



APIS Y MICROSERVICIOS

© 2017, ACTIBYTI PROJECT SLU, Barcelona
Autor: Ricardo Ahumada



MINISTERIO
DE ENERGÍA, TURISMO
Y AGENDA DIGITAL

red.es



ESTRATEGIA DE
EMPRENDIMIENTO Y
EMPLEO JUVENIL
garantía juvenil



UNIÓN EUROPEA

Fondo Social Europeo
“El FSE invierte en tu futuro”

ÍNDICE DE CONTENIDOS

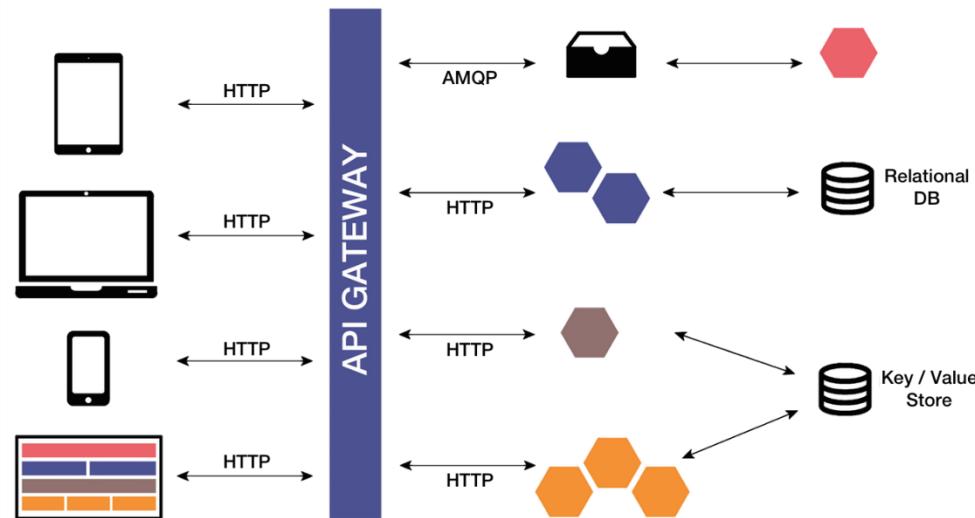
1. Introducción a los Microservicios
2. Arquitectura
3. Mensajería y contextos de dominio
4. Desarrollo de microservicios con Node.js

1

Introducción a los Microservicios

Microservicios

- Una arquitectura de microservicios es un patrón de diseño de software en el que una aplicación más grande se compone de múltiples servicios más pequeños, cada uno con su propio objetivo, que colaboran y se comunican a través de una red para lograr un objetivo particular.
- Dentro de una arquitectura de microservicios, cada servicio se ejecuta en su propio proceso y se comunica con otros procesos utilizando un mecanismo ligero como HTTP / REST con JSON.

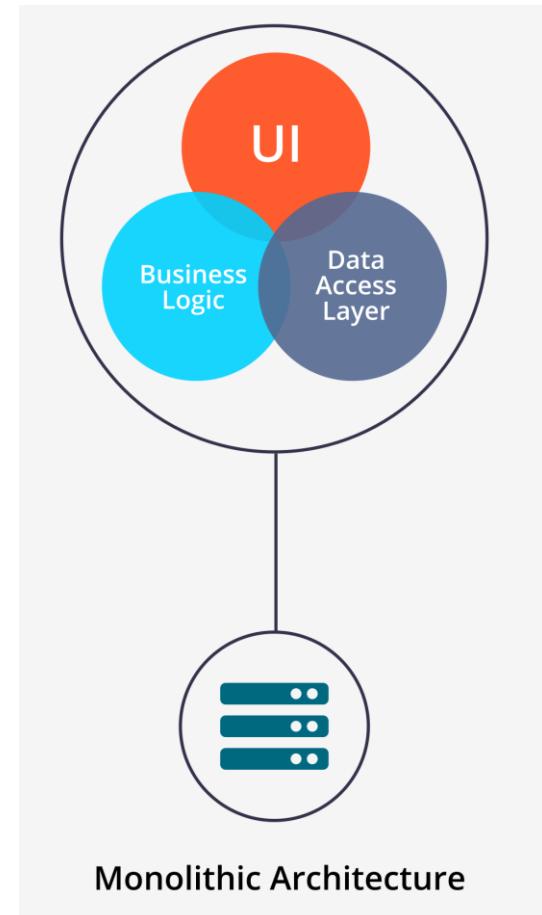


Aplicaciones monolíticas

- Para entender el estilo de microservicio es útil compararlo con el estilo monolítico.
- Una **aplicación monolítica** construida como una sola unidad. Las aplicaciones empresariales se construyen a menudo en tres partes principales:
 - Una **interfaz de usuario** del lado del cliente (que consta de páginas HTML y javascript que se ejecutan en un navegador en la máquina del usuario)
 - Una **base de datos** (que consta de muchas tablas insertadas en una base de datos común, Sistema),
 - Una **aplicación del lado del servidor**. (Gestionará las solicitudes HTTP, ejecutará la lógica del dominio, recuperará y actualizará los datos de la base de datos y seleccionará y completará las vistas HTML para enviarlas al navegador).

