPaths.MainFormActor.Path).Tell(
                new MainFormActor.LaunchRepoResultsWindow(job.Repo, Sender));

            BecomeReady();
        });
    }

    private void BecomeReady()
    {
        Become(Ready);
        Stash.UnstashAll();
    }
```

Phase 3 - Modify `GithubCommanderActor.PreStart` to use a group router for `_coordinator`

The `GithubCommanderActor` is responsible for creating one `GithubCoordinatorActor`, who manages and coordinates all of the actors responsible for downloading data from GitHub via Octokit.

We're going to change the `GithubCommanderActor` to actually create 3 different `GithubCoordinatorActor` instances, and we're going to use a `BroadcastGroup` to communicate with them!

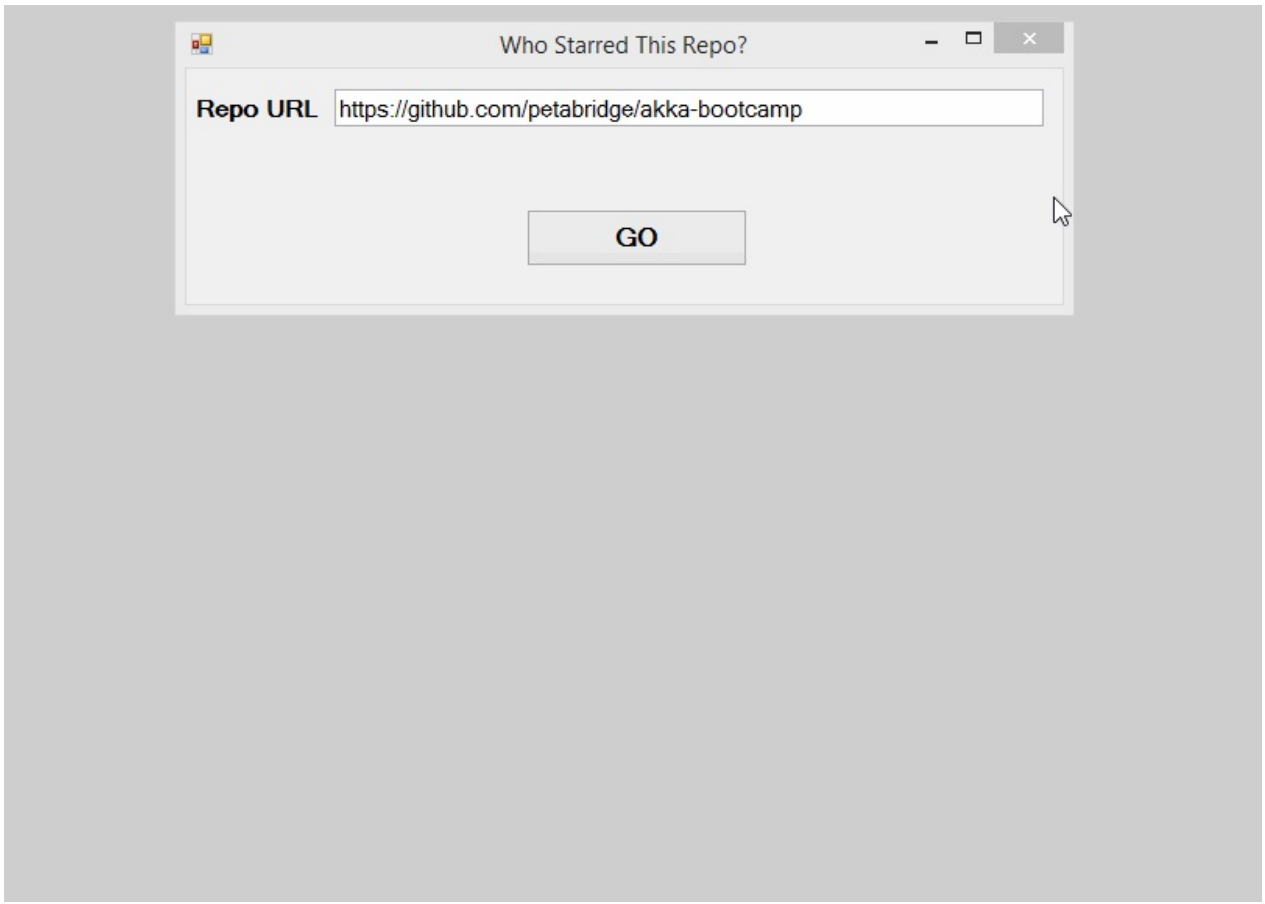
Replace the `GithubCommanderActor.PreStart` method with the following:

```
    protected override void PreStart()
    {
        // create three GithubCoordinatorActor instances
        var c1 = Context.ActorOf(Props.Create(() => new GithubCoordinatorActor()), ActorPaths.Gi
        var c2 = Context.ActorOf(Props.Create(() => new GithubCoordinatorActor()), ActorPaths.Gi
        var c3 = Context.ActorOf(Props.Create(() => new GithubCoordinatorActor()), ActorPaths.Gi

