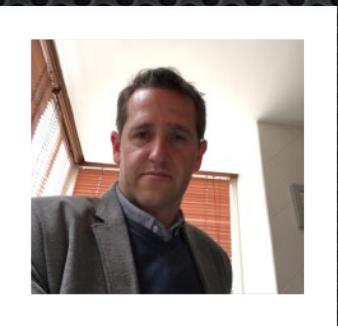
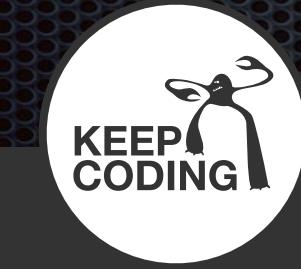


# Desarrollo Backend Avanzado



Express.js



Javier Miguel

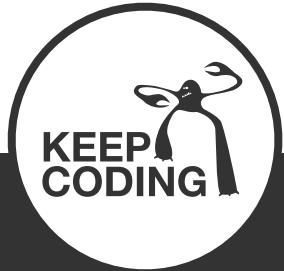
@JavierMiguelG

[jamg44@gmail.com](mailto:jamg44@gmail.com)

CTO & Freelance Developer

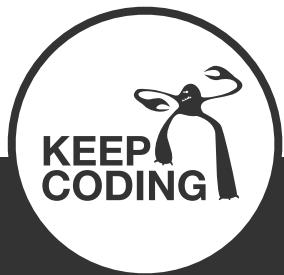


# Debugging



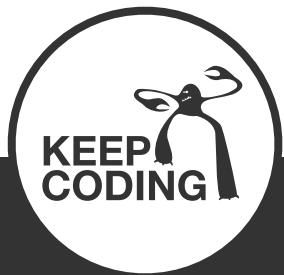


# ■ Debugging - consola



# ■ Debugging - consola

```
//mensaje con marcador para un string, salida en stdout stream
console.log("Hello %s", "World");
//mensaje con marcador para un int, salida en stdout stream
console.log("Number of items: %d", 5);
//mensaje con salida en stdout stream
console.info("Hello Info");
//mensaje con salida en stderr stream
console.error("Hello on Stdout");
//mensaje con salida en stderr stream
console.warn("Hello Warn");
```



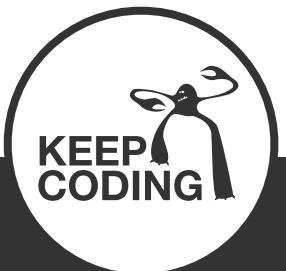
# ■ Debugging - consola

```
//empieza a contar tiempo
console.time("100mil_elementos");

for(var i=0;i<100000000;i++)
{
    let a = 1;
    a = a * i;
}

//para de contar
console.timeEnd("100mil_elementos");

// 100mil_elementos: 462ms
```

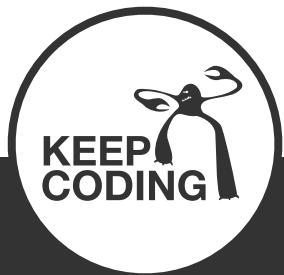


ejemplos/debug

# ■ Debugging - consola

```
//Escribe a stderr 'Trace : ', seguido de nuestro mensaje  
console.trace("Traza");
```

```
Trace: Traza  
at test (/Users/javi/traza.js:21:13)  
at Object.<anonymous> (/Users/javi/traza.js:25:1)  
at Module._compile (module.js:434:26)  
at Object.Module._extensions..js (module.js:452:10)  
at Module.load (module.js:355:32)  
at Function.Module._load (module.js:310:12)  
at Function.Module.runMain (module.js:475:10)  
at startup (node.js:117:18)  
at node.js:951:3
```



ejemplos/debug

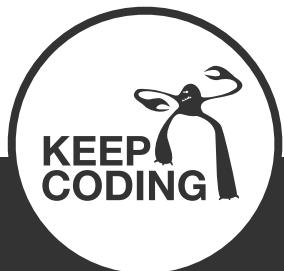
# ■ Debugging - consola

Si ponemos la instrucción `debugger;` en una línea podremos parar la ejecución ahí. Arrancamos con argumento `debug`:

```
$ node debug prueba.js

< Debugger listening on port 5858
debug> . ok
break in prueba.js:1
> 1 "use strict";
  2
  3 let neo = { name: 'Thomas', age: 33, surname: 'Andreson' };
debug>
```