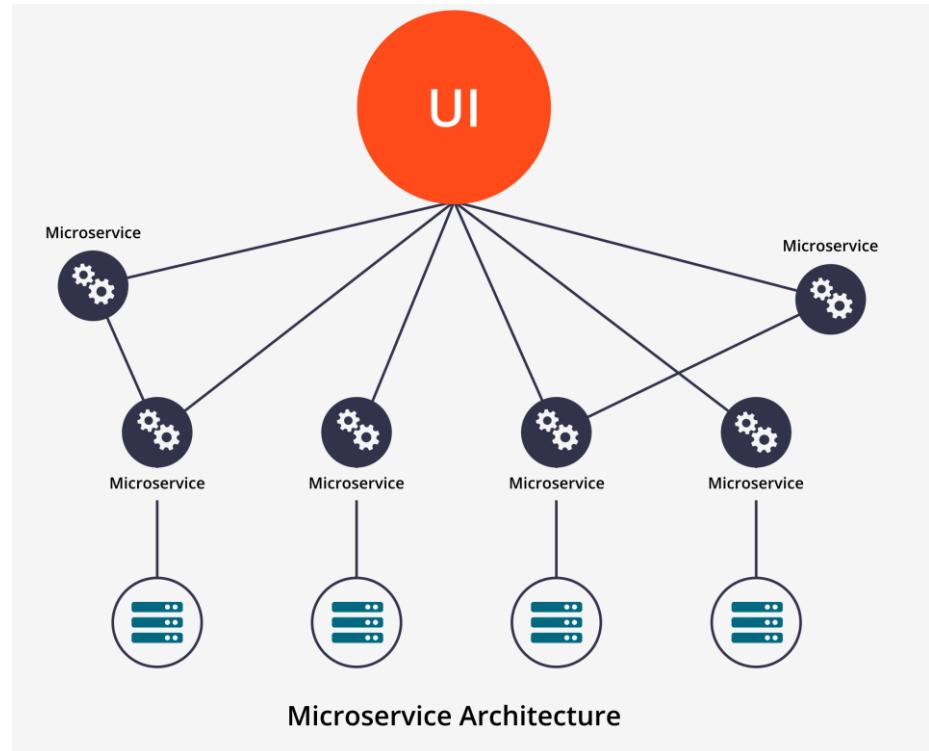
Aplicaciones monolíticas

- Un servidor monolítico es una forma natural de abordar la construcción de dicho sistema.
- Toda su lógica para manejar un request se ejecuta en un solo proceso, lo que le permite utilizar las funciones básicas para dividir la aplicación en clases, funciones y namespaces.
- Los ciclos de cambio están acoplados (un cambio hecho en una pequeña parte de la aplicación, requiere que el monolito entero sea reconstruido y desplegado).
- Con el tiempo es difícil mantener una buena estructura modular, lo que hace más difícil mantener los cambios que sólo deberían afectar a un módulo dentro de ese módulo.



Microservicios

- El concepto de dividir una aplicación en partes discretas no es nuevo.
- La idea para microservicios se origina en el patrón de diseño de arquitectura orientado a servicios más amplio, en el que los servicios independientes realizan funciones distintas y se comunican utilizando un protocolo designado.
- Sin embargo, a diferencia de la arquitectura orientada a servicios, una arquitectura de microservicios (como su nombre indica) debe contener servicios que son explícitamente pequeños y ligeros y que son desplegables de forma independiente.

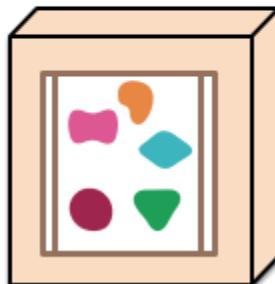
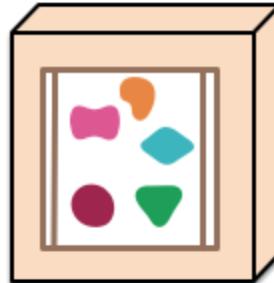


Esquema comparativo Monolítico-Microservicios

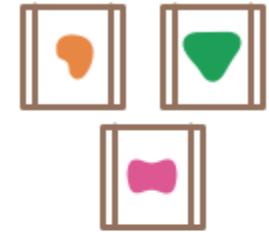
A monolithic application puts all its functionality into a single process...



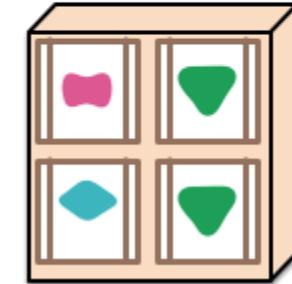
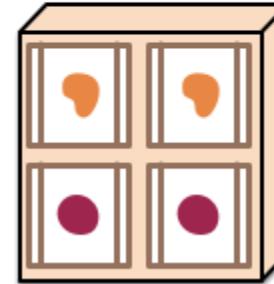
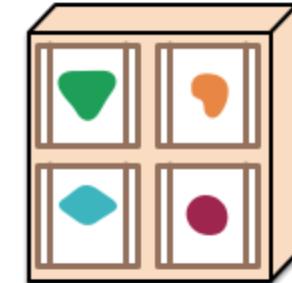
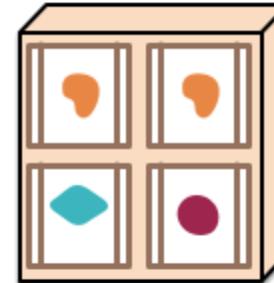
... and scales by replicating the monolith on multiple servers



A microservices architecture puts each element of functionality into a separate service...



... and scales by distributing these services across servers, replicating as needed.



Ventajas de los Microservicios

- Puedes utilizar diferentes tecnologías por cada servicio (Java EE, Node, Python, PHP, etc.).
- También permite que cada servicio tenga un ciclo de vida independiente, es decir, versión independiente del resto, inclusive equipos de desarrollo diferentes.
- Al ser servicios sin dependencia entre sí (especialmente de sesión), se puede ejecutar el mismo en varios puertos, colocando un balanceador delante.
- Se puede crear instancias en servidores de diferentes regiones, lo que permitirá crecer (tanto verticalmente como horizontal) sin necesidad de cambiar el código fuente.

2

Arquitectura

Contexto

- Se está desarrollando una aplicación de empresa del lado del servidor. Debe soportar una variedad de clientes diferentes, incluyendo navegadores de escritorio, navegadores móviles y aplicaciones móviles nativas.
- La aplicación también puede exponer una API para que terceras partes la consuman. También podría integrarse con otras aplicaciones a través de servicios web o un intermediario de mensajes.
- La aplicación gestiona las solicitudes (requests y mensajes HTTP) mediante la ejecución de la lógica empresarial; acceder a una base de datos; intercambiar mensajes con otros sistemas; y devolver una respuesta HTML / JSON / XML.
- Hay componentes lógicos que corresponden a diferentes áreas funcionales de la aplicación.

Problema

- ¿Cuál es la arquitectura de implementación de la aplicación?