        // create a broadcast router who will ask all of them if they're available for work
        _coordinator =
            Context.ActorOf(Props.Empty.WithRouter(new BroadcastGroup(ActorPaths.GithubCoordinat
                ActorPaths.GithubCoordinatorActor.Path + "2", ActorPaths.GithubCoordinatorActor.
        base.PreStart();
    }
```

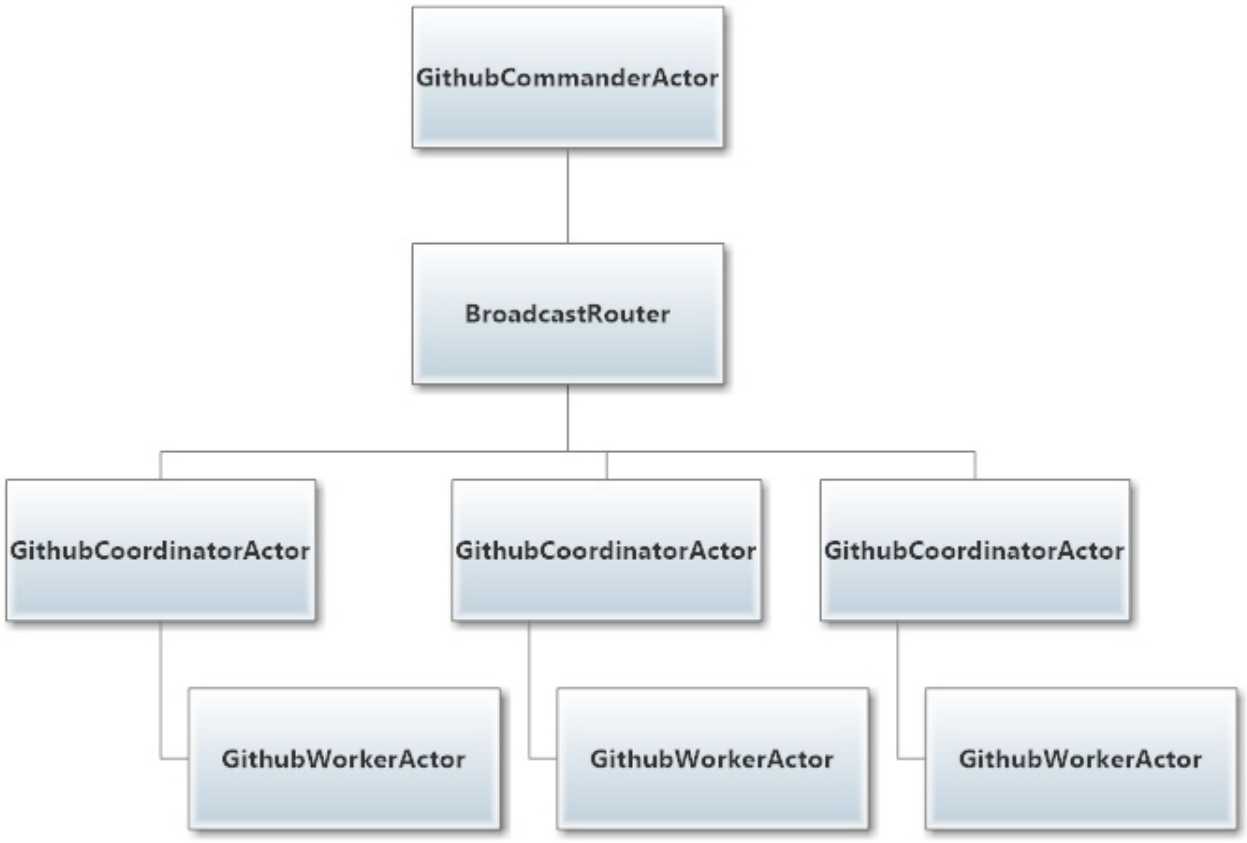
And with that, you're all set!

Once you're done

You should be able to run `GithubActors.sln` now and see that you can launch up to three jobs in parallel - a big improvement that didn't take very much code!



As a result of the changes you made, the actor hierarchy for GithubActors now looks like this:



Now we have 3 separate `GithubCoordinatorActor` instances who are all available for GitHub repository analysis jobs.

Great job!

Awesome job! You've successfully used Akka.NET routers to achieve the first layer of parallelism we're going to add to our GitHub scraper!

Let's move onto Lesson 2 - Using `Pool` routers to automatically create and manage pools of actors.

Any questions?

Don't be afraid to ask questions :).

Come ask any questions you have, big or small, in this ongoing Bootcamp chat with the Petabridge & Akka.NET teams.

Problems with the code?

If there is a problem with the code running, or something else that needs to be fixed in this lesson, please create an issue and we'll get right on it. This will benefit everyone going through Bootcamp.

Lesson 3.2: Using `Pool` routers to automatically create and manage pools of actors

In this lesson, we're going to build on the foundation we laid in the last lesson around routers and introduce you to a more powerful, flexible kind of router: the pool router.

Key Concepts / Background

What is a Pool router?

A "Pool" router is a `Router` that creates and manages its worker actors ("routees"). You provide a `NrOfInstances` to the router and the router will handle routee creation (and supervision) for you.

What's the difference between Group and Pool `Router`s?

The key difference is that pool routers create and manage their routees, whereas group routers do not.

Pool routers are safer to use than group routers. As touched on at the end of Lesson 3.1, group routers usually don't know when their routees are no longer available. This makes them less dependable and not ideal for common routing use.

This is because group routers are given the `ActorPath`s of their routees and in turn, communicate with their routees by sending messages to `ActorSelection`s. In contrast, because a pool router creates its routees, it is their parent, communicates directly with the `IActorRef` of a routee, and knows much more information about routees.

Additionally, pool routers can grow/shrink their routee pool whereas the routee pool for a group router is fixed once set. Also be aware that pool routers don't let you control the names of its routee children, so you have to talk to those routees via the router.

When should I use Group routers vs Pool routers?

You should use a pool router if you can.

We recommend using a pool router unless you have a unique situation that falls into one of these categories:

1. You need your router to be able to send messages to a group of actors via wildcard `ActorSelection`.
2. For some reason, you need to route messages to actors that the router cannot be responsible for. For example, there could be a situation where you need to front some actors with a router, but it just doesn't make sense to have those routees as children of the router. Perhaps if other critical components in your system depend on these actors living at a certain location.
3. You need your router to be able to send messages to different types of actor routees. (Pool routers can only have one type of actor as their routees.)

Frankly, we haven't come across many good use cases that call for a group router. You'll probably know it when you see it.

Our advice: stick with pool routers unless you have a very good reason to use a group router.

`RoutingStrategy`s

A pool router can use all of the routing strategies that a group router can use.

However, there are two `RoutingStrategy`s that **only** work with pool routers. The reason is that each of these strategies needs the level of routee control and information that only the pool router can offer, since it creates and supervises its routees.

Let's go through them.

`SmallestMailbox`

A `SmallestMailboxPoolRouter` will try to send the message to the routee with fewest messages in its mailbox.

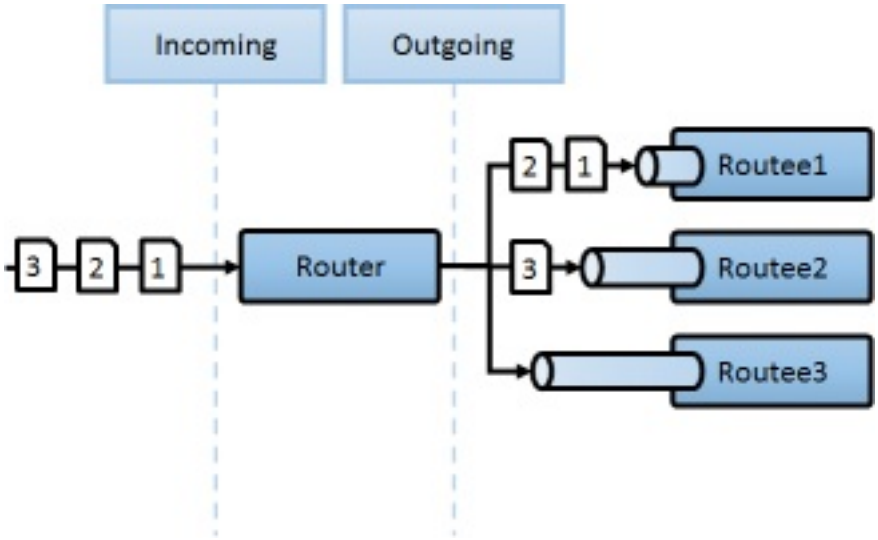
The selection is done in this order:

1. Pick any idle routee (not currently processing a message) with an empty mailbox

2. Pick any routee with an empty mailbox
3. Pick the routee with the fewest pending messages in mailbox
4. Pick any remote routee, remote actors are consider lowest priority, since their mailbox size is unknown

There is no Group router version of the `SmallestMailbox` because the information needed to execute the strategy is only practically available to a parent and not via an `ActorPath`.

Here's what the `SmallestMailbox` looks like:



`ResizableRouter`

We like to think of this as the "auto-scaling router".

A `ResizablePoolRouter` detects pressure on routee mailboxes and figures out if it needs to expand or contract the size of the routee pool.

Essentially, a `ResizableRouter` defines thresholds on the average mailbox load of its routees. Above this threshold, the router will add routee(s) to the pool to lower average pressure below the threshold. Below a different threshold, the router will remove routee(s) and reduce the size of the worker pool.

Special `Router` messages

You can send a pool router any of the special messages that you can send to a group router (`Broadcast`, `GetRoutees`, or `PoisonPill`). The function of these messages is the same across all routers.

Supervision & Pool Routers

How does supervision work with Pool routers?

A pool router supervises its routees.

Recall that a pool router creates its routees as direct child actors of the router. This means that a pool router automatically supervises (and `DeathWatch`es) its routees.

Also recall that `Router`s are just a special type of actor. Since `Router`s are actors, they have a `SupervisionStrategy` and can help their children (routees) recover from errors.

The supervision strategy of the router actor can be configured with the `supervisorStrategy` property of the Pool. If no configuration is provided, routers default to a strategy of "always escalate". This means that errors are passed up to the router's supervisor for resolution.

In this case, the router's supervising actor will treat the error as an error with the router itself. Therefore, a stop/restart directive issued would cause the router itself to stop/restart. The router, in turn, will cause its child routees to stop/restart (but it will maintain the number of routees in the pool).

What this means in practice

This is about default behavior in the case of an error.

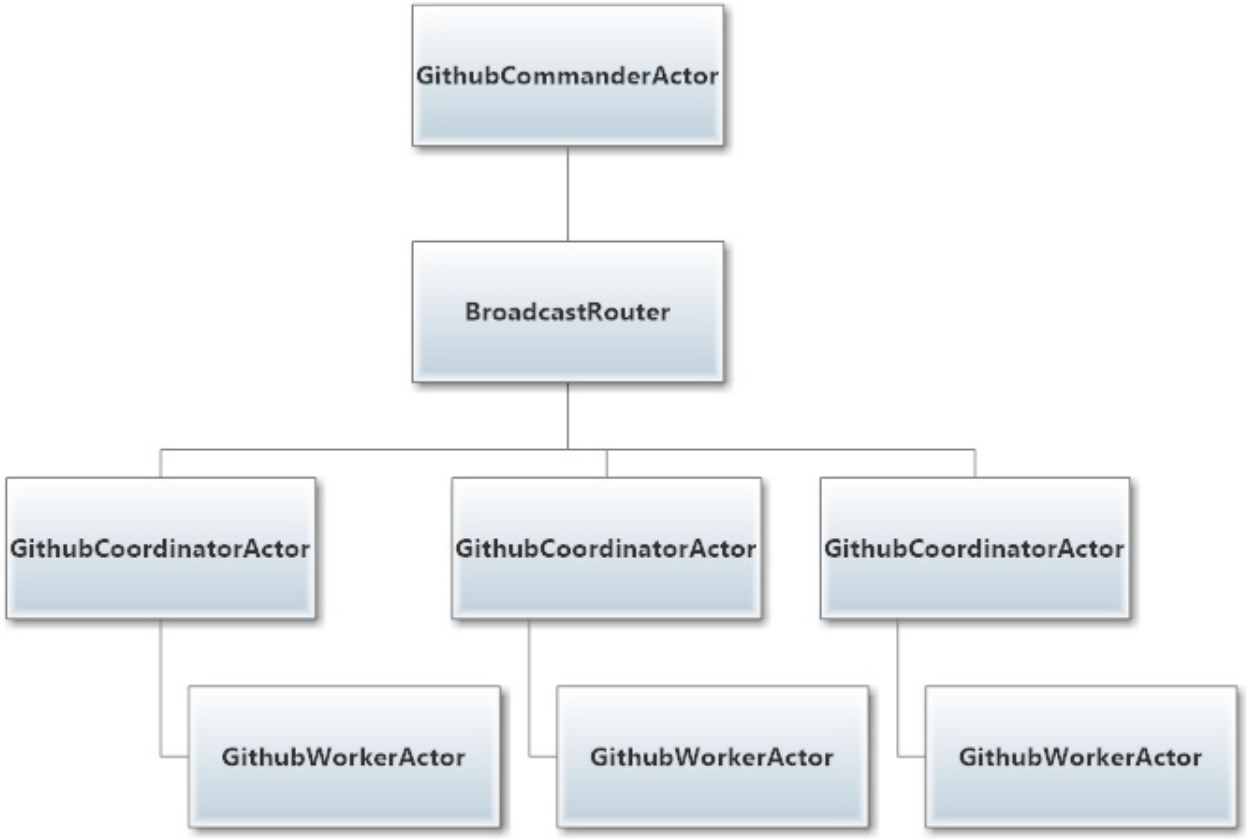
Here's what will happen, by default, if you haven't specified a `SupervisorStrategy` on the pool router:

1. A failure in a routee will bubble up to the parent of the router
2. The parent of the router will issue a `restart` directive to the router
3. The router will restart itself, and then restart its child routees

The reason is to make the default behavior such that adding `withRouter` to a child actor definition does not change the supervision strategy applied to the child. Of course, you can change this by specifying the strategy when defining the parent router.

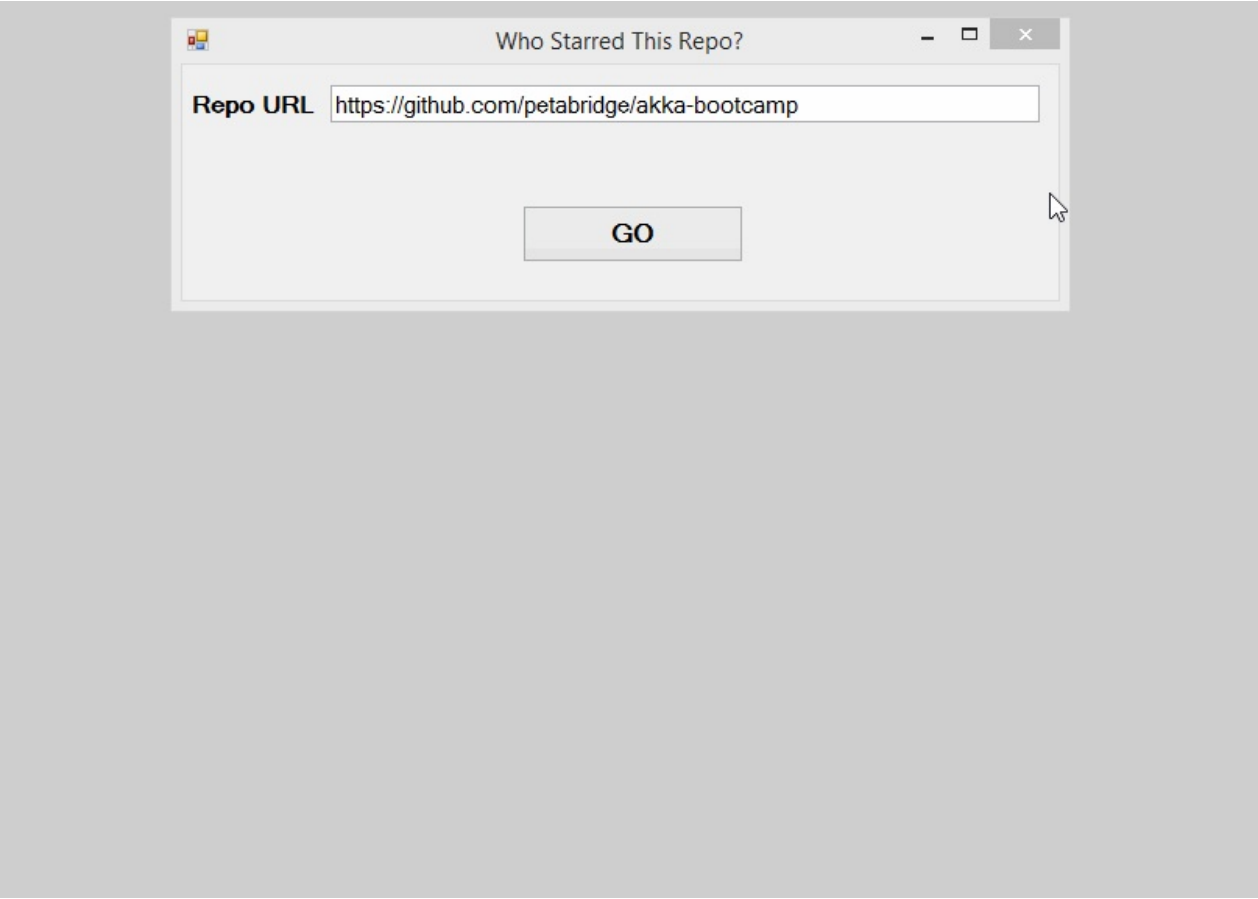
Exercise

The current state of our actor hierarchy looks like this:



In this exercise we're going to use a `RoundRobinPool` to throw additional `GithubWorkerActor` instances at our workloads, so we can increase the total throughput of the `GithubActors.sln` application.

This is how fast `GithubActors.sln` runs before we add our `Pool` router - take note:



Phase 1 - Modify `GithubCoordinatorActor.PreStart` to create `_githubWorker` as a router

This exercise consists of adding two lines of code.

First, open `Actors/GithubCoordinatorActor.cs` and add the following namespace import:

```
// add to the top of Actors/GithubCoordinatorActor.cs
using Akka.Routing;
```

And then change the `GithubCoorindatorActor.PreStart` method from this:

```
// original GithubCoordinatorActor.PreStart method from the start of the lesson
 protected override void PreStart()
{
    _githubWorker = Context.ActorOf(Props.Create(() => new GithubWorkerActor(GithubClientFac
}
```

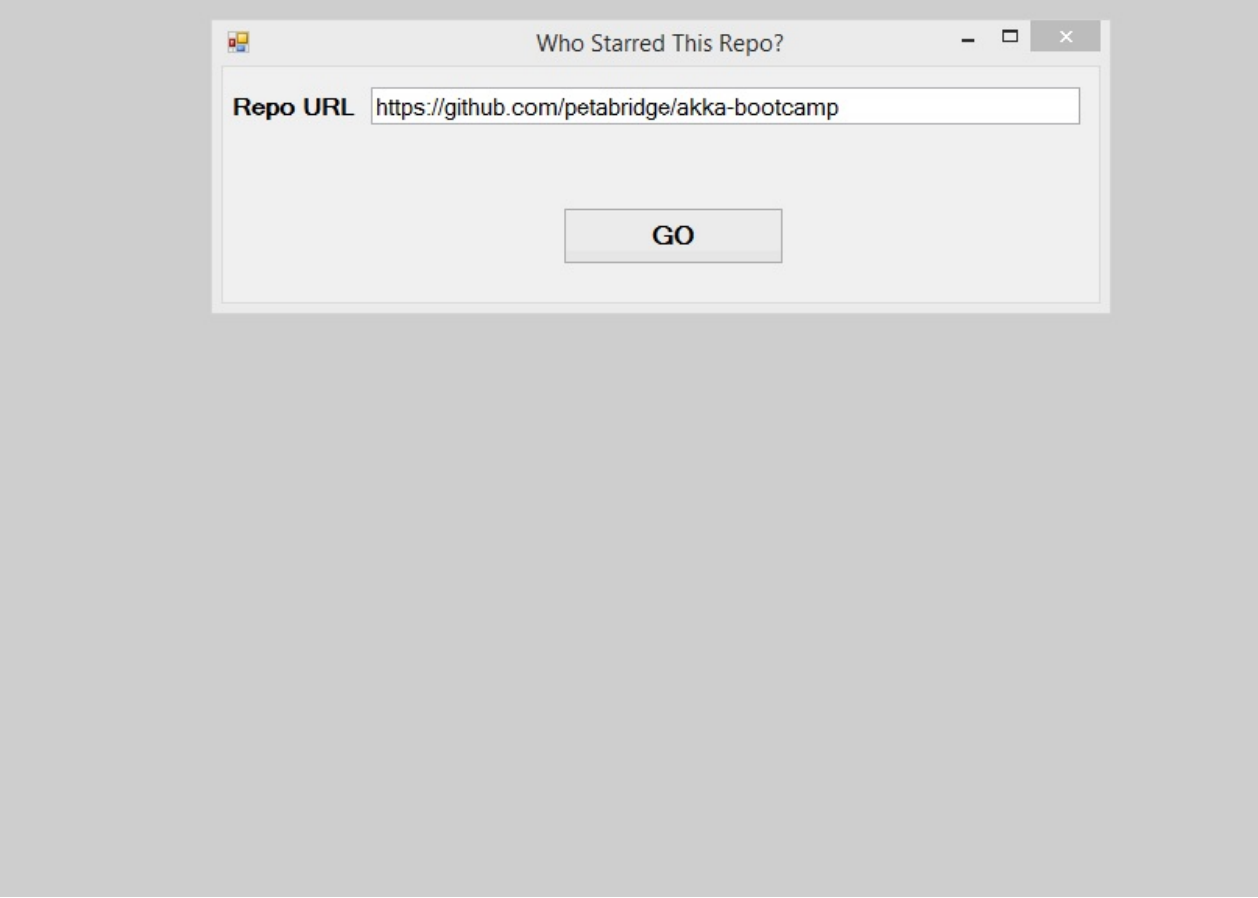
To this:

```
// CHANGE GithubCoordinatorActor.PreStart to use this code instead
protected override void PreStart()
{
    _githubWorker = Context.ActorOf(Props.Create(() => new GithubWorkerActor(GithubClientFac
        .WithRouter(new RoundRobinPool(10)));
}
```

That's it!

Once you're done

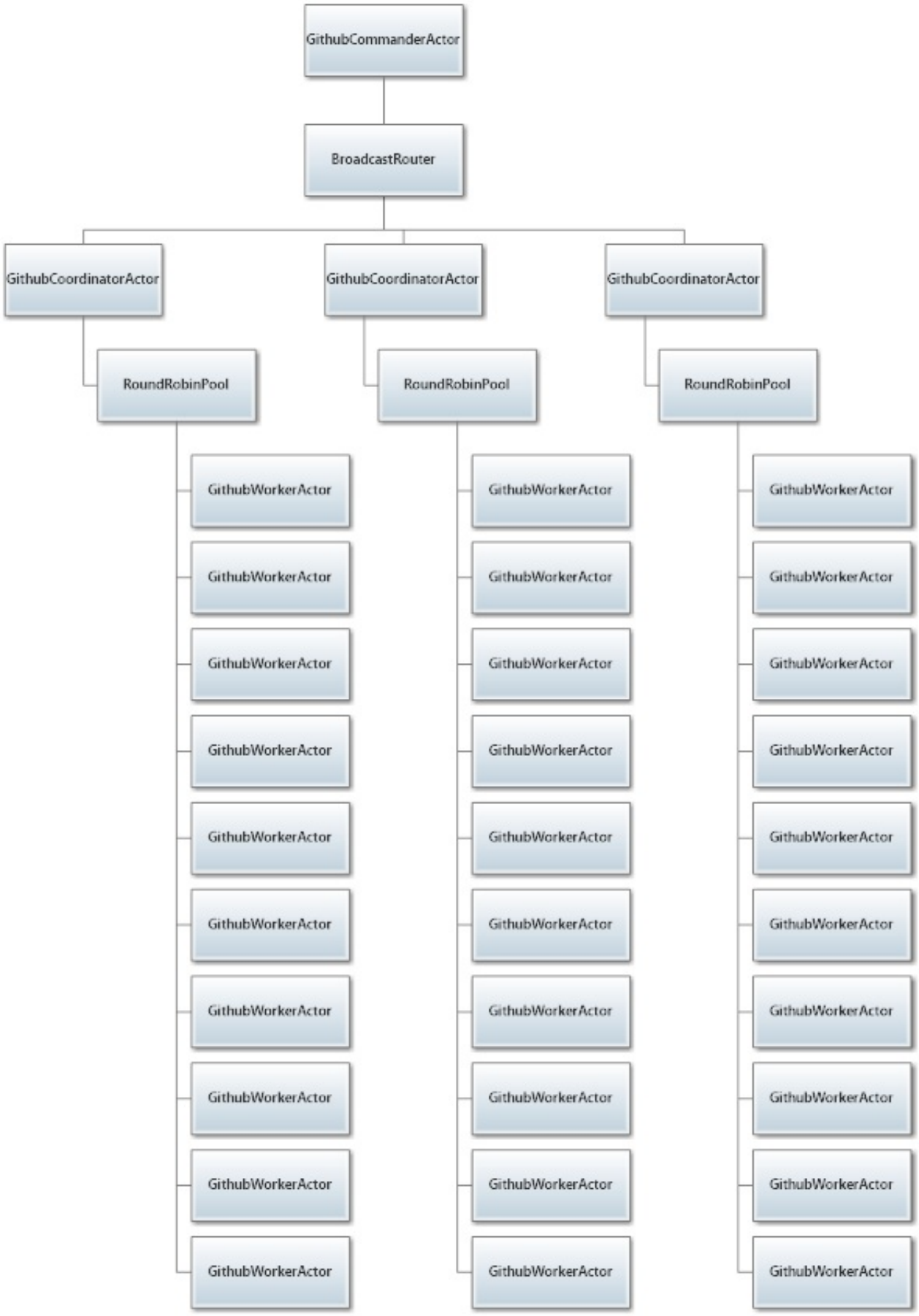
Build and run `GithubActors.sln`, and let's compare the performance of the app now that we're using 10 `GithubWorkerActor` instances per `GithubCoordinatorActor` instead of 1:



At the start of the lesson, it took us 16 seconds to download our first 4 users for https://github.com/petabridge/akka-bootcamp. **At the end of the lesson it took us less than 4 seconds**. And it only took two lines of code to do it - that's how easy it is to

parallelize work using actors.

Our final actor hierarchy at the end of the lesson looks like this:



Actors are cheap and easy to clone. Use them liberally with routers!

Great job!

Now that we've seen how both `Group` and `Pool` routers work, it's easy for us to add them in areas where we can benefit from parallelism or need scale-out for additional jobs.

Let's move onto How to use HOCON to configure your routers.

Any questions?

Don't be afraid to ask questions :).

Come ask any questions you have, big or small, in this ongoing Bootcamp chat with the Petabridge & Akka.NET teams.

Problems with the code?

If there is a problem with the code running, or something else that needs to be fixed in this lesson, please create an issue and we'll get right on it. This will benefit everyone going through Bootcamp.

Lesson 3.3: How to use HOCON to configure your routers

Awesome, look at you go! By now, you understand the massive increases in throughput that routers can give you, and what the different types of routers are.

Now we need to show you how to configure and deploy them :)

Key Concepts / Background

HOCON for `Router`s

Quick review of HOCON

We first learned about HOCON in Lesson 2.1.

To review, HOCON (Human-Optimized Config Object Notation) is a flexible and extensible configuration format. It will allow you to configure everything from Akka.NET's `IActorRefProvider` implementation, logging, network transports, and more commonly - how individual actors are deployed.

It's this last feature that we'll be using here to configure how our router actors are deployed. An actor is "deployed" when it is instantiated and put into service within the `ActorSystem` somewhere.

Why use HOCON to configure the routers?

There are three key reasons that we prefer using HOCON to configure our routers.

First, using HOCON keeps your code cleaner. By using HOCON you can keep configuration details out of your application code and keep a nice separation of concerns there. Helps readability a lot, too.

Second, like any actor, a router can be remotely deployed into another process. So if you want to remotely deploy a router (which you will), using HOCON makes that easier.

But most importantly, ***using HOCON means that you can change the behavior of actors dramatically without having to actually touch the actor code itself, just by changing config settings.***

What configuration flags usually specified?

What specific flags you need to specify will depend on the type of router you're using (e.g. you will need a `duration` with a `ScatterGatherFirstCompletedRouter`), but here are the things you'll be configuring the most.

Type of `Router`

The most common thing you'll specify is what the type of router is.

Here are the mappings between ´deployment.router' short names to fully qualified class names. You'll use these short names in `App.config`:

```
router.type-mapping {
  from-code = "Akka.Routing.NoRouter"
  round-robin-pool = "Akka.Routing.RoundRobinPool"
  round-robin-group = "Akka.Routing.RoundRobinGroup"
  random-pool = "Akka.Routing.RandomPool"
  random-group = "Akka.Routing.RandomGroup"
  balancing-pool = "Akka.Routing.BalancingPool"
  smallest-mailbox-pool = "Akka.Routing.SmallestMailboxPool"
  broadcast-pool = "Akka.Routing.BroadcastPool"
  broadcast-group = "Akka.Routing.BroadcastGroup"
  scatter-gather-pool = "Akka.Routing.ScatterGatherFirstCompletedPool"
  scatter-gather-group = "Akka.Routing.ScatterGatherFirstCompletedGroup"
  consistent-hashing-pool = "Akka.Routing.ConsistentHashingPool"
  consistent-hashing-group = "Akka.Routing.ConsistentHashingGroup"
}
```

Number of routees

The second most common flag you'll specify in HOCON is the number of routee instances to place under the router.

You do this with the `nr-of-instances` flag, like so:

```
akka {
    actor{
      deployment{
```

```
        /myRouter{
           router = broadcast-pool
           nr-of-instances = 3
         }
       }
    }
}
```

Resizer

To use a `ResizablePoolRouter` ("auto scaling router"), a `Resizer` component is required. This is the component that does the monitoring of routee mailbox load and compares that to the thresholds it has calculated.

Out of the box, there is only the default `Resizer`. You can configure your own if you want, but be forewarned, it's complicated. Which `Resizer` to use is commonly specified in HOCON, like so:

```
akka.actor.deployment {
/router1 {
    router = round-robin-pool
        resizer {
            enabled = on
            lower-bound = 2
            upper-bound = 3
        }
    }
}
```

What should I specify procedurally vs with HOCON?

The only thing we can think of that MUST be configured procedurally is the `HashMap` function given to a `ConsistentHashRouter`.

Everything else we can think of can be configured either way, but we prefer to do all our configuration via HOCON.

Which configuration wins: procedural, or HOCON?

HOCON wins. This is true for all actors, not just routers.

For example, if you procedurally specify config for a router and also configure the router

in `App.config`, then the values specified in HOCON win.

Suppose you defined the following router via configuration:

```
/router1 {
    router = consistent-hashing-pool
    nr-of-instances = 3
    virtual-nodes-factor = 17
}
```

But when it came time to create router1, you gave `Props` the following procedural router definition:

```
var router1 = MyActorSystem.ActorOf(Props.Create(() => new FooActor()).WithRouter(new RoundR
```

You'd still get a `ConsistentHashingPool` with 3 instances of `FooActor` instead of a `RoundRobinPool` with 10 instances of `FooActor`.

Forcing Akka.NET to load router definitions from configuration using `FromConfig`

Akka.NET won't create an actor with a router unless you explicitly call `WithRouter` during the time you create an actor who needs to be a router.

So, if we want to rely on the router definition we've supplied via configuration, we can use the `FromConfig` class to tell Akka.NET "hey, look inside our configuration for a router specification for this actor."

Here's an example:

```
/router1 {
    router = consistent-hashing-pool
    nr-of-instances = 3
    virtual-nodes-factor = 17
}
```

And if we make the following call to `ActorOf`:

```
var router1 = MyActorSystem.ActorOf(Props.Create(() => new FooActor()).WithRouter(FromConfig
```

Then we'll get a `ConsistentHashingPool` router.

Otherwise, if we just called this:

```
var router1 = MyActorSystem.ActorOf(Props.Create(() => new FooActor()), "router1");
```

Then we'd only get a single instance of `FooActor` in return. Use `FromConfig` whenever you need to use a router defined in configuration.

`Ask`

Bonus concept! We're also going to teach you to use `Ask` in addition to HOCON.

What is `Ask`?

`Ask` is how one actor can ask another actor for some information and wait for a reply.

When do I use `Ask`?

Whenever you want one actor to retrieve information from another and wait for a response. It isn't used that often—certainly not compared to `Tell()`—but there are places where it is ***exactly*** what you need.

Great! Let's put `Ask` and HOCON to work with our routers!

Exercise

We're not going to change the actor hierarchy much in this lesson, but we are going to replace the programmaticly defined `BroadcastGroup` router we created in lesson 1 with a `BroadcastPool` defined via HOCON configuration in `App.config`.

Phase 1 - Add new deployment to `App.config`

We'll add our configuration section first, before we modify the code inside `GithubCommanderActor`.

Open `App.config` and add the following inside the `akka.actor.deployment` section:

```
<!-- inside App.config, in the akka.actor.deployment section with all of the other HOCON -->
<!-- you can add this immediately after the /authenticator deployment specification -->
/commander/coordinator{
  router = broadcast-pool
  nr-of-instances = 3
}
```

Phase 2 - Modify `GithubCommanderActor` to use this new configuration setting

Open up `Actors/GithubCommanderActor.cs` and do the following:

Add the following import to the top of the file:

```
// add to the top of Actors/GithubCommanderActor.cs
using System.Linq;
```

Replace the `GithubCommanderActor`'s `BecomeAsking method to look like this:

```
// GithubCommanderActor's BecomeAsking method - replace it with this
private void BecomeAsking()
{
    _canAcceptJobSender = Sender;
    // block, but ask the router for the number of routees. Avoids magic numbers.
    pendingJobReplies = _coordinator.Ask<Routees>(new GetRoutees()).Result.Members.Count();
    Become(Asking);
}
```

Since the number of routees underneath `_coordinator` is now defined via configuration, we're going to `Ask<T>` the router using a built-in `GetRoutees` message to determine how many replies we need (1 per routee) before we can accept a new job. This is a special message that tells the router to return the full list of all of its current `Routees` back to the

sender.

`Ask` is usually an asynchronous operation, but in this case we're going to block and wait for the result - because the `GithubCommander` can't execute its next behavior until it knows how many parallel jobs can be run at once, which is determined by the number of routees.

> **NOTE: Blocking is not evil**. In the wake of `async` / `await`, many .NET developers have come to the conclusion that blocking is an anti-pattern or generally evil. This is ludicrous. It depends entirely on the context. Blocking is absolutely the right thing to do if your application can't proceed until the operation you're waiting on finishes, and that's the case here.

Finally, replace the `GithubCommanderActor`'s `PreStart` method with the following:

```
// replace GithubCommanderActor's PreStart method with this
protected override void PreStart()
{
    // create a broadcast router who will ask all of them if they're available for work
    _coordinator =
        Context.ActorOf(Props.Create(() => new GithubCoordinatorActor()).WithRouter(FromConf
        ActorPaths.GithubCoordinatorActor.Name);
    base.PreStart();
}
```

And that's it!

Once you're done

Build and run `GithubActors.sln` - you'll notice now that everything runs the same as it was before, *but* if you modify the `nr-of-instances` value in the deployment configuration for `/commander/coordinator` then it will directly control the number of parallel jobs you can run at once.

Effectively you've just made the number of concurrent jobs `GithubActors.sln` can run at once a configuration detail - cool!

Great job!

We've been able to leverage routers for parallelism both via explicit programmatic deployments and via configuration.

And now it's time to achieve maximum parallelism using the TPL in the next lesson: Lesson 4 - How to perform work asynchronously inside your actors using `PipeTo`

Any questions?

Don't be afraid to ask questions :).

Come ask any questions you have, big or small, in this ongoing Bootcamp chat with the Petabridge & Akka.NET teams.

Problems with the code?

If there is a problem with the code running, or something else that needs to be fixed in this lesson, please create an issue and we'll get right on it. This will benefit everyone going through Bootcamp.

Lesson 3.4: How to perform work asynchronously inside your actors using `PipeTo`

One of the first questions developers ask once they learn how Akka.NET actors work is…

> "If actors can only process one message at a time, can I still use `async` methods or `Task<T>` objects inside my actors?"

Yes! You can still use asynchronous methods and `Task<T>` objects inside your actors - using the `PipeTo` pattern (instead of using `await`)!

This lesson will show you how.

Key Concepts / Background

"But wait!", you say. "Aren't actors already asynchronous?"

Indeed they are, and you make an astute point! Due to the nature of passing immutable messages between actors, actors are inherently thread-safe and asynchronous (they don't block each other).

But what if you want to do some asynchronous work from within an actor itself, such as kick off a long-running HTTP request via a `Task`?

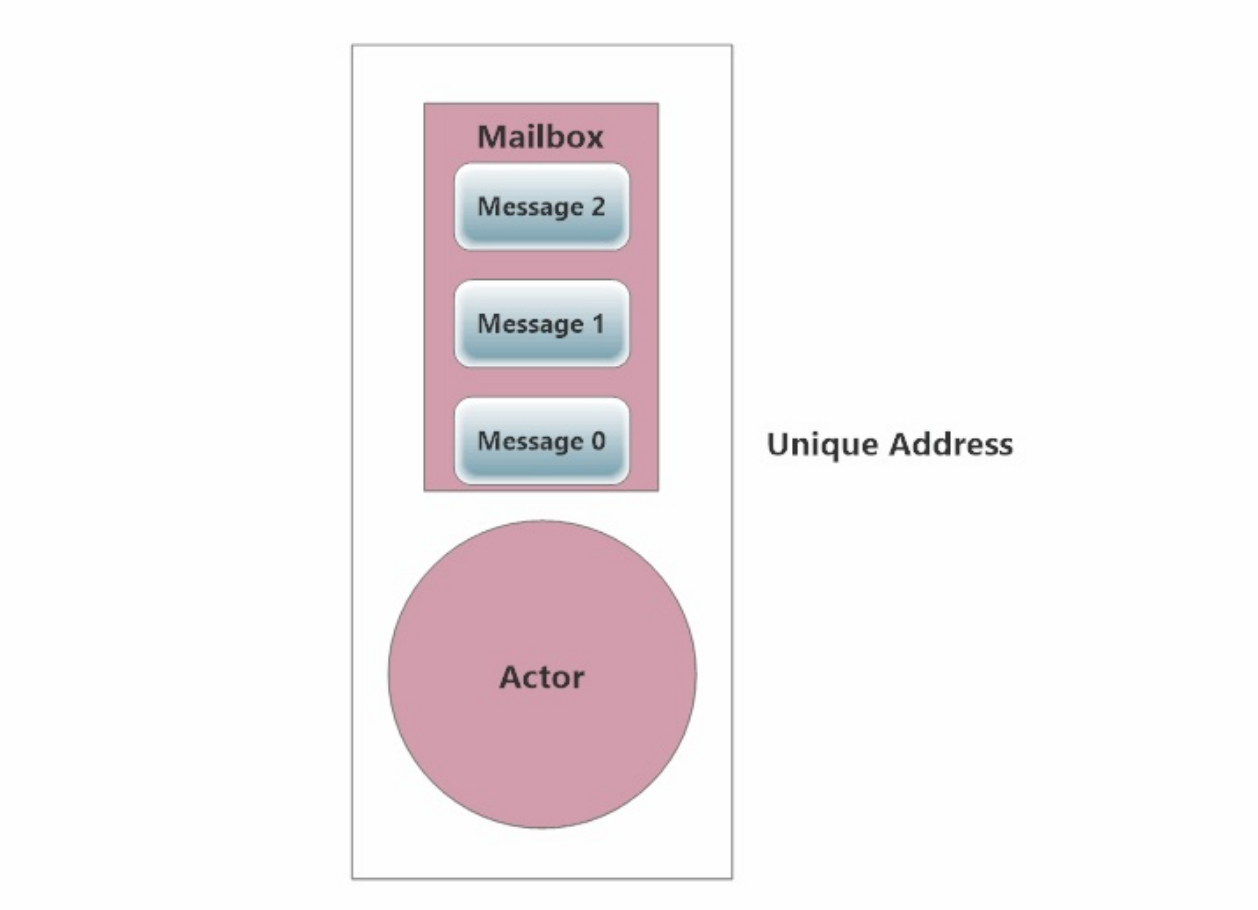
Most developers would default to using `await`, which has achieved demigod status since its release in 2012.

And they would be making the wrong choice.

Why? To answer that, we need to review how actors process messages.

Actors process messages one at a time

Actors process the contents of their mailbox one message at a time. It looks like this:



Why is maintaining this behavior critical?

Recall that immutable messages themselves are inherently thread-safe, since a different thread can't modify something that is immutable.

BUT: while the messages are inherently thread-safe, the message-processing code has no such guarantee!

Processing one message at a time is critical because making sure an actor's message processing code (`OnReceive`) can only be run *one invocation at a time* is how Akka.NET enforces thread-safety for all of the code that executes inside an actor.

An immutable message is pushed from the mailbox into `OnReceive`. Once the call to `OnReceive` exits, the actor's mailbox pushes a new message into the actor's `OnReceive` method.

That being said, it's still possible to take advantage of `async` methods and methods that return `Task<T>` objects inside the `OnReceive` method - you just have to use the `PipeTo` extension method!

Async message processing using `PipeTo`

The `PipeTo` pattern is a simple extension method built into Akka.NET that you can append to any `Task<T>` object.

```
public static Task PipeTo<T>(this Task<T> taskToPipe, ICanTell recipient, IActorRef sender =
```

`Task`s are just another source of messages

The goal behind `PipeTo` is to ***treat every async operation just like any other method that can produce a message for an actor's mailbox***.

THAT is the right way to think about actors and concurrent `Task<T>`s in Akka.NET. A `Task<T>` is not something you `await` on in Akka.NET. It's *just something else that produces a message* for an actor to process through its mailbox.

The `PipeTo` method takes an `ICanTell` object as a required argument, which tells the method where to pipe the results of an asynchronous `Task<T>`.

Here are all of the Akka.NET classes that you can use with `ICanTell`:

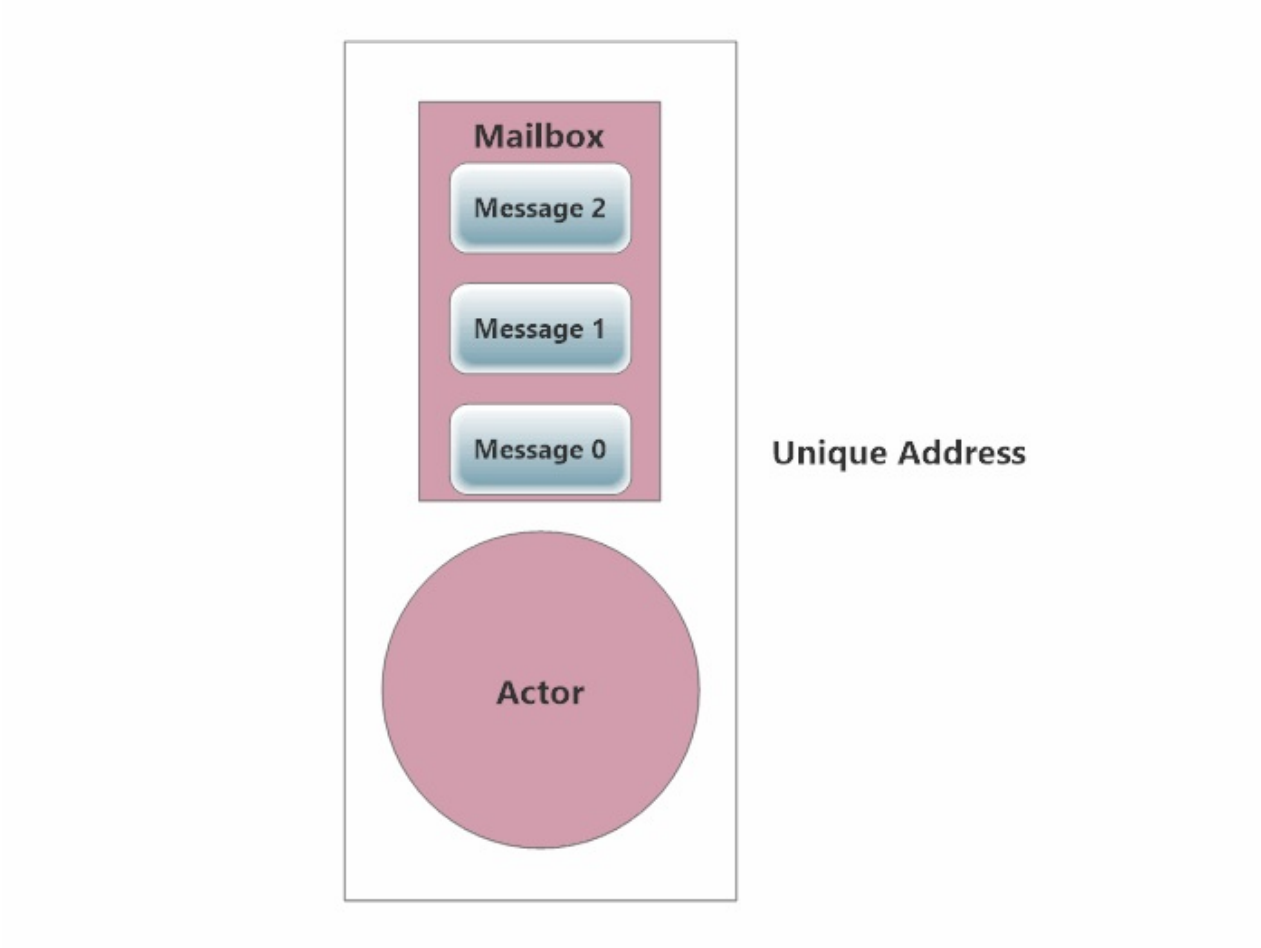
- `IActorRef` - a reference to an actor instance.
- `ActorSelection` - a selection of actors at a specified address. This is what gets returned whenever you look up an actor based on its path.

Most of the time, you're going to want to have your actors pipe the results of a task back to themselves. Here's an example of a real-world use case for `PipeTo`, drawn from our **official Akka.NET PipeTo code sample**.