ejemplos/debug



# ■ Debugging - consola

Algunos comandos en debugger:

cont, c - Continue execution

next, n - Step next

step, s - Step in

out, o - Step out

pause - Pause running code

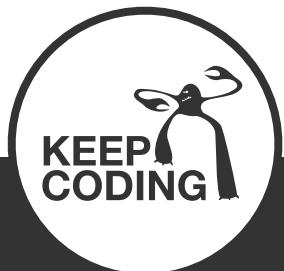
repl - Entra en modo evaluación (Ctrl+c para salir)

help - Muestra comandos

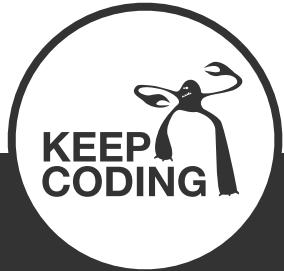
restart - Re-inicia aplicación

kill - Mata la aplicación

scripts - Muestra scripts cargados



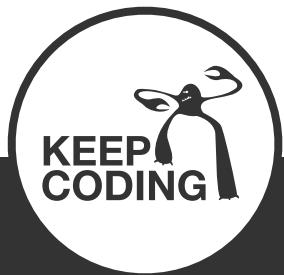
# ■ --inspect y Chrome



--inspect

node --inspect-brk index.js

chrome://inspect



# --inspect



## DevTools

Devices

Pages

Extensions

Apps

Shared workers

Service workers

Other

## Devices

Discover USB devices

Port forwarding...

Discover network targets

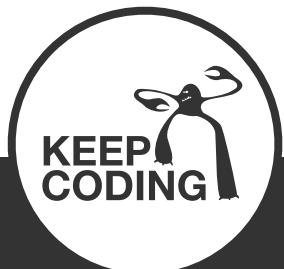
Configure...

[Open dedicated DevTools for Node](#)

## Remote Target #LOCALHOST

### Target (v8.1.0)

 ./bin/www file:///Users/javi/www/cursoronode/keepcoding/\_final/nodepop/bin/www inspect





# ■ Debugging - IDE

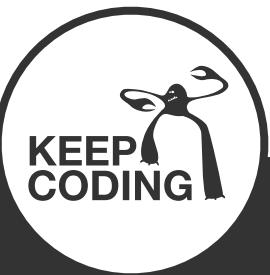


# ■ Debugging - IDE

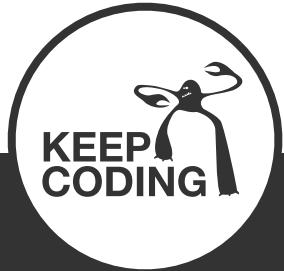
The screenshot shows a Node.js debugger interface with the following details:

- Title Bar:** index.js — ejemplo\_modulos
- Toolbar:** Includes icons for file operations, a search bar, and various debugging controls.
- Code Editor:** Displays the following JavaScript code:

```
1 "use strict";
2
3 const calculadora = require('./calculadora');
4
5 calculadora.marca = 'Siemens';
6
7 console.log(calculadora.suma(5,3));
8
9 const otraCalculadora = require('./calculadora');
10
11 console.log(otraCalculadora.marca);
```
- Variables View:** Shows the current state of variables. The `calculadora` object is expanded, showing its properties: `suma`, `resta`, `mult`, and `division`. The `calculadora.marca` property is highlighted in green, indicating it is being inspected.
- Call Stack View:** Shows the stack trace with the following entries:
  - (anonymous function) at index.js:5:1
  - Module.\_compile at module.js:632:14
  - Module.\_extensions..js at module.js:646:10



