



[Nueva guía] Descargue la Guía 2017 de Microservicios: Rom...

[Descargar la guía](#) ▶

Microservicios en la JVM con actores

por Markus Eisele MVB · 12 y 17 de diciembre · Zona de Java

Recién publicado, un libro gratuito de O'Reilly sobre Reactive Microsystems: The Evolution of Microservices at Scale . Presentado en asociación con Lightbend.

Este artículo aparece en la nueva Guía DZone para Microservicios. ¡Obtenga su copia gratis para artículos más perspicaces, estadísticas de la industria y más!

A medida que las aplicaciones móviles y basadas en datos dominan cada vez más, los usuarios exigen acceso en tiempo real a todo en cualquier lugar. La flexibilidad y la capacidad de respuesta del sistema ya no son "agradables de tener"; son requisitos comerciales esenciales. Las empresas necesitan cada vez más intercambiar desde arquitecturas estáticas y centralizadas a favor de sistemas flexibles, distribuidos y elásticos.

Pero por dónde empezar y qué enfoque de arquitectura utilizar es aún un poco borroso, y la publicidad de microservicios se está asentando lentamente mientras que la industria del software explora varias arquitecturas y estilos de implementación.

Durante una década o más, los equipos de desarrollo empresarial han desarrollado sus proyectos Java EE dentro de contenedores de servidores de aplicaciones grandes y monolíticos sin tener en cuenta el ciclo de vida individual de su módulo o componente. Conectarse a los eventos de inicio y cierre fue simple, ya que acceder a otros componentes era solo una instancia inyectada. Era relativamente fácil mapear objetos en bases de datos relacionales individuales o conectarse a otros sistemas a

través de mensajes. Una de las mayores ventajas de esta arquitectura era la transaccionalidad, que era sincrónica, fácil de implementar y fácil de visualizar y monitorear.

Al mantener la fuerte modularidad y la separación de componentes como una prioridad de primera clase, fue manejable implementar los sistemas más grandes que todavía alimentan nuestro mundo. Trabajar con módulos de compartimentación e introducción pertenece a las habilidades básicas de los arquitectos. Nuestra industria ha aprendido cómo combinar servicios y construirlos en torno a las capacidades organizacionales.

La parte nueva en las arquitecturas basadas en microservicios es la forma en que los servicios verdaderamente independientes se distribuyen y se conectan entre sí. Crear un servicio individual es fácil. Construir un sistema a partir de muchos es el verdadero desafío, porque nos introduce en el espacio problemático de los sistemas distribuidos. Esta es la principal diferencia con las infraestructuras clásicas y centralizadas.

No hay una sola forma de hacer microservicios

Hay muchas maneras de implementar una arquitectura basada en microservicios en o alrededor de la Máquina Virtual Java (JVM). La pirámide en la Figura 1 fue presentada en mi primer libro. Categoriza algunas tecnologías en capas, lo que puede ayudar a identificar el nivel de aislamiento que se necesita para un sistema basado en microservicios.

Comenzando en la infraestructura de virtualización con máquinas virtuales y contenedores, ya que son medios de aislar aplicaciones del hardware, subimos por la pila hasta algo que resumí bajo el nombre de "servicios de aplicaciones". Esta categoría contiene marcos de microservicios específicos destinados a proporcionando soporte de microservicios en todo el ciclo de vida de desarrollo de software.



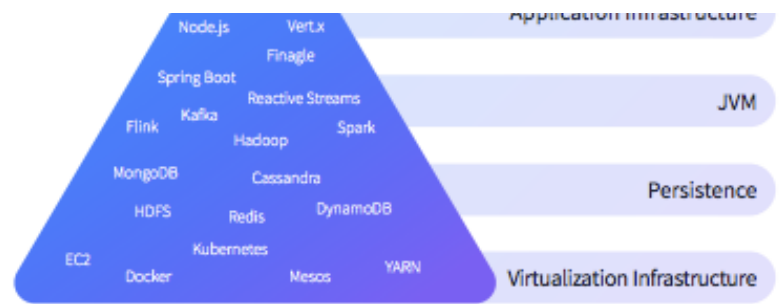


Figura 1: Pirámide del desarrollo de Java empresarial moderno (Fuente: Modern Java EE Design Patterns , Eisele)

Los tres marcos en las categorías de infraestructura y servicios de aplicaciones se basan en los principios del Manifiesto reactivo . Define rasgos que conducen a sistemas grandes que se componen de sistemas más pequeños, que son más flexibles, poco flexibles y escalables. Como están esencialmente orientados a mensajes y distribuidos, estos marcos se ajustan a los requisitos de las arquitecturas de microservicios actuales.