Fuerzas

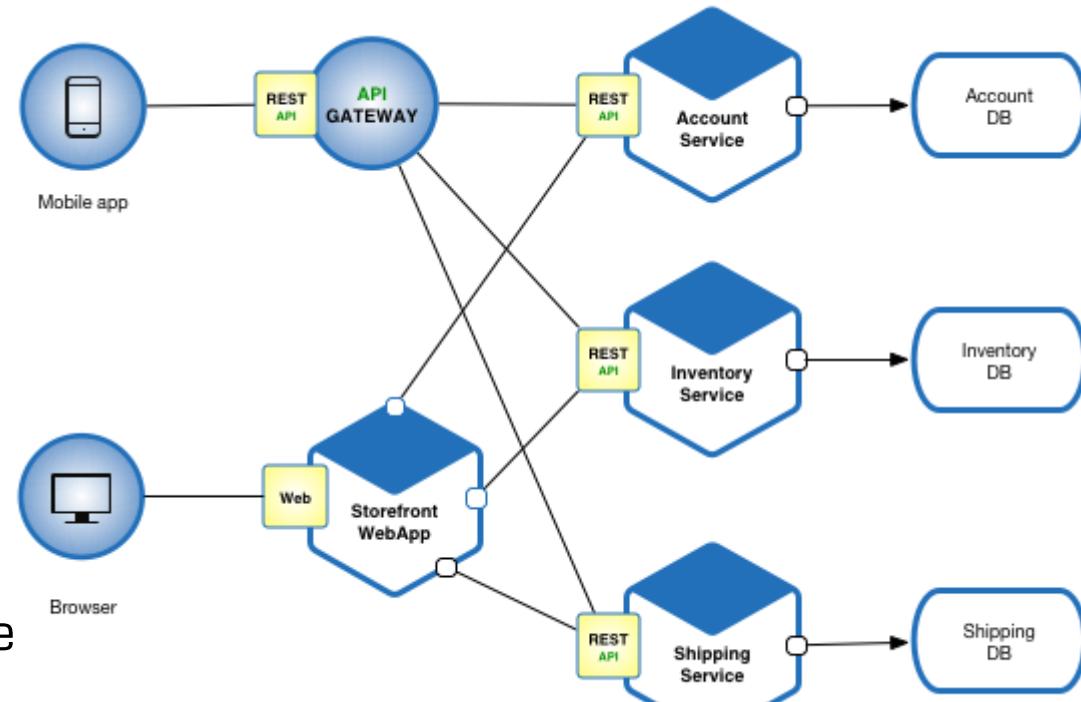
- Hay un equipo de desarrolladores trabajando en la aplicación
- Los nuevos miembros del equipo deben ser rápidamente productivos
- La aplicación debe ser fácil de entender y modificar
- Se quiere hacer despliegue continuo de la aplicación
- Se debe ejecutar varias copias de la aplicación en varias máquinas para satisfacer los requisitos de escalabilidad y disponibilidad
- Se desea aprovechar las tecnologías emergentes (frameworks, lenguajes de programación, etc.)

Solución

- Definir una arquitectura que estructura la aplicación como un conjunto de servicios de colaboración poco acoplados.
- Este enfoque corresponde al eje Y del cubo de escala (<http://microservices.io/articles/scalercube.html>).
 - Cada servicio implementa un conjunto de funciones estrechamente relacionadas.
 - Por ejemplo, una aplicación puede consistir en servicios como el servicio de gestión de pedidos, el servicio de gestión de clientes, etc.
- Los servicios se comunican mediante protocolos síncronos como HTTP / REST o protocolos asincrónicos como AMQP.
- Los servicios se pueden desarrollar y desplegar independientemente uno de otro.
- Cada servicio tiene su propia base de datos para poder ser desacoplado de otros servicios. La coherencia de los datos entre los servicios se mantiene utilizando el patrón Saga (<http://microservices.io/patterns/data/saga.html>)

Ejemplo: Aplicación E-commerce

- Imaginemos que está construyendo una aplicación de comercio electrónico que recibe pedidos de clientes, verifica el inventario y el crédito disponible y los envía.
- La aplicación consta de varios componentes, incluyendo el StoreFrontUI, que implementa la interfaz de usuario, junto con algunos servicios de back-end para verificar el crédito, mantener el inventario y los pedidos de envío. La aplicación consiste en un conjunto de servicios.





Examinando ejemplos

- Accede al sitio <http://eventuate.io/exampleapps.html>
- Examina alguno de los ejemplos de arquitectura propuesta

Características de la arquitectura de Microservicios

Componentización a través de Servicios

- Un componente es una unidad de software que es reemplazable y actualizable de manera independientemente.
- Las arquitecturas de Microservicio usarán librerías, pero su manera primaria de componentización es dividirlo en servicios.
- Definimos las **librerías** como componentes que están vinculados a un programa y se llaman mediante llamadas de función en memoria, mientras que los **servicios** son componentes fuera de proceso que se comunican con un mecanismo como una solicitud de servicio web o una llamada a procedimiento remoto.
- La razón principal para usar servicios como componentes (en lugar de bibliotecas) es que los servicios son desplegables de forma independiente.
- Otra consecuencia es una interfaz de componentes más explícita.

Organización de equipos alrededor de Capacidades Empresariales

- Cuando se busca dividir una aplicación grande en partes, a menudo la administración de equipos se centra en la capa tecnológica, lo que lleva a tener equipos de interfaz de usuario, equipos lógicos del servidor y equipos de bases de datos.
- Cuando los equipos están separados de esta manera, incluso los cambios simples pueden conducir a proyectos cruzados entre equipos que suponen tiempo y coste.
- El enfoque de microservicio es diferente, dividiendo los equipos por servicios organizados alrededor de la capacidad empresarial.
- Cada servicio requiere una implementación completa de software, incluyendo interfaz de usuario, almacenamiento persistente y colaboración externa.
- En consecuencia, los equipos son interdisciplinares, incluyendo toda la gama de habilidades necesarias para el desarrollo: experiencia de usuario, base de datos y gestión de proyectos.