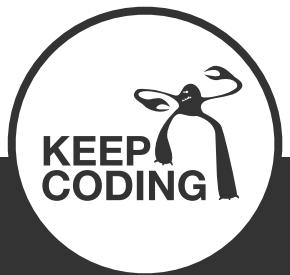
# Upload



# ■ Upload

<https://github.com/expressjs/multer>

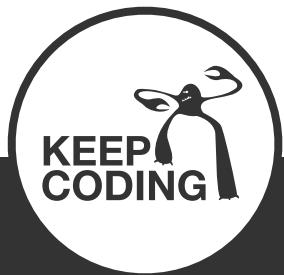
npm install multer



# ■ Upload

```
import multer from 'multer'

const storage = multer.diskStorage({
  destination: function(req, file, callback) {
    const ruta = path.join(__dirname, '..', 'public', 'images', 'products')
    callback(null, ruta)
  },
  filename: function(req, file, callback) {
    const filename = `${file.fieldname}-${Date.now()}-${file.originalname}`
    callback(null, filename)
  }
})
```



# Upload

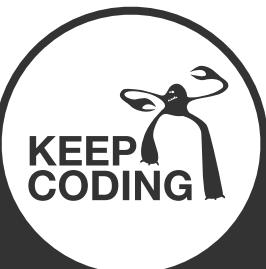
POST ▾ http://localhost:3000/api/v1/anuncios

Params Authorization Headers (2) **Body** ● Pre-request Script Tests

form-data  x-www-form-urlencoded  raw  binary

KEY	VALUE	DESCRIP
nombre	Taza	
venta	1	
precio	240	
tags	lifestyle	
tags	breakfast	
foto	<input type="file"/> Elegir archivos vlcsnap-2015-09-...10h48m16s638.png	
Key	Value	Descri

Body Cookies Headers (6) Test Results Status: 200





# ■ Internacionalización y localización

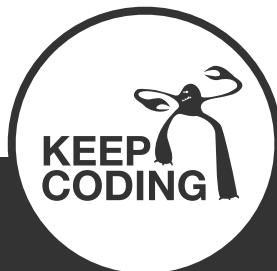
Nuestra web, en varios idiomas



# ■ Internacionalización y localización

El objetivo de la internacionalización y la localización es permitir a un único sitio web ofrecer sus contenidos en diferentes idiomas y formatos adaptados a su audiencia.

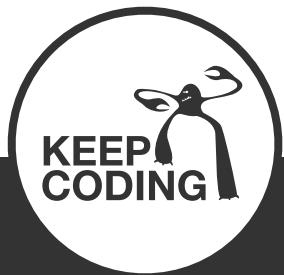
No sólo vale con traducir, también hay que formatear los contenidos en función de las preferencias del usuario.



# Ejemplo

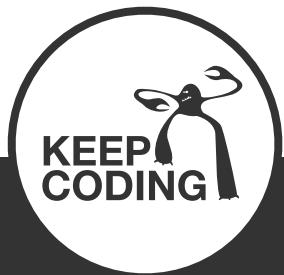
Formato de fecha España: dd/mm/yyyy

Formato de fecha USA: mm/dd/yyyy



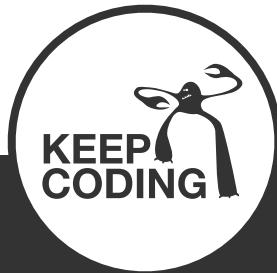
# ■ Definiciones

- **Internacionalización:** preparar el software para que sea localizable (trabajo de desarrolladores).
- **Localización:** escribir la traducción y los formatos locales (trabajo de traductores)

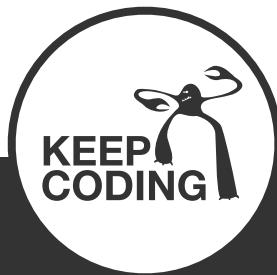


# ■ ¿Cómo sé qué idioma quiere el usuario?

- A través de la cabecera “Accept-Language” de una petición HTTP los navegadores envían la preferencia de idioma de un usuario.
- Si no se envía la cabecera, devolver el idioma por defecto.
- Existen otras técnicas como la geolocalización por IP (si la IP es de España, redirigir al sitio en español) aunque son más complejas.
- [i18n-node](#) lo gestiona por nosotros automáticamente