Si bien Lagom ofrece un enfoque dogmático sobre barandillas cerradas que solo admiten arquitecturas de microservicios, Play y Akka le permiten aprovechar los rasgos reactivos para construir un sistema al estilo de los microservicios, pero no lo limita a este enfoque.

Microservicios con Akka

Akka es un kit de herramientas y tiempo de ejecución para crear aplicaciones controladas por mensajes altamente concurrentes, distribuidas y resistentes en la JVM. Los "actores" Akka son una de las herramientas en el kit de herramientas de Akka que le permiten escribir código concurrente sin tener que pensar en los hilos y bloqueos de bajo nivel. Otras herramientas incluyen Akka Streams y Akka HTTP. Aunque Akka está escrito en Scala, también hay una API de Java.

Los actores fueron inventados hace décadas por Carl Hewitt . Pero, relativamente recientemente, su aplicabilidad a los desafíos de los sistemas informáticos modernos ha sido reconocida y comprobada como efectiva. El modelo de actor proporciona una abstracción que le permite pensar en su código en términos de comunicación, no a diferencia de las personas en una organización grande.

Los sistemas basados en el modelo de actor que usa Akka se pueden diseñar con una resistencia increíble. El uso de jerarquías de supervisor significa que la cadena parental de componentes es responsable de detectar y corregir fallas, dejando que los clientes se preocupen solo por el servicio que requieren.

A diferencia del código escrito en Java que arroja excepciones, los clientes de los servicios basados en actores nunca se preocupan por tratar las fallas del actor desde el que están solicitando un servicio. En cambio, los clientes solo deben comprender el contrato de solicitud y respuesta que tienen con un servicio determinado, y posiblemente reintentar las solicitudes si no se da respuesta en algún período de tiempo. Cuando las personas hablan de microservicios, se enfocan en la parte "micro", diciendo que un servicio debe ser pequeño.

Quiero enfatizar que lo importante a tener en cuenta al dividir un sistema en servicios es encontrar los límites correctos entre los servicios, alineándolos con contextos delimitados, capacidades comerciales y requisitos de aislamiento. Como resultado, un sistema basado en microservicios puede alcanzar sus requisitos de escalabilidad y resiliencia, lo que facilita su implementación y administración. La mejor manera de entender algo es mirar un ejemplo. La documentación de Akka contiene una amplia guía de una aplicación de gestión de IoT simplista que permite a los usuarios consultar los datos de los sensores. No expone ninguna API externa para simplificar las cosas, solo se centra en el diseño de la aplicación y utiliza una API basada en actores para que los dispositivos informen sus datos a la parte de administración. Puede encontrar un diagrama de arquitectura de alto nivel en la Figura 2.

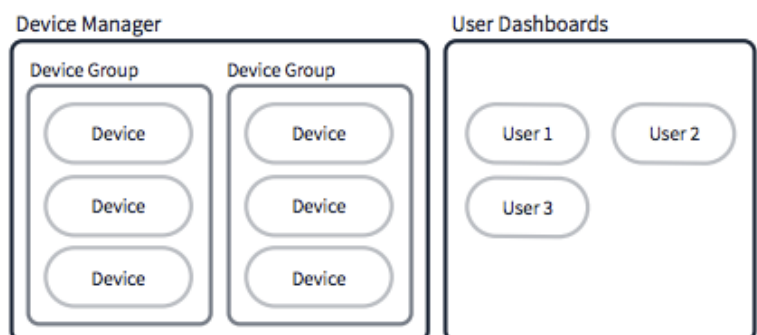


Figura 2: Arquitectura de aplicación de ejemplo de IoT

(Fuente: documentación de Akka)

Los actores están organizados en un árbol estricto, donde el ciclo de vida de cada niño está ligado a los padres, y donde los padres son responsables de decidir el destino de los niños reprobados. Todo lo que necesita hacer es reescribir su diagrama de arquitectura para que contenga cuadros anidados en un árbol, como se muestra en la Figura 3.

