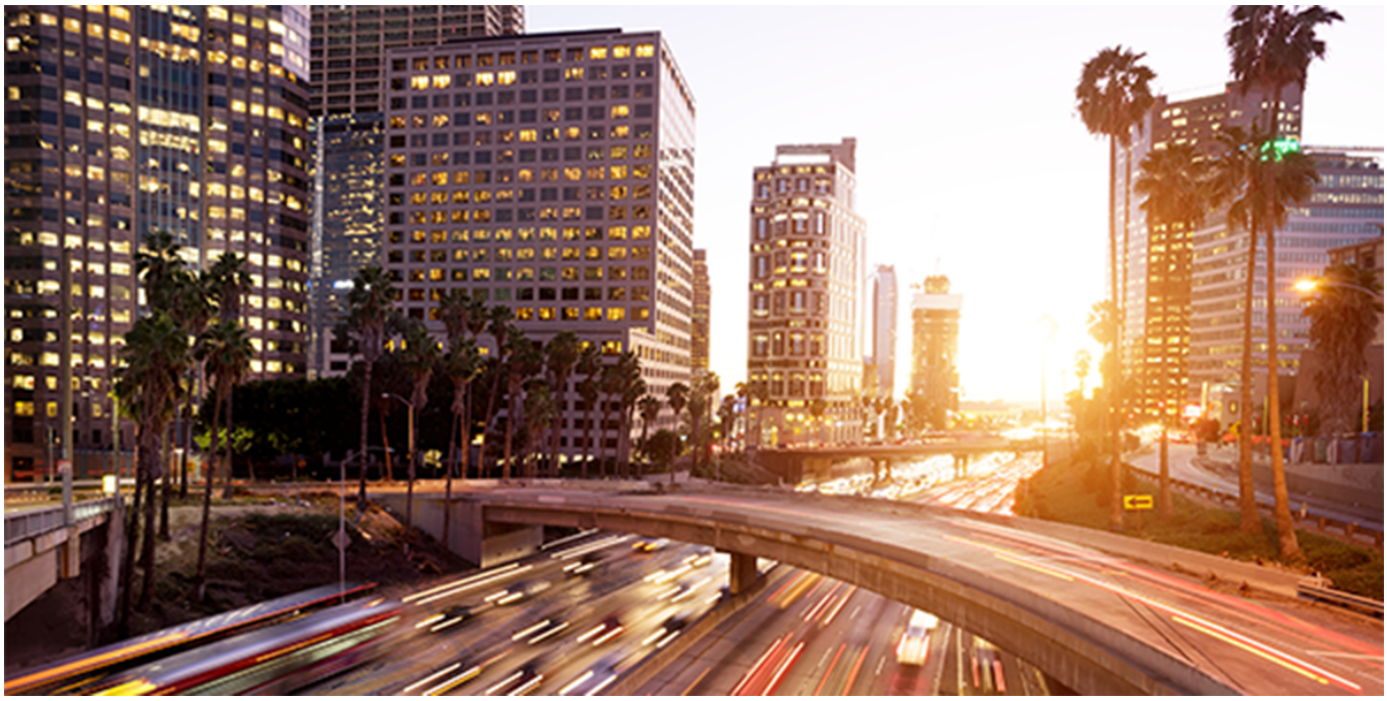
¿Debería ser un microservicio? Mantenga estos seis factores en mente

19 DE ENERO DE 2018 **NATHANIEL SCHUTTA (HTTPS://CONTENT.PIVOTAL.IO/AUTHORS/NATHANIEL-SCHUTTA)**



Estás escribiendo más código que nunca. El truco es saber qué debería ser un microservicio y qué no.

En estos días, no se puede mover un marcador de borrado en seco sin golpear a alguien hablando de microservicios. Los desarrolladores están estudiando el libro profético de Eric Evan, **Domain Driven Design (https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215)** . Los equipos están refabricando aplicaciones monolíticas, buscando contextos delimitados y definiendo un

lenguaje ubicuo. Y aunque ha habido un sinnúmero de artículos, videos y charlas para ayudarte a convertir a microservicios, pocos han pasado un tiempo apreciable preguntándoles si una aplicación determinada debería ser un microservicio.