Productos y Gobernanza

➤ Productos no Proyectos

- › La mayoría de los esfuerzos de desarrollo de aplicaciones que vemos utilizan un modelo de proyecto: donde el objetivo es entregar algún software que se considera completado.
- › Al terminar el software se entrega a una organización de mantenimiento y el equipo de proyecto que lo construyó se disuelve.
- › Los proponentes de microservicios tienden a evitar este modelo, prefiriendo en cambio la noción de que un equipo debe poseer un producto durante toda su vida útil.

➤ Gobernanza descentralizada

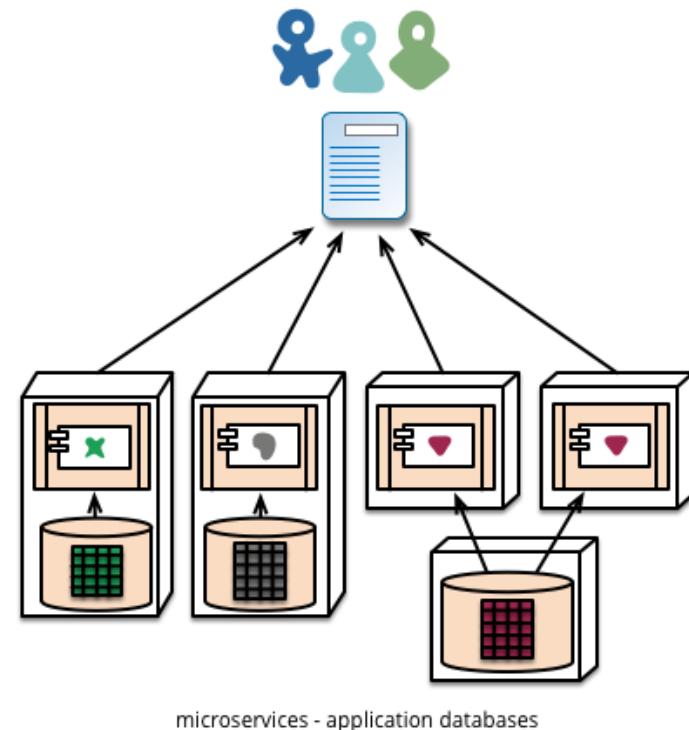
- › Una de las consecuencias de la gobernanza centralizada es la tendencia a estandarizar las plataformas con tecnología única. La experiencia demuestra que este enfoque es limitante.
- › Los microservicios permiten usar la herramienta adecuada para cada caso.

Endpoints inteligentes y tubos mudos

- Al construir estructuras de comunicación entre diferentes procesos, hemos visto muchos productos y enfoques que ponen énfasis en el mecanismo de comunicación.
- Un buen ejemplo de esto es el Enterprise Service Bus (ESB), donde los productos de ESB a menudo incluyen sofisticadas instalaciones para enrutamiento de mensajes, coordinación, transformación y aplicación de reglas de negocio.
- La comunidad entre microservicio favorece un enfoque alternativo: **puntos finales inteligentes y tubos mudos**.
- Las aplicaciones construidas a partir de microservicios tienen como objetivo estar tan desacopladas y cohesivas como sea posible (poseen su propia lógica de dominio y actúan más como filtros) recibiendo una petición, aplicando la lógica según corresponda y produciendo una respuesta.
- Estos son coordinados utilizando simples protocolos REST en lugar de complejos protocolos como WS o BPEL o orquestación de una herramienta central.

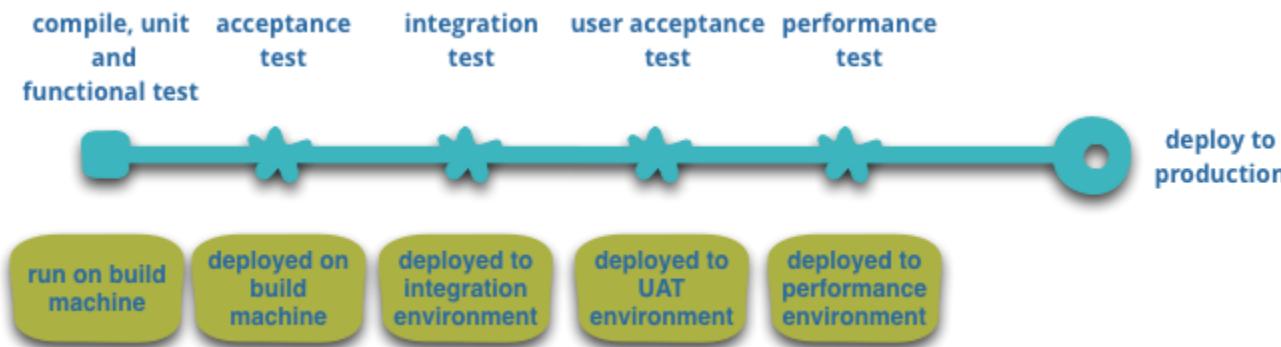
Gestión Descentralizada de Datos

- La descentralización de la gestión de datos se presenta de diferentes maneras.
- En el nivel más abstracto, significa que el modelo conceptual diferirá entre sistemas. Este es un problema común cuando se necesita hacer integración en una gran empresa.
- Una forma útil de pensar sobre esto es la noción de Contexto Limitado (<https://martinfowler.com/bliki/BoundedContext.html>) del Diseño Guiado por Dominio (DDD). DDD divide un dominio complejo en múltiples contextos acotados y mapea las relaciones entre ellos.
- Este proceso es útil tanto para arquitecturas monolíticas como para microservicios.
- Los microservicios prefieren dejar que cada servicio administre su propia base de datos, ya sea diferentes instancias de la misma tecnología de base de datos, o sistemas de base de datos completamente diferentes - un enfoque llamado Polyglot Persistence (<https://martinfowler.com/bliki/PolyglotPersistence.html>).



Automatización de Infraestructura

- Las técnicas de automatización de la infraestructura han evolucionado enormemente en los últimos años. Se quiere hacer uso de conceptos como Continuous Integration (<https://martinfowler.com/bliki/ContinuousDelivery.html>).
- El proceso de construcción que se ilustra a continuación.