En términos simples, cada componente administra el ciclo de vida de los subcomponentes. Ningún subcomponente puede sobrevivir al componente principal. Así es exactamente como funciona la jerarquía de actores. Además, es deseable que un componente maneje la falla de sus subcomponentes. Una relación de componentes "contenida" se asigna a la relación de "actores secundarios" de los actores.

Si observa las arquitecturas de microservicio, habría esperado que los componentes de nivel superior también sean los actores de nivel superior. Eso es de hecho posible, pero no recomendado. Como no tenemos que volver a conectar los servicios individuales a través de protocolos externos y el marco Akka también gestiona el ciclo de vida del actor, podemos crear un solo actor de alto nivel en el sistema actor y modelar los servicios principales como hijos de este actor. La arquitectura del actor se basa en los mismos rasgos en los que debe basarse una arquitectura de microservicio, que son aislamiento, autonomía, responsabilidad única, estado exclusivo, comunicación asíncrona, protocolos de comunicación explícitos y transparencia de distribución y ubicación.

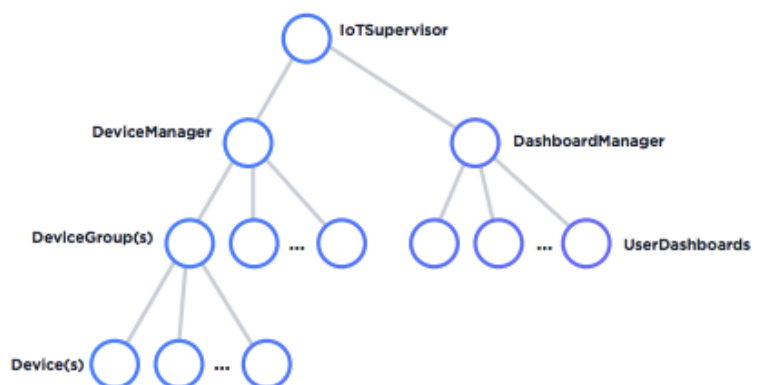


Figura 3: Representación de un actor de la arquitectura IoT.

Encontrará los detalles sobre cómo implementar las

clases `IoTSupervisor` y `DeviceManager` en el tutorial oficial de Akka . Hasta ahora, solo miraba el sistema completo en general. Pero también está el actor individual que representa un dispositivo. Su tarea simple será recopilar las mediciones de temperatura e informar los últimos datos medidos a solicitud. Al trabajar con objetos, normalmente se diseñan API como interfaces, que son básicamente colecciones de métodos abstractos que debe completar la implementación real. En el mundo de los actores, las contrapartes de las interfaces son protocolos. El protocolo en una aplicación basada en el actor es el mensaje para los dispositivos.

```

1  contador de función ( estado : AppState = 0 ,
2  static final class ReadTemperature {
3      long requestId ;
4      ReadTemperature público ( long requestId )
5          esta . requestId = requestId ;
6      }
7  }
8  pública estática última clase RespondTemperat
9      long requestId ;
10     Valor opcional de < Double > ;
11     RespondTemperature público ( long requestId
12         valor ) {
13         esta . requestId = requestId ;
14         esta . valor = valor ;
15     }
16 }

```

Código 1: protocolo de mensaje para el actor del dispositivo

Me estoy saltando una gran cantidad de antecedentes sobre pedidos de mensajes y garantías de entrega . Diseñar un sistema con la suposición de que los mensajes se pueden perder en la red es la forma más segura de construir una arquitectura basada en microservicios. Esto se puede hacer, por ejemplo, implementando una funcionalidad de "reenviar" si se pierde un mensaje. Y esta es la razón por la cual el mensaje también contiene un ID de solicitud. Ahora será responsabilidad del actor consultar para hacer coincidir

las solicitudes con los actores. Un primer boceto del Actor del dispositivo está debajo.

```

El dispositivo de clase extiende AbstractActor
1  ◀────────────────────────────────────────▶
2      // ...
   Optional < Double > lastTemperatureReading
3  ◀────────────────────────────────────────▶
4      @Anular
5      public void preStart () {
        log . información ( "Dispositivo actor {
6  ◀────────────────────────────────────────▶
           started " , groupId , deviceId );
7  ◀────────────────────────────────────────▶
8      }
9      @Anular
10     public void postStop () {
        log . información ( "Dispositivo actor {
11  ◀────────────────────────────────────────▶
           stopped " , groupId , deviceId );
12  ◀────────────────────────────────────────▶
13     }
14     @Anular
        // reacciona a los mensajes recibidos de Read
15  ◀────────────────────────────────────────▶
16     public Receive createReceive () {
        return receiveBuilder ()
17         . match ( ReadTemperature . class , r
18  ◀────────────────────────────────────────▶
           getSender (). tell ( new Respond
19  ◀────────────────────────────────────────▶
           ))
20         . construir ();
21     }
22 }
23 }