Hay muchas buenas razones para usar una arquitectura de microservicios. Pero no hay almuerzos gratis. Los aspectos positivos de los microservicios vienen con mayor complejidad. Los equipos deberían asumir esa complejidad ... siempre que la aplicación en cuestión se beneficie de la ventaja de los microservicios.

Por favor, Microservicio Responsablemente

Matt Stine (https://twitter.com/mstine) y yo recientemente pasamos unos días con un cliente que revisa algunas de sus aplicaciones. La discusión comenzó desde el punto de vista de que "todo debería ser un microservicio", como suele ser en estos días. La conversación se estancó cuando las personas discutieron sobre varios detalles de implementación.

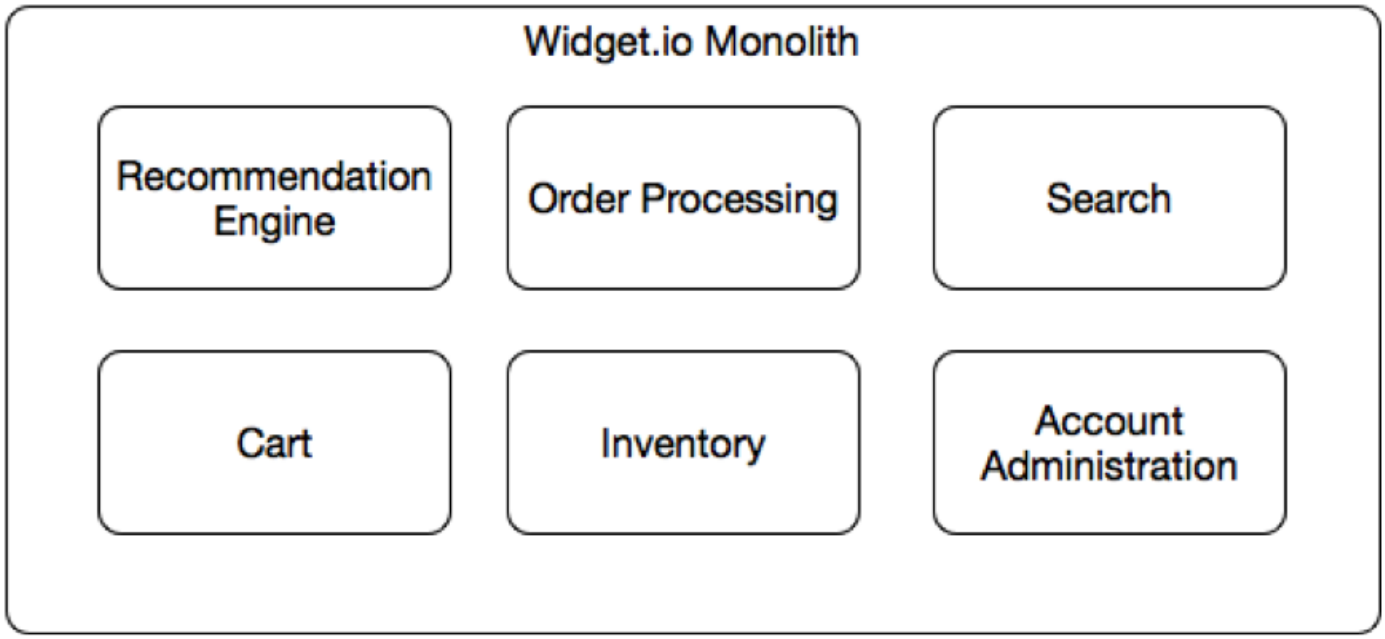
Eso llevó a Matt a escribir un conjunto de principios en la pizarra. Estas simples declaraciones nos guiaron el resto del día. Nos llevaron a cuestionar cada parte de la arquitectura de la aplicación, buscando los lugares donde un microservicio ofrecería valor. La lista cambió fundamentalmente el tono de la conversación y ayudó al equipo a tomar buenas decisiones arquitectónicas.

Para librar al mundo de los microservicios excedentes, presentamos esta lista para ayudarlo a enfocar sus esfuerzos. Lea los siguientes principios y pregunte si la aplicación en cuestión se beneficia de un principio dado. Si responde "sí" a uno o más de los siguientes principios, la característica es un buen candidato para ser un microservicio. Si responde "no" a cada principio, es probable que introduzca **una complejidad accidental (https://www.amazon.com/Mythical-Man-Month-Software-Engineering-Anniversary/dp/0201835959)** en su sistema.