- Un par de características clave:
 - Se necesitan ejecutar un montón de pruebas automatizadas.
 - La promoción del software en el proceso significa que automatizamos el despliegue en cada nuevo entorno.

Diseño para el fallo

- Una consecuencia del uso de servicios como componentes, es que las aplicaciones deben diseñarse de manera que puedan tolerar el fallo de los servicios. Cualquier llamada de servicio podría fallar debido a la falta de disponibilidad del proveedor.
- Esto es una desventaja en comparación con un diseño monolítico ya que introduce complejidad adicional para manejarlo.
- Dado que los servicios pueden fallar en cualquier momento, es importante poder detectar los fallos rápidamente y, si es posible, restaurar automáticamente el servicio.
- Las aplicaciones de microservicio ponen mucho énfasis en el monitoreo en tiempo real de la aplicación, comprobando los elementos arquitectónicos (cuántas solicitudes por segundo tiene la base de datos) y métricas relevantes para el negocio (por ejemplo, cuántas órdenes por minuto se reciben).
- El monitoreo semántico puede proporcionar un sistema de alerta temprana de algo que va mal que provoca que los equipos de desarrollo investiguen.

Diseño Evolutivo

- La implementación de componentes en los servicios añade una oportunidad para una planificación de entrega más granular.
- Con un monolito, cualquier cambio requiere una compilación completa y despliegue de toda la aplicación.
- Con los microservicios, sin embargo, sólo es necesario volver a implementar el servicio que modificó.
- Esto puede simplificar y acelerar el proceso de entrega.
- La desventaja es que nos tendremos que preocupar de cambios a un servicio que rompe a sus consumidores.
- El enfoque de integración tradicional es tratar de abordar este problema utilizando el control de versiones, pero la preferencia en el mundo de los microservicios es utilizar sólo el versionado como último recurso: Deberemos diseñar los servicios para que sean lo más tolerantes posible a los cambios en sus proveedores.

Cuándo usar la arquitectura de microservicio

- Un desafío con el uso de este enfoque es decidir cuándo tiene sentido usarlo.
- Al desarrollar la primera versión de una aplicación, a menudo no se tienen los problemas que este enfoque resuelve. Además, el uso de una arquitectura elaborada y distribuida ralentizará el desarrollo.
- Esto puede ser un problema importante para las startups cuyo mayor desafío es a menudo cómo evolucionar rápidamente el modelo de negocio y la aplicación que lo acompaña.
- Más adelante, sin embargo, cuando el desafío es cómo escalar y se necesita utilizar descomposición funcional, las interdependencias podrían dificultar la descomposición de tu aplicación monolítica en un conjunto de servicios.
- Unido a esto está la madurez de la empresa y de los equipos que la componene.

Cómo descomponer la aplicación en servicios

- Descomponer por capacidad empresarial y definir servicios que correspondan a las capacidades empresariales.
- Descomponer por subdominio DDD.
- Descomponer por verbo o caso de uso y definir servicios que son responsables de acciones particulares. p.ej. un servicio de envío que es responsable de enviar pedidos completos.
- Descomponer por sustantivos o recursos definiendo un servicio que es responsable de todas las operaciones en entidades / recursos de un tipo determinado. p.ej. un Servicio de Cuentas que es responsable de administrar cuentas de usuario.
- Idealmente, cada servicio debe tener sólo un pequeño conjunto de responsabilidades. (Tío) Bob Martin habla de diseñar clases usando el Principio de Responsabilidad Única (SRP). El SRP define una responsabilidad de una clase como una razón para cambiar, y establece que una clase sólo debe tener una razón para cambiar. Tiene sentido aplicar el SRP al diseño del servicio también.

Cómo mantener la consistencia de los datos

- Para garantizar el acoplamiento débil, cada servicio debe tener su propia base de datos.
- Mantener la coherencia de los datos entre los servicios es un reto. Una aplicación debe usar el patrón Saga:
 - Un servicio publica un evento cuando sus datos cambian.
 - Otros servicios consumen ese evento y actualizan sus datos.
- Existen varias maneras de actualizar de manera fiable los datos y los eventos de publicación, incluyendo el Sourcing de eventos y el registro de “registros de transacciones”.

Cómo implementar consultas

- Otro desafío es implementar consultas que necesitan recuperar datos pertenecientes a múltiples servicios.
- Los siguientes patrones pueden ser útiles:
 - Composición de API: <http://microservices.io/patterns/data/api-composition.html>
 - Segregación de Responsabilidad de Consultas de Comando (CQRS): <http://microservices.io/patterns/data/cqrs.html>