```

Código 2: El actor del dispositivo

La temperatura actual se establece inicialmente en `Optional.empty ()` y simplemente se informa cuando se consulta. Una prueba simple para el dispositivo se muestra a continuación.

```

1  @Prueba
   public void testReplyWithEmptyReadingIfNoTemper
2  ◀────────────────────────────────────────▶
       TestKit probe = new TestKit ( sistema );
3  ◀────────────────────────────────────────▶
       ActorRef deviceActor = sistema . actorOf (
4  ◀────────────────────────────────────────▶

```

```
5      deviceActor . tell ( nuevo Dispositivo . Rea
6      Dispositivo . RespondTemperature response =
7      expectMsgClass ( Device . RespondTemperature
8      assertEquals ( 42 L , respuesta . RequestID
9      assertEquals ( Opcional . empty (), response
10 }
```

Código 3: probar el dispositivo actor

El ejemplo completo del sistema IoT se encuentra en la documentación de Akka .

Por dónde empezar

La mayoría del software empresarial de hoy en día se construyó hace años y aún se somete a un mantenimiento regular para adoptar las últimas regulaciones o nuevos requisitos comerciales. A menos que haya un caso de negocios completamente nuevo o una reestructuración interna significativa, la necesidad de reconstruir un software desde cero casi nunca se brinda.

Si este es el caso, se lo conoce comúnmente como desarrollo "greenfield", y usted es libre de seleccionar el marco base de su elección. En un escenario de "brownfield", solo desea aplicar la nueva arquitectura a un área determinada de una aplicación existente. Ambos enfoques ofrecen riesgos y desafíos, y hay defensores de ambos. El terreno común para ambos escenarios es su conocimiento del dominio comercial.

Especialmente en proyectos empresariales existentes y de larga ejecución, esta podría ser la ruta crítica. Tienden a ser escasos en la documentación, y es aún más importante tener acceso a los desarrolladores que trabajan en este dominio y tienen conocimiento de primera mano.

El primer paso es una evaluación inicial para identificar qué partes de una aplicación existente pueden aprovechar una arquitectura de microservicios. Hay varias maneras de hacer esta evaluación inicial. Sugiero pensar en las características del servicio. Primero desea

identificar los servicios centrales o de proceso.

Mientras que los servicios centrales son componentes modelados después de sustantivos o entidades, los servicios de proceso ya contienen lógica de flujo o negocio complejo.

Mejoras selectivas

El enfoque de migración más libre de riesgos es solo agregar mejoras selectivas. Al raspar las partes identificadas en uno o más servicios y agregar el pegamento necesario a la aplicación original, puede escalar áreas específicas de su aplicación en varios pasos.

El patrón estrangulador

Creado por primera vez por Martin Fowler como la aplicación Strangler, los candidatos de extracción se mueven a un sistema separado que se adhiere a una arquitectura de microservicios, y las partes existentes de las aplicaciones permanecen intactas. Un equilibrador de carga o proxy decide qué solicitudes deben llegar a la aplicación original y cuáles van a las nuevas. Hay algunos problemas de sincronización entre las dos pilas. Lo más importante es que no se puede permitir que la aplicación existente cambie las bases de datos de los microservicios.

Big Bang: Refactorizar un sistema existente

En casos muy raros, la refactorización completa de la aplicación original puede ser el camino correcto. Es raro porque las aplicaciones empresariales necesitarán mantenimiento continuo durante la refactorización completa.

Además, no habrá tiempo suficiente para detenerse por completo durante un par de semanas, o incluso meses, dependiendo del tamaño de la aplicación, para reconstruirla en una nueva pila. Este es el enfoque menos recomendado porque conlleva un riesgo comparablemente alto de falla.