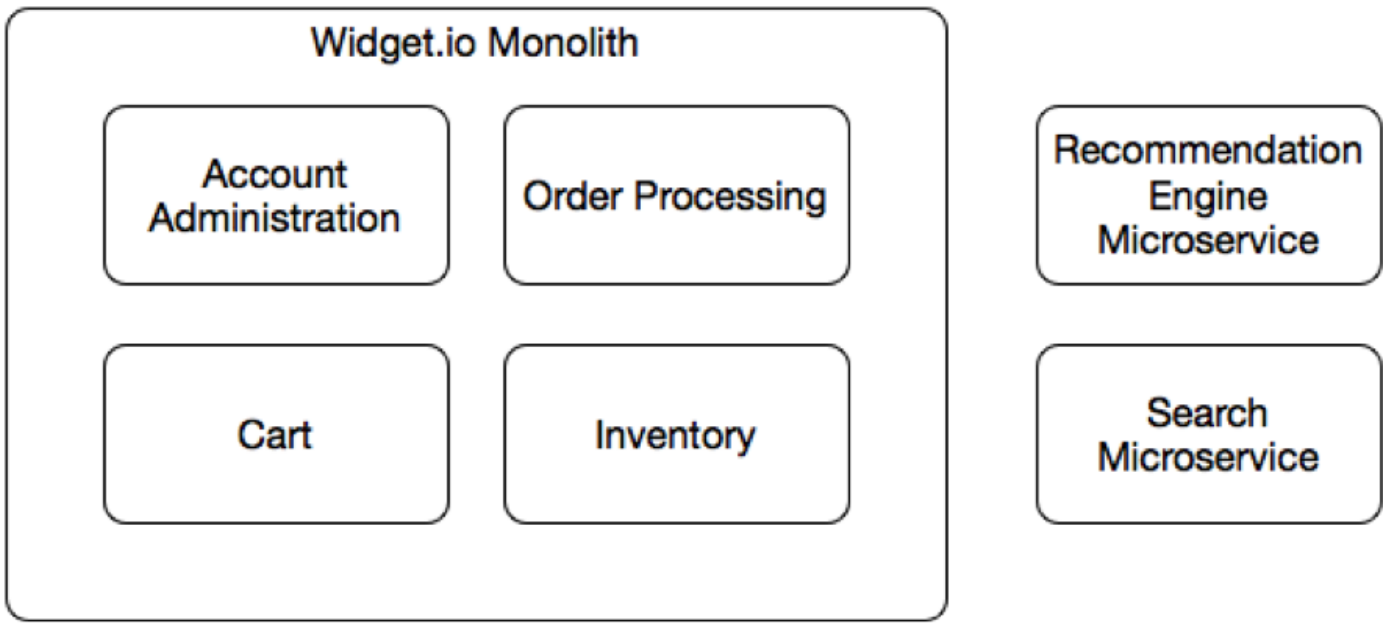
1. Múltiples tasas de cambio

¿Deben evolucionar partes de su sistema a diferentes velocidades o en diferentes direcciones? Luego sepárelos en microservicios. Esto permite que cada componente tenga ciclos de vida independientes.

En cualquier sistema, algunos módulos apenas se tocan, mientras que otros parecen cambiar cada iteración. Para ilustrarlo, tomemos un ejemplo, digamos una aplicación monolítica de comercio electrónico para minoristas en línea.



Nuestras funciones de **Carro** e **Inventario** pueden no haber sido tocadas en el trabajo de desarrollo diario. Pero podríamos estar experimentando constantemente con nuestro **Motor de recomendación** . También queremos mejorar diligentemente nuestra capacidad de **búsqueda** . Dividir esos dos módulos en microservicios permitiría a esos equipos respectivos iterar a un ritmo más rápido, permitiéndonos entregar rápidamente valor comercial.