3

Mensajería y contextos de dominio

Mensajería y contextos de dominio

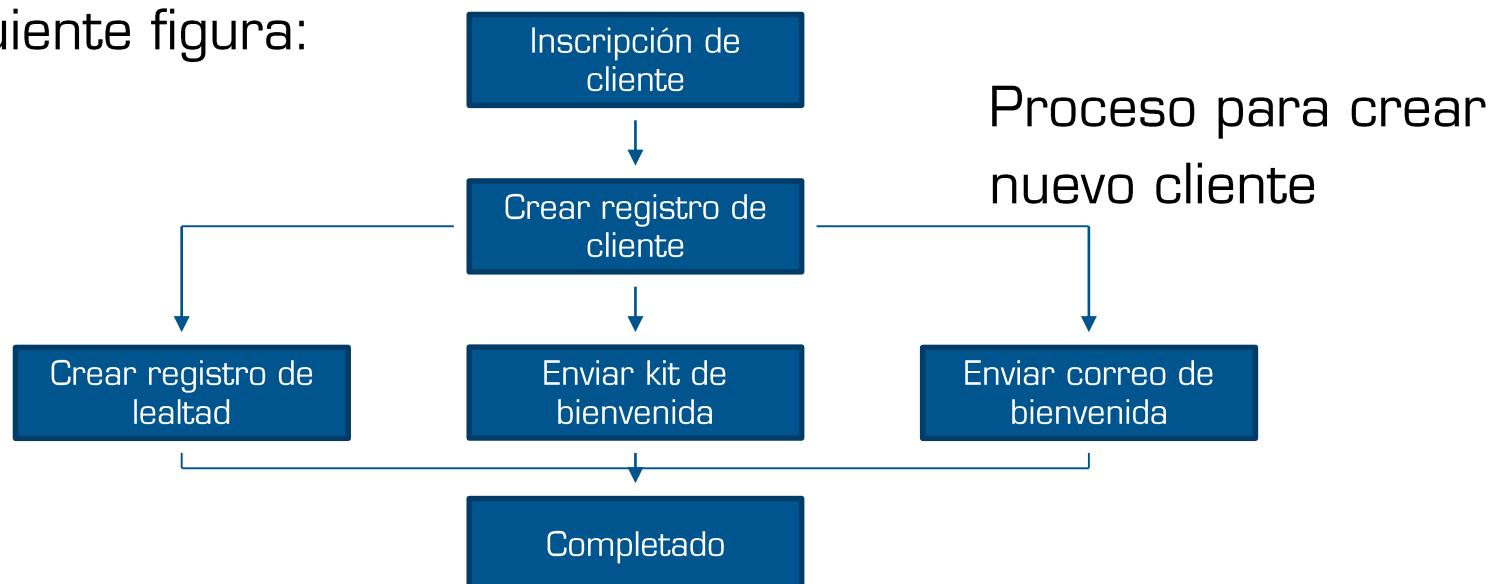
- En términos de integración, cuando se construyen estructuras de comunicación entre los diferentes procesos, es común ver productos y enfoques que ponen el énfasis de la inteligencia en el mecanismo de comunicación en sí.
- normalmente los productos de ESB (Enterprise Service Bus) son un ejemplo donde generalmente se incluyen sofisticadas estructuras para el ruteo de mensajes, la coreografía, la transformación, la aplicación de reglas de negocio, etc.
- En términos de diseño, también es común ver que a la hora de definir dónde colocamos cierto código que representa reglas de negocio, a veces, decidimos hacerlo dentro del ESB por conveniencia o rapidez. Entonces, convertimos al ESB en un elemento de integración clave para el funcionamiento del proceso en su totalidad, y esa excesiva inteligencia puesta en un único componente (la tubería), lo torna al sistema en algo más frágil que antes.

Mensajería y contextos de dominio

- Las aplicaciones creadas con microservicios pretenden ser tan disociadas y cohesivas como sean posible, ellas poseen su propia lógica de dominio y actúan más como filtros en el clásico sentido Unix:
 - recibe una solicitud, aplica la lógica apropiada, y produce una respuesta.
 - Estos pasos son coordinados utilizando protocolos REST simples en lugar de protocolos complejos WS-BPEL o la coordinación por una herramienta central.
- Los dos protocolos que se utilizan con mayor frecuencia son:
 - Petición/respuesta de HTTP con recursos API
 - Mensajería liviana, como puede ser RabbitMQ o ZeroMQ.
- Estos principios y protocolos hacen que los equipos de microservicios, a la hora de integrar servicios mas complejos, prefieran el concepto de **coreografía**, en lugar de **orquestación**.

Ejemplo

- Veámoslo con un ejemplo sobre un servicio del CRM, por ejemplo, luego que damos de alta un nuevo cliente, debemos completar las siguientes operaciones:
 - Acreditarle una cantidad de puntos de bienvenida en su cuenta de fidelidad.
 - Enviarle un kit de bienvenida a través del correo postal.
 - Enviarle un correo electrónico de bienvenida al cliente.
- Si modelamos esto como un gráfico de flujos obtenemos la siguiente figura:

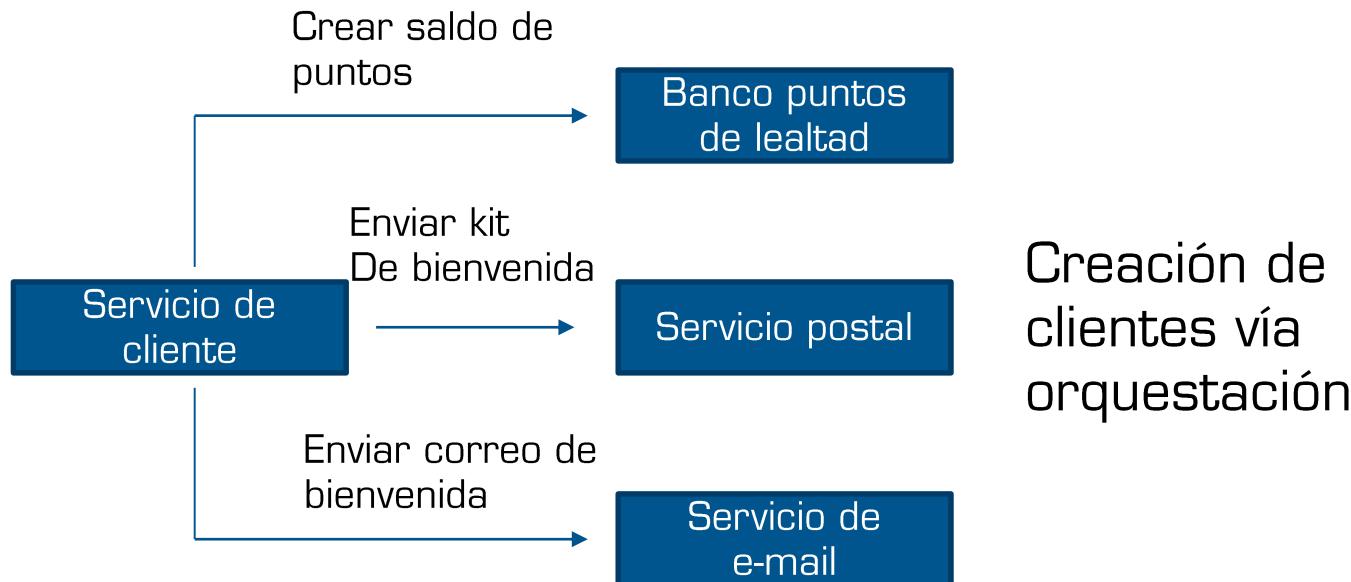


Mensajería y contextos de dominio

- Para implementar este flujo podemos seguir dos estilos de arquitectura.
 - Con **orquestación**, contamos con un cerebro central para orientar y conducir el proceso, al igual que el conductor de una orquesta. Con **coreografía**, le informamos a cada parte del sistema de su trabajo, y se les deja trabajar en los detalles, como los bailarines en un ballet que se encuentran en su camino y reaccionan ante otros a su alrededor.
- Así, yendo al ejemplo de orquestación, probablemente lo más simple sería que el *Servicio de Cliente* sea nuestro cerebro central.
- En la creación, éste habla con el *Banco de puntos de lealtad* o fidelidad, el *Servicio de correo postal* y el *Servicio de correo electrónico* a través de una serie de llamadas de petición/respuesta.