Cuándo no usar microservicios

Los microservicios son la elección correcta si tiene un sistema que es demasiado complejo como para ser manejado como un monolito. Y esto es exactamente lo que hace que este estilo arquitectónico sea una opción válida para aplicaciones empresariales.

Como afirma Martin Fowler en su artículo sobre "Microservice Premium", el punto principal es ni siquiera considerar el uso de una arquitectura de microservicios a menos que tenga un sistema demasiado grande y complejo para ser construido como un simple monolito. Pero también es cierto que hoy en día, los procesadores multinúcleo, la computación en la nube y los dispositivos móviles son la norma, lo que significa que los sistemas completamente nuevos son sistemas distribuidos desde el principio.

Y esto también resulta en un mundo completamente diferente y más desafiante para operar. El paso lógico ahora es cambiar el pensamiento de la colaboración entre objetos en un sistema a una colaboración de sistemas de escalamiento individual de microservicios.

Resumen

El modelo de actor proporciona un mayor nivel de abstracción para escribir sistemas concurrentes y distribuidos, lo que protege al desarrollador contra el bloqueo explícito y la administración de subprocesos. Proporciona la funcionalidad central de los sistemas reactivos, definidos en el Manifiesto reactivo como receptivos, flexibles, elásticos y basados en mensajes. Akka es un marco basado en actores que es fácil de implementar con soporte completo de Java 8 Lambda. Los actores permiten a los desarrolladores diseñar e implementar sistemas de manera que ayuden a enfocarse más en la funcionalidad central y menos en la plomería. Los sistemas basados en actores son la base perfecta para las arquitecturas de microservicios que evolucionan rápidamente.

Este artículo aparece en la nueva Guía DZone para Microservicios. ¡Obténla su copia gratis para artículos

... más perspicaces, estadísticas de la industria y más!

Estrategias y técnicas para crear microservicios escalables y resistentes para refactorizar una aplicación monolítica paso a paso, un libro gratuito de O'Reilly .
Presentado en asociación con Lightbend.

Temas: JAVA, JVM, MICROSERVICIOS, MODELO DE ACTOR, TUTORIAL

Las opiniones expresadas por los contribuidores de DZone son suyas.

Obtenga lo mejor de Java en su bandeja de entrada.

Manténgase actualizado con DZone's Bi-weekly Java Newsletter. [VER UN EJEMPLO](#)

[SUSCRIBIR](#)

Java Partner Resources

Get Started with Spring Boot, OAuth 2.0, and Okta
Okta



Ensuring Visibility into Microservices and Containers
AppDynamics



Advanced Linux Commands [Cheat Sheet]
Red Hat Developer Program



Modern Java EE Design Patterns: Building Scalable Architecture for Sustainable Enterprise Development
Red Hat Developer Program



3 Tips for Managing Scope Creep

by Scott Colin... Dec 12, 17 - Agile Zone

Managing an app project for a client, whether large or small, is no easy task. Keeping the ship on course while it's pulled about by the currents and winds of changing client demands, budget constraints and your looming deadlines can be a rough job, especially when the ugly sea monster known as scope creep rears its head.

The Standish Group's 2017 CHAOS Summary found that fewer than a third of projects (29%) were successful, meaning they were delivered on-time, on-budget, and with all the required features.

Of the remaining projects, 52% percent were delivered late, over budget or missing features and 19% failed completely, either through being canceled or delivered and never used. That means that in 2015, 71% of all IT projects were either failed or challenged.

And the main reason for these failures? You guessed it, that nasty old sea monster, scope creep.

What Is Scope Creep?

Scope creep, otherwise known as feature creep or "kitchen sink syndrome" (as in, including everything and the kitchen sink), is basically where, over the course of the project, more and more features are added to the point where the allotted budget, time, and resources can no longer cover the workload and the project starts to flounder.

It happens for various reasons, but the two main causes are:

- Clients asking for more features.
- "Gold plating" – the situation where the development team decides to add more features in order to impress the client (or themselves).

Both of these things usually happen slowly, a small change here, a suggestion there, but they quickly build up into a formidable extra workload that can crush the project or grind it to a standstill.