```
// time to kick off the feed parsing process, and send the results to this same actor
Receive<BeginProcessFeed>(feed =>
{
    SendMessage(string.Format("Downloading {0} for RSS/ATOM processing...", feed.FeedUri));
    _feedFactory.CreateFeedAsync(feed.FeedUri).PipeTo(Self);
});
```

View the full source for this example..

Whenever you kick off a `Task<T>` and use `PipeTo` to deliver the results to some `IActorRef` or `ActorSelection`, here's how your actor is really processing its mailbox.



In this case we're using `PipeTo` to send the results back to itself, but you can just as easily send these results to different actor.

The important thing to notice in this animation is that the actor continues processing other messages while the asynchronous operation is happening.

That's why `PipeTo` is great for allowing your actors to parallelize long-running tasks, like HTTP requests.

Composing `Task<T>` instances using `ContinueWith` and `PipeTo`

Have some post-processing you need to do on a `Task<T>` before the result gets piped into an actor's mailbox? No problem - you can still use `ContinueWith` and all of the other

TPL design patterns you used in procedural C# programming.

Here's another example from our **PipeTo code sample**:

```
// asynchronously download the image and pipe the results to ourself
_httpClient.GetAsync(imageUrl).ContinueWith(httpRequest =>
{
    var response = httpRequest.Result;

    // successful img download
    if (response.StatusCode == HttpStatusCode.OK)
    {
        var contentStream = response.Content.ReadAsStreamAsync();
        try
        {
            contentStream.Wait(TimeSpan.FromSeconds(1));
            return new ImageDownloadResult(image, response.StatusCode, contentStream.Result)
        }
        catch //timeout exceptions!
        {
            return new ImageDownloadResult(image, HttpStatusCode.PartialContent);
        }
    }

    return new ImageDownloadResult(image, response.StatusCode);
},
  TaskContinuationOptions.AttachedToParent &
  TaskContinuationOptions.ExecuteSynchronously)
 .PipeTo(Self);
```

View the full source for this example..

So in this case, we're downloading an image via a HttpClient.aspx) inside an Akka.NET actor, and we want to check the status code of the HTTP response before we use `PipeTo` to deliver a message back to this actor.

So we do the HTTP code handling inside a `ContinueWith` block and use that to return an `ImageDownloadResult` message that will be piped to the actor using the `PipeTo` block. Pretty easy!

Why is `await` an Anti-pattern inside actors?

`await` is not magic, and breaks the core message processing guarantees

While `await` is a powerful and convenient construct, it isn't magic. It's just syntactic sugar for TPL continuation. If this is confusing or unfamiliar, we highly recommend reviewing Stephen Cleary's excellent Async/Await primer.

`Await` does two key things which break the core message processing guarantees of Akka.NET:

1. Exits the containing `async` function, while
2. Sets a continuation point in the containing method where the asynchronous `Task` (the `awaitable`) will return to and continue executing once it is done with its async work.

These actions by `await` have two negative effects:

First, `await` makes it harder to reason about exactly what is happening and on which thread. `ContinueWith` (which `await` is just syntactic sugar for anyway) makes it explicit and clear what is happening, and on which thread it's happening.

As we've discussed, an actor's mailbox pushes messages into the actor's `OnReceive` method as soon as the previous iteration of the `OnReceive` function exits. Whenever you `await` an `async` operation inside the `OnReceive` method, *you prematurely exit* the `OnReceive` method and the mailbox will push a new message into it, which is generally not what is intended.

Second, `await` breaks the "actors process one message at a time" guarantee. By the time the `await`ed `Task` returns and starts referencing things in the `ActorContext`, that context will have changed because the actor has moved on from the original message that `await`ed. Variables such as the `Sender` of the previous message may be different, or the actor might even be shutting down when the `await` call returns to the previous context.

So don't use `await` inside your actors. `Await` is evil inside an actor. `Await` is just syntactic sugar anyway. Use `ContinueWith` instead, and pair it with `PipeTo`.

This will turn the results of `async` operations into messages that get delivered to your actor's mailbox and you can take advantage of `Task` and other TPL methods just as you did before, and you'll enjoy nicely parallel processing!

Update: Akka.NET v1.0 now supports `async` / `await` inside

`ReceiveActor`

Per the Akka.NET v1.0 release notes, native support for `async` and `await` is now available inside `ReceiveActor`s.

```
public class MyActor : ReceiveActor
{
        public MyActor()
        {
              Receive<SomeMessage>(async some => {
                    //we can now safely use await inside this receive handler
                    await SomeAsyncIO(some.Data);
                    Sender.Tell(new EverythingIsAllOK());
              });
        }
}
```

There's some magic under the hood that takes care of this.

However, the `PipeTo` pattern is still the preferred way to perform async operations inside an actor, as it is more explicit and clearly states what is going on.

Do I need to worry about closing over (closures) my actor's internal state when using `PipeTo`?

Yes, you need to close over *any state whose value might change between messages* that you need to use inside your `ContinueWith` or `PipeTo` calls.

This usually means closing over the `Sender` property and any private state you've defined that is likely to change between messages.

For instance, the `Sender` property of your actor will definitely change between messages. You'll need to use a C# closure for this property in order to guarantee that any asynchronous methods that depend on this property get the right value.

Doing a closure is as simple as stuffing the property into an instance variable (`var`) and using that instance variable in your `PipeTo` call, instead of the field or property defined on your actor.

Here's an example of closing over the `Sender` property:

```
Receive<BeginProcessFeed>(feed =>
{
    // instance variable for closure
    // close over the current value of Sender, since it changes between
    // messages and accessing by property later would give different value
    var senderClosure = Sender;
    SendMessage(string.Format("Downloading {0} for RSS/ATOM processing...", feed.FeedUri));

    // send result of this async task back to the sender of the current message
    _feedFactory.CreateFeedAsync(feed.FeedUri).PipeTo(senderClosure);
});
```

> NOTE: Assuming you're piping the result of the `Task` back to the same actor, you don't need to close over `Self` or `Parent`. Those `IActorRef`s will be the same when the `Task` returns. You just need to close over the state that is going to change by the time the `Task` completes and executes its continuation delegate.

Now, let's get to work and use this powerful parallelism technique inside our actors!

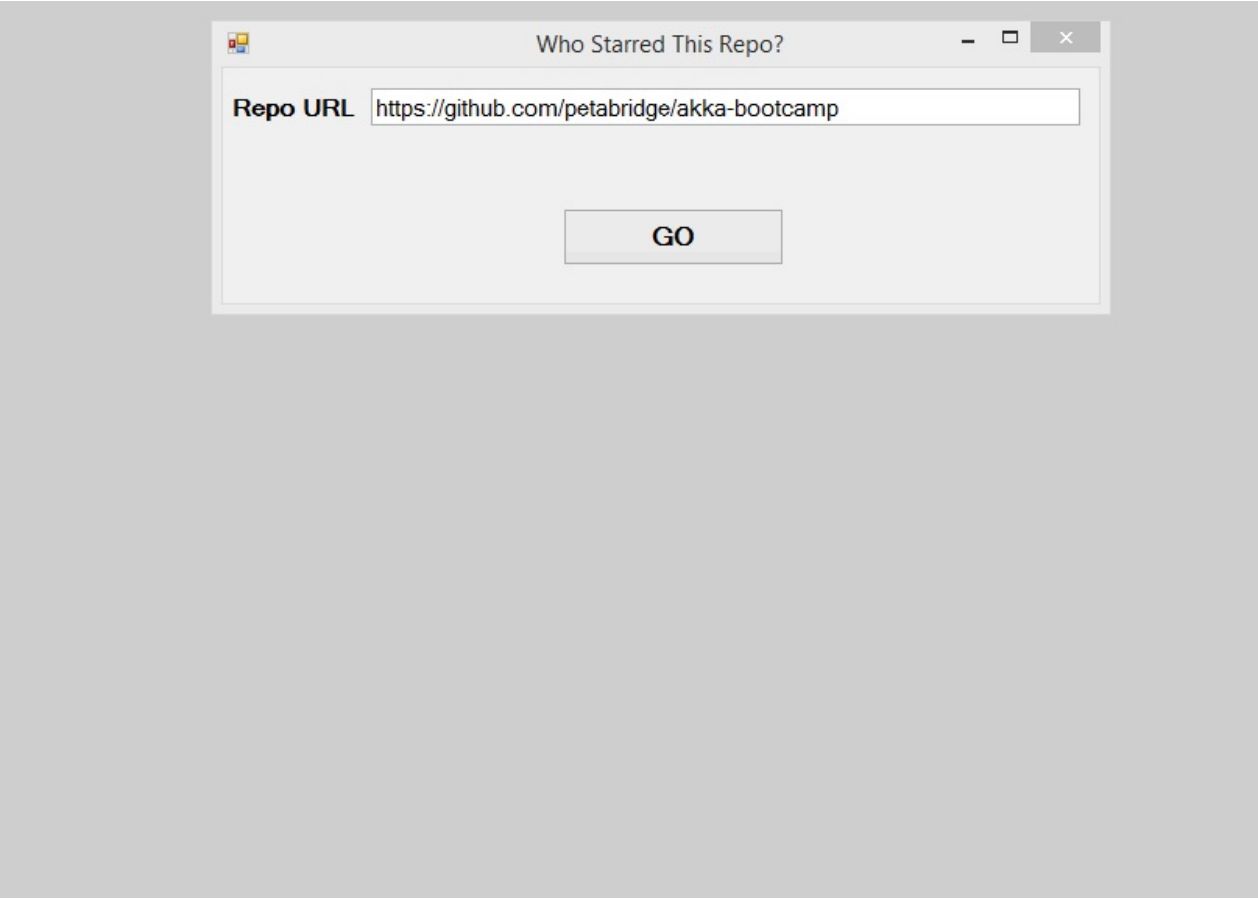
Exercise

Currently our `GithubWorkerActor` instances all block when they're waiting for responses back from the GitHub API, using the following code:

```
 var getStarrer = _gitHubClient.Activity.Starring.GetAllForUser(starrer);