2. Ciclos de vida independientes

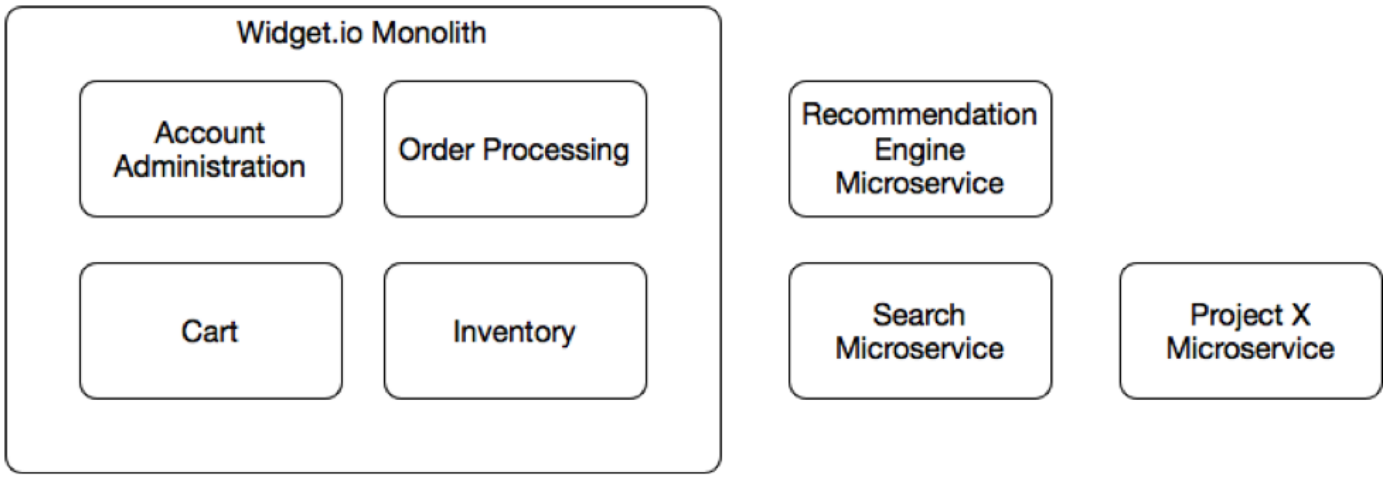
Si un módulo necesita tener un ciclo de vida completamente independiente (es decir, el código se compromete con el flujo de producción), entonces debe ser un microservicio. Debe tener su propio repositorio de código, canalización de CI / CD, etc.

Un alcance más pequeño hace que sea mucho más fácil probar un microservicio. ¡Recuerdo un proyecto con un conjunto de pruebas de regresión de 80 horas! Huelga decir que no ejecutamos una prueba de regresión completa muy a menudo (aunque realmente queríamos hacerlo). Un enfoque de microservicio admite pruebas de regresión de grano fino. Esto nos habría ahorrado innumerables horas. Y hubiéramos entendido los problemas antes.

Las pruebas no son la única razón por la que podemos dividir un microservicio. En algunos casos, una necesidad comercial puede llevarnos a una arquitectura de microservicio. Examinemos nuestro ejemplo de Widget.io Monolith.

Nuestro liderazgo empresarial podría haber identificado una nueva oportunidad, y la velocidad de comercialización es primordial. Si decidimos agregar las nuevas características deseadas al monolito, tomaría demasiado tiempo. No pudimos avanzar al ritmo que el negocio requiere.

Pero como un microservicio independiente, el **Proyecto X** (que se muestra a continuación) puede tener su propia cartera de implementación. Este enfoque nos permite iterar rápidamente y aprovechar la nueva oportunidad de negocio.