Don't let the thought of an unsuccessful project get you down, here are three tips for managing scope creep to keep your project development as smooth as possible:

1. Scope Creep Will Happen. Own it.

First things first: scope creep is going to happen. You're not going to get around it. So it's good to start with that assumption.

When you start working with a client, take time to define end results. That means, not only the required features of the app but also the budget, the time the project should take, and the resources that will be required to make sure the project is delivered within these constraints.

This is usually broken down into the “project triangle” of Scope/Time + Budget/Resources. If any one of these sides changes then so must the others to accommodate. Because of this, getting clear requirements is essential, as it covers the “scope” side of the triangle, and once that is in place the rest can be worked out.

When working with a client, good negotiation is essential so that you both understand exactly what the project is aiming to deliver, and, before the project begins, establish a formalized way for the client to request changes or new features, no matter how minor.

A great way to make sure that both you and your client are on the same page is by asking them the right questions before you even think about building their app. Then, when you and the client are ready, codify exactly which features are going to be in App 1.0 – the Minimum Viable Product.

By doing this, you not only manage to establish a way to organize and prioritize requests but also force the client to recognize whenever they are asking for something new, which can help with requests for tiny adjustments that are perhaps trivial.

2. Sometimes It's Ok to 'Just Say No'

When you're working for a client, it can be very hard to break the mindset of “the customer is always right.” It's heavily ingrained in the customer service industry, no matter their area, and it's an almost expected part of

customer interaction

In the case of project management though, it's not always true.

The customer may make a request for a change that to them seems like a minor job, but you know that it will take a few days at least and be a major headache. Rather than heed their every whim, it's your job to negotiate with the customer and to try to change their mind, especially if the change is, to you and your team, unnecessary.

That's not to say you should utterly discount their idea, and especially not rudely, but by showing them that either through time or budget constraints that it's not possible without changing the project plan, they are more likely to back down or be willing to take a watered down version of the idea.

It's also in your best interests to provide a professional counterbalance to your customer's ideas and demands. If they've, say, requested that an app is to be developed for both iOS and Android after you have already started developing it for iOS, it's up to you to convince them that within your project's constraints, it is a much better idea to continue on a single platform. Show your customer that you and your team know your stuff and can provide good reasons as to why it is better to follow your advice. Remember, your customer doesn't necessarily have to like you. It helps, but would you rather have a reputation for being "nice guys" who are average quality or the guys that always deliver great products but are more strict?

At the end of the day, it's about profit for the customer; and if your app makes them a good profit, they'll be much more willing to turn a blind eye to you not always heeding to their every whim.

3. Know What You're Getting Into

Even with all of the above suggestions, it's likely you'll find yourself working on a few extra bits and pieces with the project; like we said before, scope creep will happen. The important part is to always research every request before you agree to do it. Here's a good example: a client makes a simple request that you migrate the app database backend from your servers to theirs. Simple.

right? But what your client forgot to tell you is that they're running on Windows servers, while your app agency hosts everything on Linux. A simple request could turn into a week-long nightmare.

The same rule applies to teams on their own projects. If someone suddenly has a great sounding idea, always make sure to do as much research into how realistic making it happen would actually be.


It's something that can be seen frequently in the games industry, where developers invest millions and millions of dollars into adding a feature into their games at the last minute that ends up pushing the game way over their deadlines and over their budget. This then snowballs as then they have to sell more of the game to recoup their losses and it makes it more likely the game, and all too commonly, the studio will fail.

Realise though, that scope creep isn't just about vetoing obviously bad customer or fellow developer ideas. A lot of the time it is a two-way street. Someone suggests a feature idea and then, through either wanting to prove your worth or also being excited about the idea, you, as the project manager, allow the idea to be added to the scope of the project. Then, before you know it, you're over budget, over time, and wondering just where the hell it all went off the rails.

Managing Scope Creep

Scope creep is an animal all projects, regardless of size, have to face. It can kill them dead if it's allowed to, but with careful management and by keeping a firm grasp of the actual requirements of the project, it can be tamed. It can even become useful, providing ideas and features that take a project from mediocre to something special. It's a hard line to walk, but, hopefully, these tips will help you keep it under control.

Topics: AGILE, SCOPE CREEP, MINIMUM VIABLE PRODUCT, AGILE TEAMS

Published at DZone with permission of Scott Calonico.
[See the original article here.](#) 

Opinions expressed by DZone contributors are their own.