// ewww
getStarrer.Wait();
var starredRepos = getStarrer.Result;
Sender.Tell(new StarredReposForUser(starrer, starredRepos));
```

We're going to leverage the full power of the TPL and allow each of our `GithubWorkerActor` instances kick off multiple parallel Octokit queries at once, and then use `PipeTo` to asynchronously deliver the completed results back to our `GithubCoordinatorActor`.

Take note - this the current speed of our GitHub scraper at the end of lesson 2:



Phase 1 - Replace `GithubWorkerActor.InitialReceives`

Open up `Actors/GithubWorkerActor.cs` and replace the `InitialRecieves` method with the following code:

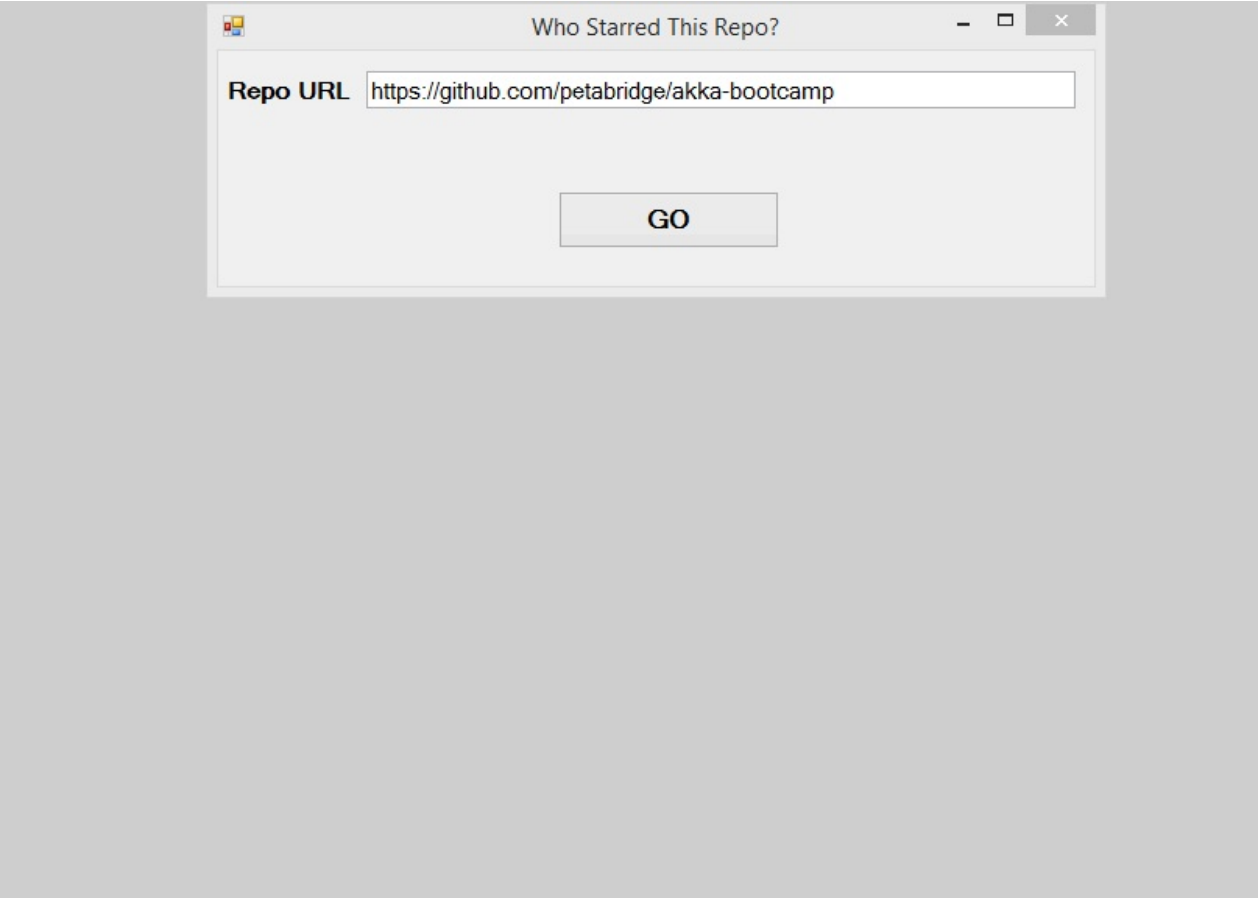
```
private void InitialReceives()
{
    // query an individual starrer
    Receive<RetryableQuery>(query => query.Query is QueryStarrer, query =>
    {
        // ReSharper disable once PossibleNullReferenceException
        // (we know from the previous IS statement that this is not null)
        var starrer = (query.Query as QueryStarrer).Login;

        // close over the Sender in an instance variable
        var sender = Sender;
        _gitHubClient.Activity.Starring.GetAllForUser(starrer).ContinueWith<object>(tr =>
        {
            // query faulted
            if (tr.IsFaulted || tr.IsCanceled)
                return query.NextTry();
            // query succeeded
            return new StarredReposForUser(starrer, tr.Result);
        }).PipeTo(sender);
```

```
        });

        // query all starrers for a repository
        Receive<RetryableQuery>(query => query.Query is QueryStarrers, query =>
        {
            // ReSharper disable once PossibleNullReferenceException
            // (we know from the previous IS statement that this is not null)
            var starrers = (query.Query as QueryStarrers).Key;


            // close over the Sender in an instance variable
            var sender = Sender;
            _gitHubClient.Activity.Starring.GetAllStargazers(starrers.Owner, starrers.Repo)
                .ContinueWith<object>(tr =>
                {
                    // query faulted
                    if (tr.IsFaulted || tr.IsCanceled)
                        return query.NextTry();
                    return tr.Result.ToArray();
                }).PipeTo(sender);

        });
    }
```

That's it!

Once you're done

Build and run `GithubActors.sln` - the performance should be *really fast* now.



At the start of the lesson, it took us 4 seconds to download our first 4 users for https://github.com/petabridge/akka-bootcamp. **At the end of the lesson we downloaded 22 users in 4 seconds**. All of this without adding any new actors or doing anything other than just letting the TPL work in concert via `PipeTo`.

> **NOTE:** The GitHub API appears to be *really* slow for a handful of users on every repository we've tested. We have no idea why.

Great job!

Awesome - now you can use `Task<T>` instances in combination with your actors for maximum concurrency! Hooray!

Now it's time to move onto the final lesson: Lesson 5 - How to prevent deadlocks with `ReceiveTimeout`.

Further reading

See our full Akka.NET `PipeTo` sample.

Any questions?

Don't be afraid to ask questions :).

Come ask any questions you have, big or small, in this ongoing Bootcamp chat with the Petabridge & Akka.NET teams.

Problems with the code?

If there is a problem with the code running, or something else that needs to be fixed in this lesson, please create an issue and we'll get right on it. This will benefit everyone going through Bootcamp.

Lesson 3.5: How to prevent deadlocks with `ReceiveTimeout`

Wow, look at you! Here we are on our last lesson of Bootcamp together. We want to say thank you for coming on this journey with us, and to give yourself a big pat on the back for your dedication to your craft.

In this lesson, we'll be going over how to handle timeouts within actors and prevent deadlocks, where one actor is waiting for another indefinitely.

This lesson will show you how to prevent deadlocks by using a `RecieveTimeout`.

Key Concepts / Background

What is `ReceiveTimeout`?

`ReceiveTimeout` lets you specify what an actor should do when it hasn't received a message for a certain period of time. Once this timeout has been hit, the actor will send itself the `ReceiveTimeout` singleton as a message.

Once set up, the `ReceiveTimeout` stays in effect and will continue firing repeatedly every time the specified interval passes without the actor receiving a message.

When do I use `ReceiveTimeout`?

You can use `ReceiveTimeout` whenever you want to take some action after a period of inactivity.

Here are some common cases where you may want to use a `ReceiveTimeout`:

- To shut an actor down after it goes a certain amount of time without receiving a message
- To confirm that other actors are doing work and sending in their status messages
- To prevent deadlocks where one actor thinks another is doing work

How do I set up a `ReceiveTimeout`?

You call `Context.SetReceiveTimeout()` and pass it a `TimeSpan`. If that amount of time passes and the actor hasn't received a message, the actor will send itself the `ReceiveTimeout` singleton as a message, e.g.

```
// send ourselves a ReceiveTimeout message if no message within 3 seconds
Context.SetReceiveTimeout(TimeSpan.FromSeconds(3));
```

Then, you just need to handle the `ReceiveTimeout` message and take whatever action is appropriate.

Let's assume you wanted to shut an actor down after a period of inactivity, and inform its parent. Here's one basic way you could do that:

```
// have actor shut down after a long period of inactivity
Receive<ReceiveTimeout>(timeout =>
{
    // inform parent that shutting down
    Context.Parent.Tell(new ImShuttingDown());
    // shut self down
    Context.Stop(Self);
});
```

How do I cancel a `ReceiveTimeout` that I've set?

You call `Context.SetReceiveTimeout()` and pass it `null`, e.g.