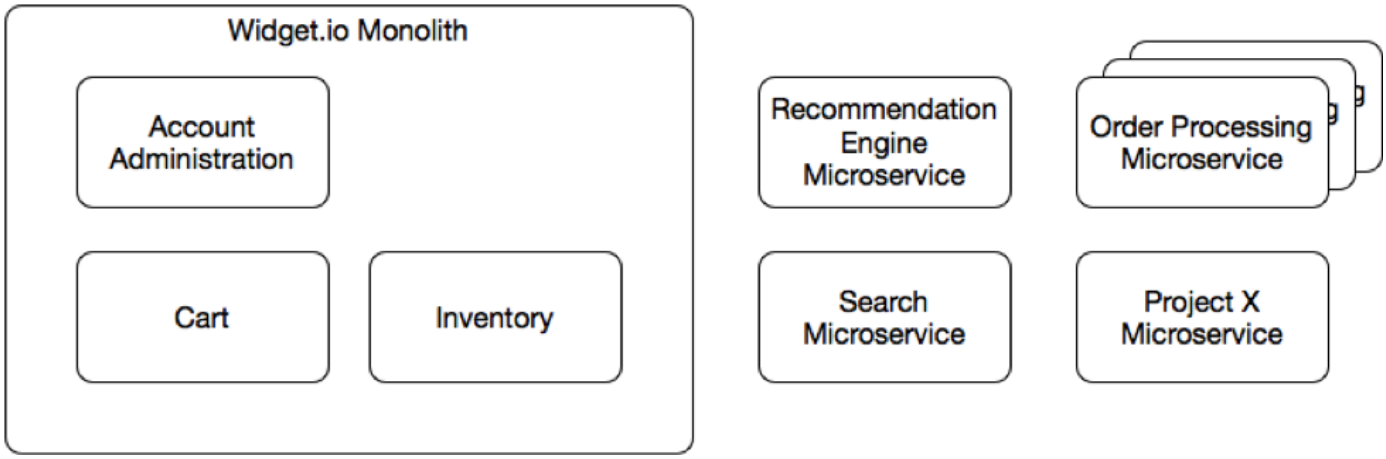
3. Escalabilidad independiente

Si las características de carga o rendimiento de partes del sistema son diferentes, pueden tener diferentes requisitos de escala. La solución: ¡separe estos componentes en microservicios independientes! De esta forma, los servicios pueden escalar a diferentes velocidades.

Incluso una revisión superficial de una arquitectura típica revelará diferentes requisitos de escala entre los módulos. Repasemos nuestro Widget.io Monolith a través de este lente.

Las probabilidades son que nuestra funcionalidad de **administración de cuentas** no se ve tan estresada como el sistema de **procesamiento de pedidos** . En el pasado, hemos tenido que escalar todo el monolito para admitir nuestro componente más volátil. Este enfoque se traduce en mayores costos de infraestructura, porque estamos obligados a "sobre provisionar" para el peor de los escenarios de solo una parte de nuestra aplicación.

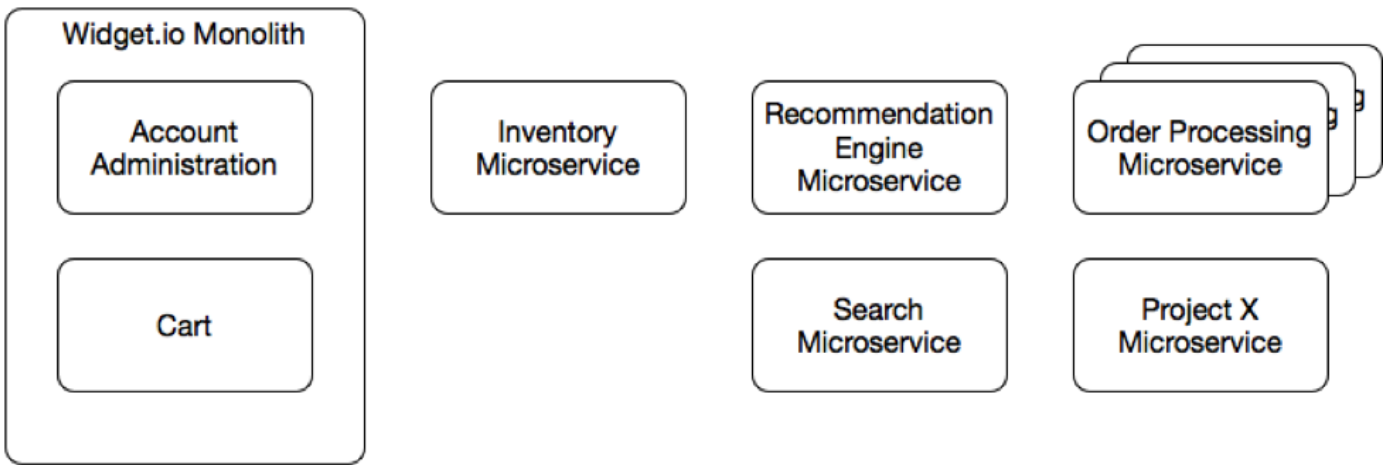
Si refabricamos la funcionalidad de **Procesamiento de pedidos** a un microservicio, podemos escalar hacia arriba y hacia abajo según sea necesario. El resultado es algo así como este diagrama:



4. Fracaso aislado

A veces queremos aislar nuestra aplicación de un tipo particular de falla. Por ejemplo, ¿qué sucede cuando tenemos una dependencia en un servicio externo que no cumple con nuestros objetivos de disponibilidad? Podríamos crear un microservicio para aislar esa dependencia del resto del sistema. A partir de ahí, podemos construir mecanismos de conmutación por error adecuados en ese servicio.

Volviendo una vez más a nuestra muestra Widget.io Monolith, la funcionalidad de **Inventario** interactúa con un sistema de almacenamiento heredado, uno con un tiempo de actividad inferior al estelar. Podemos proteger nuestro objetivo de nivel de servicio para la disponibilidad mediante la refacturación del módulo Inventario en un microservicio. Es posible que necesitemos agregar algo de redundancia para tener en cuenta la falta de fluidez de los sistemas de almacenamiento. También podemos presentar algunos mecanismos de coherencia eventuales, como un inventario de almacenamiento en caché en Redis. Pero por ahora, el cambio a microservicios mitiga el bajo rendimiento de una dependencia de terceros poco confiable.



5. Simplifique las interacciones con dependencias externas (también conocido como el patrón Fachada)

Este principio es similar a "Fracaso aislado". El giro: nos enfocamos más en proteger nuestros sistemas de las dependencias externas que cambian con frecuencia. (Esto también podría ser una dependencia del proveedor, donde un proveedor de servicios se intercambia por otro, como un cambio en quién procesa el pago).

Los microservicios pueden actuar como **una capa de indirección (https://www2.dmst.aueb.gr/dds/pubs/inbook/beautiful_code/html/Spi07g.html)** , para aislarte de una dependencia de terceros. En lugar de llamar directamente a la dependencia, podemos colocar una capa de abstracción (que controlamos) entre la aplicación principal y la dependencia. Además, podemos construir esta capa para que sea fácil de consumir para nuestra aplicación, ocultando la complejidad de la dependencia. Si las cosas cambian en el futuro y tienes que migrar, tus cambios solo se verán limitados a la fachada, en lugar de una refactorización más grande.

6. La libertad de elegir la tecnología adecuada para el trabajo

Con microservicios, los equipos pueden usar sus pilas de tecnología preferidas. En algunos casos, un requisito comercial se ajusta a una opción de tecnología específica. Otras veces, es impulsado por la preferencia del desarrollador y la familiaridad.

NOTA: ¡este principio no es una licencia para utilizar todas las tecnologías bajo el sol! Proporcione orientación a sus equipos sobre la elección de tecnología. Demasiadas pilas dispares agregan una carga cognitiva, y pueden ser peores que la estandarización en un modelo de "talla única". Mantener las bibliotecas de terceros al día para una pila es todo un desafío. Multiplique ese trabajo por cuatro o cinco, y se ha registrado para una gran carga organizacional. Haga lo que funciona, y concéntrese en los "caminos pavimentados" que sabe cómo apoyar.

En nuestro ejemplo de Widget.io, la funcionalidad de **búsqueda** podría beneficiarse de una elección de idioma o base de datos diferente que el resto de nuestros módulos. Es sencillo hacer esto si lo desea. ¡Y por supuesto, ya lo hemos refabricado por otros motivos!

Verificación cultural

Esa es la discusión de la tecnología. ¿Y ahora qué hay de la cultura?

Ninguna decisión técnica existe en el vacío. Entonces, antes de sumergirse en el maravilloso mundo de los microservicios, eche un vistazo a su organización. ¿Tu estructura de organización soporta fácilmente una arquitectura de microservicios? ¿Qué dice **la Ley de Conway (http://www.melconway.com/Home/Conways_Law.html)** sobre sus posibilidades de éxito?