Mensajería y contextos de dominio

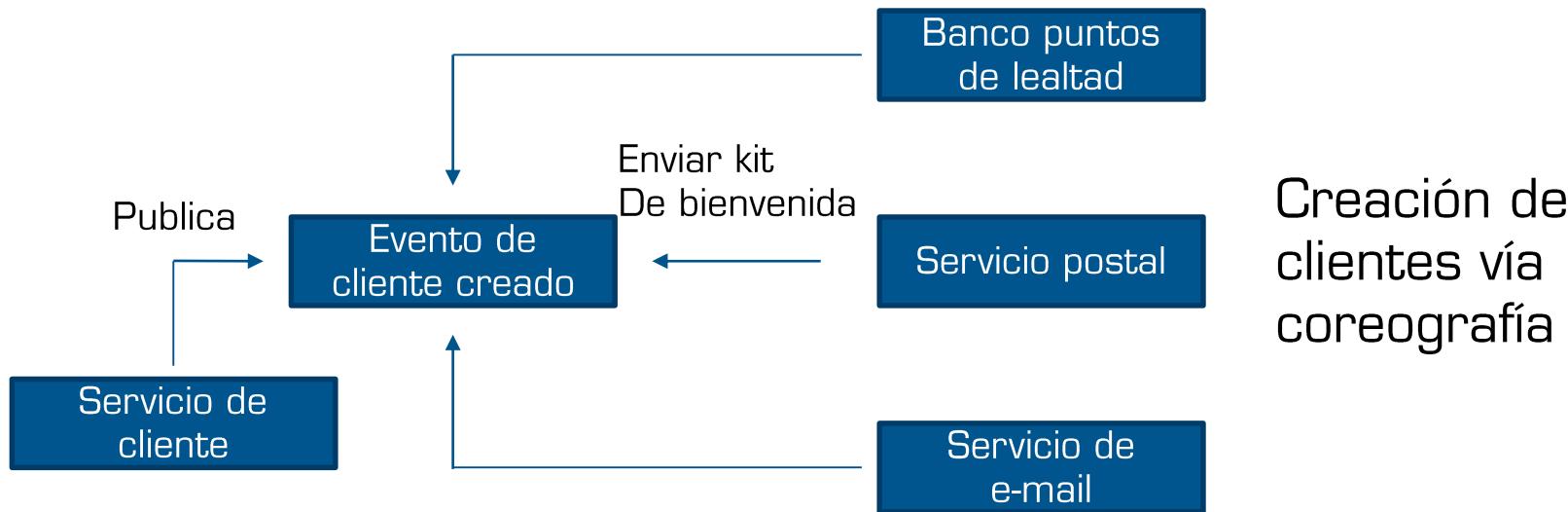
- El *Servicio de Cliente* puede en sí hacer un seguimiento cuando un cliente está en cada etapa del proceso. Asumiendo que utilizamos llamadas sincrónicas de petición/respuesta, podríamos incluso saber si cada fase ha funcionado.



- La desventaja de este enfoque de orquestación, es que el *Servicio de Cliente* se puede convertir en una autoridad de gobierno central demasiado fuerte. Pasa a ser el punto central donde toda la lógica gira alrededor de él.

Mensajería y contextos de dominio

- En cambio, con un enfoque coreografiado, podríamos tener al *Servicio de Cliente* emitiendo un mensaje asíncrono cuando se crea un cliente. Así el *Servicio de Correo Electrónico*, el *Servicio Postal*, y el *Banco de puntos de Fidelidad*, sólo se suscriben a esos eventos y reaccionan en consecuencia.



- Este enfoque es significativamente más desacoplado. Si algún otro servicio necesita llegar a la creación de un nuevo cliente, simplemente tiene que suscribirse a los eventos y hacer su trabajo cuando sea necesario.

Mensajería y contextos de dominio

- La desventaja es que la vista explícita del proceso de negocio que veíamos en la figura anterior del modelado de proceso, ahora sólo se refleja de manera implícita en nuestro sistema.
- Esto implica que se necesita de trabajo adicional para asegurarnos de que alguien pueda controlar y realizar un seguimiento de que hayan sucedido las cosas correctas. Por ejemplo, podemos saber si el banco de puntos de fidelidad tuvo un error y por alguna razón no estableció correctamente los puntos?.
- Para solucionar este problema, normalmente es necesario crear un sistema de monitoreo que refleje explícitamente la vista del proceso de negocio que muestra la figura anterior del modelado de proceso, pero a su vez, luego haga un seguimiento de lo que cada uno de los servicios realiza como entidad independiente que le permite ver excepciones mapeadas al flujo de proceso más explícito.

4

Desarrollo de microservicios con Node.js

Arquitectura de Microservicios con NodeJS.

- Como tal, los microservicios no se crean con Node.js sino que se incluyen al código de Node.js.
- Existen múltiples Frameworks, ya maduros
 - Express
 - Hapi
 - Micro.js
- En este caso el framework elegido es **Hapi**, debido a que está creado específicamente para desarrollar servicios REST de manera muy rápida y flexible.
- También Hapi nos permite crear las pruebas automáticas, usando un framework de pruebas específico: Lab.