```
// cancel ReceiveTimeout
Context.SetReceiveTimeout(null);
```

Can I change the timeout value?

Yes, you can set the `ReceiveTimeout` after every message, or as often as you want. Setting a new timeout will cancel the previous timeout and schedule a new one.

What's the smallest `ReceiveTimeout` interval I can

specify?

1 millisecond is the minimum timeout interval.

Does `ReceiveTimeout` work with all actor types?

Yes, you can use `ReceiveTimeout` with any actor type.

The only difference would be the syntax differences between how you match the message between a `ReceiveActor` and an `UntypedActor`.

What if another message comes in right before I process the timeout?

`ReceiveTimeout` can create false positives. For example, it's possible for the timeout to occur and another message to arrive in the actors mailbox before the `ReceiveTimeout` message does. In this case, another message would get processed before the `ReceiveTimeout` message, making it invalid.

It is not *guaranteed* that upon reception of the `ReceiveTimeout` that there must have been an idle period beforehand as configured via this method.

This is an edge case, but there are ways to code around it.

Exercise

We're going to use `ReceiveTimeout` to eliminate a potential deadlock that might occur inside the `GithubCommanderActor` - if one of the `GithubCoordinatorActor` it routes to suddenly dies before it has a chance to reply to a `CanAcceptJob` message, the `GithubCommanderActor` will be permanently stuck in its `Asking` state.

We can prevent this from happening using `ReceiveTimeout`!

Phase 1 - Add a new private field to the `GithubCommanderActor`

We're going to hang onto the current job we're inquiring about as an instance variable

inside the `GithubCommanderActor`, so open up `Actors/GithubCommanderActor.cs` and make the following changes:

```
// add this field anywhere inside the GithubCommanderActor
private RepoKey _repoJob;
```

And modify the `GithubCommanderActor.Ready` method to look like this:

```
// modify the GithubCommanderActor.Ready method to look like this
private void Ready()
{
    Receive<CanAcceptJob>(job =>
    {
        _coordinator.Tell(job);
        _repoJob = job.Repo;
        BecomeAsking();
    });
}
```

Phase 2 - Wire up `ReceiveTimeout` inside `GithubCommanderActor`

We need to set a few calls to `Context.ReceiveTimeout` in order to get it to work properly with our `GithubCommanderActor` when we're inside the `Asking` state.

First, modify the `BecomeAsking` method on the `GithubCommanderActor` to look like this:

```
// modify the `BecomeAsking` method on the `GithubCommanderActor` to look like this
private void BecomeAsking()
{
    _canAcceptJobSender = Sender;
    // block, but ask the router for the number of routees. Avoids magic numbers.
    pendingJobReplies = _coordinator.Ask<Routees>(new GetRoutees()).Result.Members.Count();
    Become(Asking);

    // send ourselves a ReceiveTimeout message if no message within 3 seonds
    Context.SetReceiveTimeout(TimeSpan.FromSeconds(3));
}
```

This means that once the `GithubCommanderActor` enters the `Asking` behavior, it will

automatically send itself a `ReceiveTimeout` message if it hasn't received any other message for longer than three seconds.

Speaking of which, let's add a handler for the `ReceiveTimeout` message type inside the `Asking` method on `GithubCommanderActor`.

```
// add this inside the GithubCommanderActor.Asking method
// means at least one actor failed to respond
Receive<ReceiveTimeout>(timeout =>
{
    _canAcceptJobSender.Tell(new UnableToAcceptJob(_repoJob));
    BecomeReady();
});
```

We're going to treat every `ReceiveTimeout` as a "busy" signal from one of the `GithubCoordinatorActor` instances, so we'll send ourselves a `UnableToAcceptJob` message every time we receive a `ReceiveTimeout`.

Once the `GithubCommanderActor` has received all of the replies its expecting and it switches back to its `Ready` state, we need to cancel the `ReceiveTimeout`.

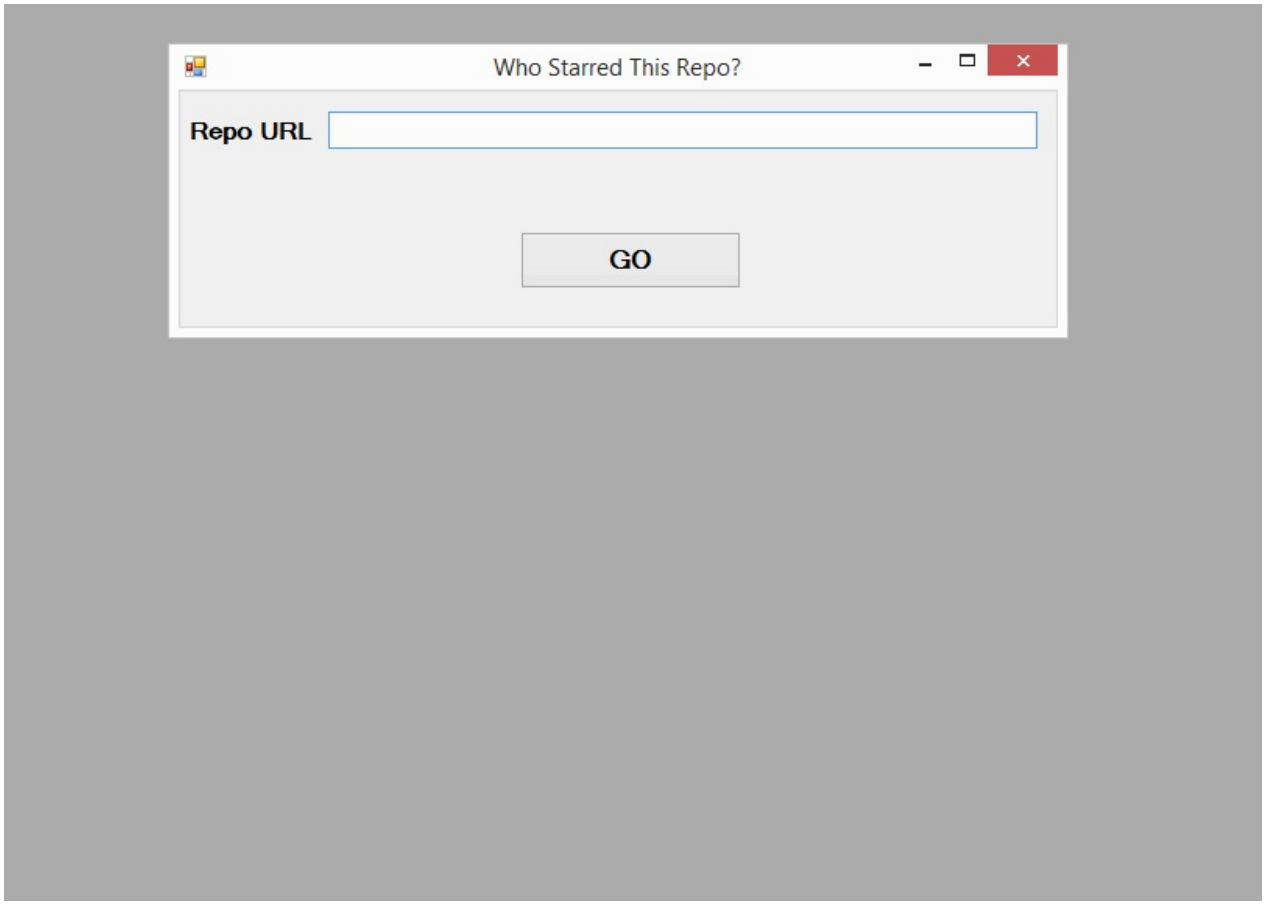
Modify the `GithubCommanderActor`'s `BecomeReady` method to look like the following:

```
// modify the GithubCommanderActor.BecomeReady method to read like the following:
private void BecomeReady()
{
    Become(Ready);
    Stash.UnstashAll();

    // cancel ReceiveTimeout
    Context.SetReceiveTimeout(null);
}
```

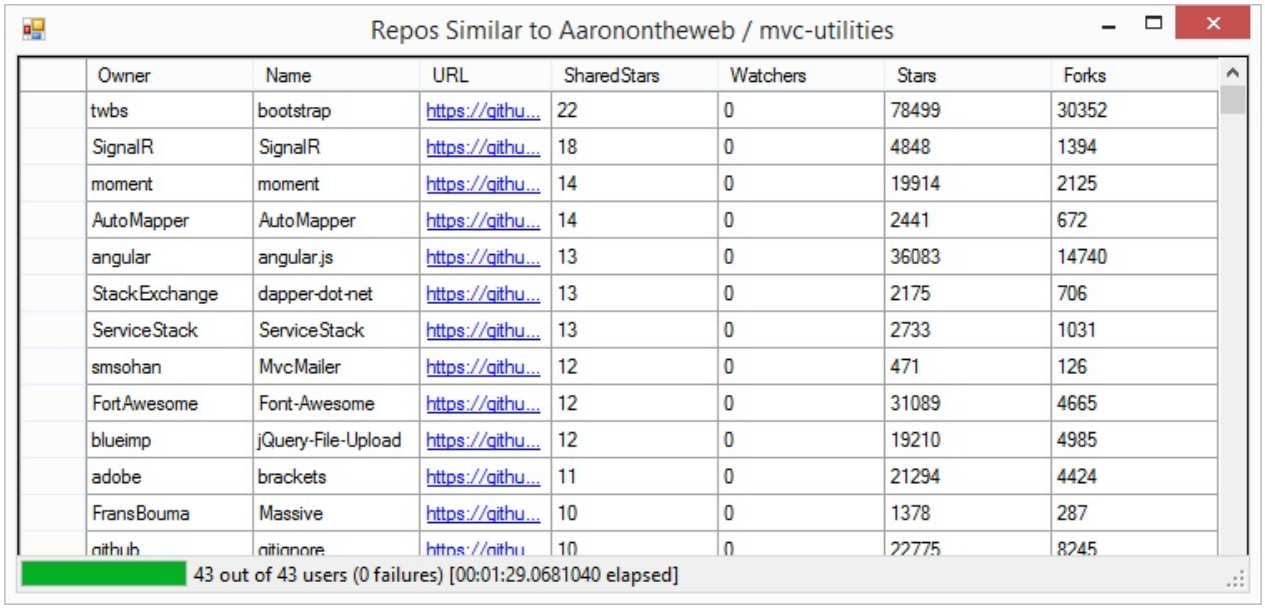
And that's it!

Once you're done

Build and run `GithubActors.sln`, and you should see the following output if you try querying the Akka.NET GitHub Repository (go give them a star while you're at it!)



And here's what the final output looks like - sadly, for a different repo since hit the GitHub API rate limit with Akka.NET :(



Great job!

Wow! You made it, awesome!

We're really proud of you, and want to express our gratitude for sticking with us all the way through. Thank you, and kudos to you. Your dedication to your craft inspires us.

Sharing is caring: click here to Tweet about Bootcamp! (you can edit first)

We want to help more people get this knowledge and learn to use Akka.NET. Direct them to the Bootcamp information page or to this repo.

If we at Petabridge can be of any help to you whatsoever, please reach out to us by email or say hello on Twitter.

Want to level up your company or team with Akka.NET?

Please email us to discuss your situation.

We work with companies all the time to **implement production systems and do advanced Akka.NET training** (Clustering, Remoting, Testing, DevOps, best practices, etc).

We'd love to help you, too.

Gratefully,
Aaron & Andrew
Petabridge co-founders

Any questions?

Don't be afraid to ask questions :).

Come ask any questions you have, big or small, in this ongoing Bootcamp chat with the Petabridge & Akka.NET teams.

Problems with the code?

If there is a problem with the code running, or something else that needs to be fixed in this lesson, please create an issue and we'll get right on it. This will benefit everyone

going through Bootcamp.