Pivotal
@pivotal

"Effects of Conway's Law can be counteracted by deliberate application of kindergarten lessons." - @jmckenty. More: bit.ly/2t74vwr

11:50 PM - Jun 16, 2017

2 16 29

Hace cincuenta años, Mel Conway teorizó que un sistema diseñado por cualquier organización creará un sistema que refleje su organigrama. En otras palabras, si sus equipos no están organizados como grupos pequeños y autónomos, es poco probable que sus ingenieros creen software compuesto por servicios pequeños y autónomos. Esta realización ha provocado la **maniobra Inverse Conway. (https://www.thoughtworks.com/radar/techniques/inverse-conway-maneuver)**Esto anima a los equipos a cambiar su organigrama para reflejar la arquitectura que desean ver en sus aplicaciones.

También debe considerar su preparación cultural. Los microservicios fomentan cambios pequeños y frecuentes, una cadencia que a menudo entra en conflicto con un ciclo de publicación trimestral tradicional. Con microservicios, no tendrá congelaciones de código, o una integración de código "big bang". Si bien una arquitectura basada en microservicios ciertamente puede funcionar en un entorno de cascada tradicional, no verá todos los beneficios.

Dormain Drewitz
@DormainDrewitz

Whenever you try to lead through a transformation, those complacency antibodies will kick in/@Boeing's Niki Allen on why changing a 100+ yo company is hard

6:22 PM - Dec 6, 2017

3 5

Considere también cómo aprovisiona la infraestructura. Los equipos que se centran en el autoservicio y optimizan el flujo de valor a menudo adoptan un paradigma de microservicios. Las plataformas como Pivotal Cloud Foundry ayudan a sus equipos a implementar rápidamente los servicios, probarlos y perfeccionarlos en cuestión de minutos en lugar de semanas (o meses). Los desarrolladores pueden activar una instancia con solo presionar un botón, una práctica que fomenta la experimentación y el aprendizaje. Buildpacks automatiza la administración de dependencias. Eso significa que los desarrolladores y operadores pueden enfocarse en entregar valor comercial.

Nate Schutta
@ntschutta

Yes, even your company in your industry can move away from four deploys a year to, well thousands a month. #springone

7:14 PM - Dec 5, 2017

6 4

Finalmente, formulemos dos preguntas específicas de la aplicación:

- ¿Esta aplicación tiene múltiples dueños de negocios? Si un sistema tiene múltiples propietarios independientes y autónomos, entonces tiene dos fuentes distintas de cambio. El conflicto puede surgir de esta situación. Con microservicios, puede lograr "ciclos de vida independientes" y complacer a estos diferentes grupos.

- ¿Esta aplicación es propiedad de múltiples equipos? El "costo de coordinación" para múltiples equipos que trabajan en un solo sistema puede ser alto. En su lugar, defina las API para ellos. A partir de ahí, cada equipo puede construir un microservicio independiente usando **Spring Cloud Contract (https://cloud.spring.io/spring-cloud-contract/)** o **Pact (https://docs.pact.io)** para pruebas de **contratos dirigidos (https://martinfowler.com/articles/consumerDrivenContracts.html)** por el **consumidor (https://martinfowler.com/articles/consumerDrivenContracts.html)** .

Responder afirmativamente a cualquiera de estas preguntas debería conducirlo hacia una solución de microservicio.

Terminando

El camino hacia los microservicios está pavimentado con buenas intenciones. Pero más que unos pocos equipos se están subiendo al carro sin analizar primero sus necesidades. Los microservicios son poderosos. ¡Deben estar absolutamente en tu caja de herramientas! Solo asegúrate de considerar las compensaciones. No hay sustituto para comprender los controladores comerciales de sus aplicaciones; esto es esencial para determinar el enfoque arquitectónico adecuado.

¿Listo para experimentar con microservicios en Pivotal Cloud Foundry hoy? Consulte **Small Footprint (https://docs.pivotal.io/pivotalcf/2-0/customizing/small-footprint.html)** , **PCF Dev (https://pivotal.io/pcf-dev)** o desarrolle una versión de prueba gratuita en **Pivotal Web Services (http://run.pivotal.io)** . Mientras

lo hace, consulte **nuestra página de temas de microservicios (https://pivotal.io/microservices)** y este documento técnico **Running Microservices en Pivotal Cloud Foundry (https://content.pivotal.io/white-papers/running-microservices-on-pivotal-cloud-foundry)** .