Construcción de servicios con Hapi

Construcción de servicios con Hapi



- Antes que nada debemos tener instalado Node.js en nuestro equipo. Como primer paso, creamos un proyecto con Hapi:

```
npm init hapi-ejemplo-1  
npm install --save hapi  
npm install
```

- A continuación creamos el archivo ejecutable *app.js*:

```
var config = require('./config')  
var hapi = require('hapi')  
var server = new hapi.Server()  
server.connection({  
  host: config.server.host,  
  port: config.server.port  })  
var routes = require('./routes')  
routes.init(server, config)  
server.start(function () {  
  console.log('Servidor ejecutándose en:', server.info.uri);  });  
if (module.parent) {  console.log("Llamada de ejecución como módulo")  
  module.exports = server  }
```

Construcción de servicios con Hapi



- Las últimas líneas del archivo, se valida si el script fue llamado como módulo o no. Si fue llamado como módulo, entonces entrega la instancia del servidor creado. Esto nos permitirá utilizar las pruebas con Lab. Además en el archivo anterior llamamos a dos archivos extras:
 - *config.js*: Es el archivo de configuración.
 - *routes.js*: Es el archivo con las URL que definiremos.
- A continuación la definición del archivo *config.js*:

```
var config = {};
if (process.env.NODE_ENV == 'pro') { config = { server: {
  host: 'localhost',
  port: process.env.PORT || 3000    } } }
if (process.env.NODE_ENV == 'dev' || !process.env.NODE_ENV) { config = { server: {
  host: 'localhost',
  port: process.env.PORT || 3010    } } }
if (process.env.NODE_ENV == 'test') { config = { server: {
  host: 'localhost',
  port: process.env.PORT || 3020    }, } }
module.exports = config
```

Construcción de servicios con Hapi



- Como podrán darse cuenta, la configuración varía de acuerdo al ambiente definido (*producción, desarrollo y test*).
- Después si utilizamos una base de datos, podremos incluir diferentes conexiones a ésta. Así podremos tener DB independientes por ambientes.
- La última línea, nos permitirá dejar el archivo como módulo, lo que nos permite llamarlo con *require* y utilizarlo como objeto dentro de *app.js*.

Construcción de servicios con Hapi



- Ahora es el turno de definir las rutas en *routes.js*:

```
exports.init = function (server) { server.route({ method: 'GET',  
path: '/bienvenido/{name}',  
handler: function (request, reply) { reply({  
statusCode: 0,  
mensaje: 'Bienvenida(o) ' + request.params.name }) } })  
server.route({ method: 'POST', path: '/bienvenido', handler: function (request, reply) {  
reply({  
statusCode: 0,  
mensaje: 'Bienvenida(o) ' + request.payload.name }) } }) }
```

- Se agregaron dos rutas: una para GET y otra para POST. Para obtener los parámetros enviados por el usuario y procesarlos, deberás conocer el método de entrada:
 - GET: Deberás obtenerlos desde *request.params.XXX*.
 - POST: Deberás obtenerlos desde *request.payload.XXX*.

Configuración de los scripts de arranque



- Aunque existen muchas formas de ejecutar nuestros script en node, la forma más rápida es utilizando NPM. Para ello editaremos el archivo *package.json*

```
{ "name": "hapi-ejemplo-1",
  "version": "1.0.0",
  "main": "app.js",
  "scripts": { "start": "node app.js", "test": "NODE_ENV='test' ./node_modules/lab/bin/lab -c" },
  "author": "Zankuda",
  "license": "Apache-2.0",
  "dependencies": {
    "hapi": "^8.8.1",  } }
```

- Ahora para ejecutar el servidor, bastará ejecutar el comando:

npm start

- Como verán, dejaremos lista la línea de ejecución de las pruebas automáticas (*npm test*).



Creación de pruebas automáticas.

- Para comenzar agregaremos Lab:

```
npm install --save lab  
npm install --save code
```

- Como parte del Framework de pruebas, se ejecutarán todos los script que estén en la carpeta *test*. Entonces deberás crear esta carpeta en tu directorio de trabajo.

Creación de pruebas automáticas.



- Definiremos un archivo por ruta, entonces comenzamos con *test/bienvenido.js*:

```
var Lab = require('lab')
var lab = exports.lab = Lab.script()
var code = require('code');
var server = require('../app')
lab.experiment('Pruebas de todas las formas de Bienvenida', function() {lab.test('Bienvenido con GET', function(done) { var options = { method: 'GET', url: '/bienvenido/Zankuda' } server.inject(options, function(response) { var result = response.result code.expect(response.statusCode).to.equal(200) code.expect(result.statusCode).to.equal(0) code.expect(result.mensaje).to.equal("Bienvenida(o) Zankuda") done() }) })
lab.test('Bienvenido con POST', function(done) { var options = { method: 'POST', url: '/bienvenido', payload: { name: 'Zankuda' } } server.inject(options, function(response) { var result = response.result code.expect(response.statusCode).to.equal(200) code.expect(result.statusCode).to.equal(0) code.expect(result.mensaje).to.equal("Bienvenida(o) Zankuda") done() }) }) })
```

Creación de pruebas automáticas.



- Como verán las pruebas de ambas rutas son similares, salvo que la forma que son enviados los parámetros de entrada.
- Para la ejecución de las pruebas, sólo deben escribir:

```
npm test
```



Generando la API de BananaTube

- Implementa la API definida usando Node y la arquitectura de microservicios para BananaTube
- Adapta tu SPA para consumir nuestra API



[...]**netmind**

WeKnowIT

Barcelona

C. Almogàvers, 123
08018 Barcelona
Tel. 93 304.17.20
Fax. 93 304.17.22

Madrid

Plaza Carlos Trías Bertrán, 7
28020 Madrid
Tel. 91 442.77.03
Fax. 91 442.77.07

www.netmind.es



GOBIERNO
DE ESPAÑA

MINISTERIO
DE ENERGÍA, TURISMO
Y AGENDA DIGITAL

red.es



ESTRATEGIA DE
EMPRENDIMIENTO Y
EMPLEO JUVENIL
garantía juvenil



Agenda Digital para España



UNIÓN EUROPEA

Fondo Social Europeo
"El FSE invierte en tu futuro"