¿Desea obtener más información sobre cada uno de estos factores de diseño? ¡Manténganse al tanto! Exploraremos cada uno de ellos con más profundidad, como parte de una serie de blogs en curso.

Sobre el Autor

Nathaniel T. Schutta es un arquitecto de software centrado en la computación en la nube y la construcción de aplicaciones utilizables. Un defensor de la programación políglota, Nate ha escrito varios libros y apareció en varios videos. Nate es un orador experimentado que presenta regularmente conferencias en todo el mundo, simposios No Fluff Just Stuff, reuniones, universidades y grupos de usuarios. Impulsado por librar al mundo de las malas presentaciones, Nate fue coautor del libro Presentation Patterns con Neal Ford y Matthew McCullough.

Seguir en Twitter (https://twitter.com/@ntschutta)

Más contenido por Nathaniel Schutta (https://content.pivotal.io/authors/nathaniel-schutta)

Previous (https://content.pivotal.io/blog/curing-handoff-itis)

Curing Handoff-itis

How the waterfall methodology is exhausting y...

Next (https://content.pivotal.io/blog/diversity-and-inclusion-in-2017-how-we-showed-up)

Diversidad e inclusión en 2017: ...

Pivotal lanza métricas de diversidad e inclusió...

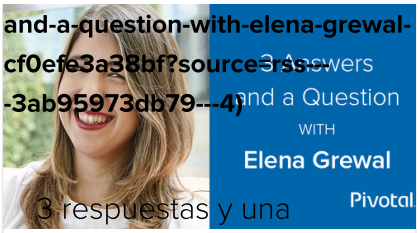
Contenido relacionado en este Stream

(https://builttoadapt.io/3-answers-

(https://builttoadapt.io/a-letter-to-

(https://builttoadapt.io/curing-

(https://conte



3 respuestas y una pregunta con Elena Grewal

Una carta a un ingeniero junior

Curing Handoff-itis

Diversidad 2017: cóm

LEARN (HTTPS://PIVOTAL.IO/LEARN)

Topics (https://pivotal.io/topics)

Products & Services (https://pivotal.io/products)

Industries (https://pivotal.io/industries)

Resources (/resources)

Training (https://pivotal.io/training)

Webinars (/webinars)

Customers (https://pivotal.io/customers)

BUILD (HTTPS://PIVOTAL.IO/BUILD)

Tutorials (https://pivotal.io/platform/pcf-tutorials/getting-started-with-pivotal-cloud-foundry?utm_source=uberflipfooterfree&utm_medium=link&utm_campaign=gettingstartedwithpcf)

PWS (http://run.pivotal.io)

PCF Dev (https://pivotal.io/pcf-dev)

Documentation (http://docs.pivotal.io/)

Downloads (https://network.pivotal.io/)

Knowledge Base (https://discuss.pivotal.io)

Support (https://support.pivotal.io)

Pivotal Services Marketplace (https://pivotal.io/platform/services-marketplace)

CONNECT (HTTPS://PIVOTAL.IO/CONNECT)

Advocates (https://pivotal.io/advocates)

#ImAPivot Community (https://pivotal.io/imapivot)

Events (https://pivotal.io/events)

Product Consultation (https://pivotal.io/office-hours)

Blog (/)

Podcasts (/podcasts)

Social (/social)

Newsletter (https://pivotal.io/newsletter-subscription)

COMPANY (HTTPS://PIVOTAL.IO/ABOUT)

About (https://pivotal.io/about)

Careers (https://pivotal.io/careers)

Contact (https://pivotal.io/contact)

Executive Team (https://pivotal.io/team)

Investors (https://pivotal.io/investors)

Legal (https://pivotal.io/legal)

Locations (https://pivotal.io/locations)

Partners (https://pivotal.io/partners)

Press Center (https://pivotal.io/press-center)

(https://twitter.com/pivotal)

(https://www.linkedin.com/com

(https://medium.com/built-to-adapt)

(https://www.youtube.com/c/pi

(https://plus.google.com/10532

(https://www.facebook.com/piv

(https://pivotal.io/cn) (https://pivotal.io/de/) (https://pivotal.io/fr/) (https://pivotal.io/jp/) (https://pivotal.io/kr/)