# Ejemplo de cabecera





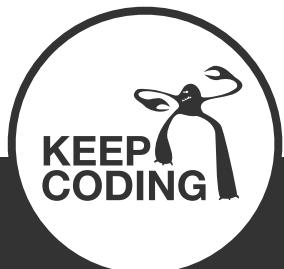
Let's code  
Jesse



# Cómo traducir en Node.js

- Instalar i18n-node - **npm install i18n**
- Inicializar i18n con **i18n.configure({ ... })**
- Crear archivos de mensajes en carpeta **locales**
- En nuestro código, usar la función **i18n.\_()**

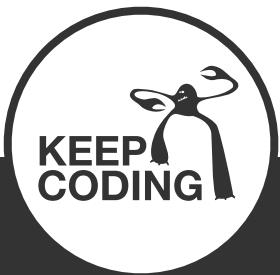
<https://github.com/mashpie/i18n-node>



# ■ Configuración

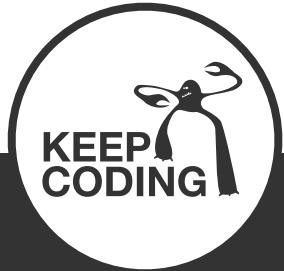
```
const i18n = require("i18n");

i18n.configure({
  directory: path.join(__dirname, '../locales'),
  defaultLocale: 'en',
  queryParameter: 'lang',
  autoReload: true,
});
```



# ■ Configuración de Express

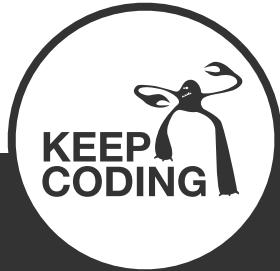
```
app.use(i18n.init);
```



# ■ Configuración scripts

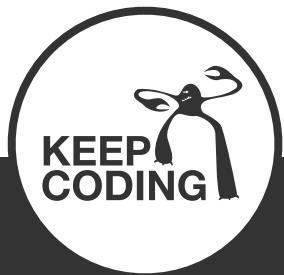
Poner locale para scripts:

```
i18n.setLocale(defaultLocale); // default locale for scripts
```



# ■ Object notation

```
./locales/en.json
{
    "EXAMPLE": {
        "TEXT": "Text",
        "HELLO": "Hello %s, i'm from %s",
        "NAME_AND AGE": "My name is {{name}}",
        "MOUSE": {
            "one": "%s Mouse",
            "other": "%s Mice"
        }
    }
}
```



# ■ Object notation

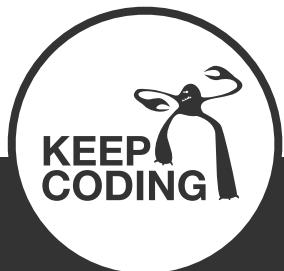
```
_('EXAMPLE.TEXT')
```

```
_('EXAMPLE.HELLO', 'Pepe', 'Madrid')
```

```
_('EXAMPLE.NAME_AND_AGE', {name: 'Javier', age: 33})
```

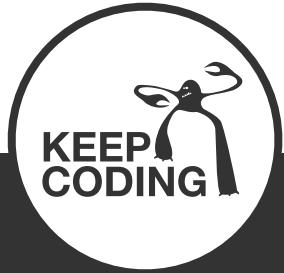
```
_({phrase: 'EXAMPLE.TEXT', locale: 'es'})
```

```
_n('EXAMPLE.MOUSE', 2)
```





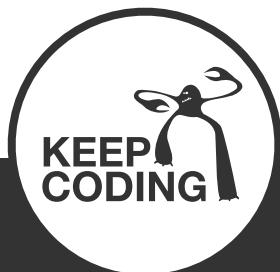
# API



HTTP pone a nuestra disposición varios métodos, de los cuales generalmente hacemos distintos usos:

- **GET** para pedir datos, es idempotente (p.e. listas)
- **POST** para crear un recurso (p.e. crear un usuario)
- **PUT** para actualizar, es idempotente (p.e. guardar un usuario existente)
- **DELETE** eliminar un recurso, es idempotente (p.e. eliminar un usuario)
- **PATCH** actualización parcial de un recurso

\* idempotente: si lo ejecutas varias veces los resultados no cambian



# ■ API

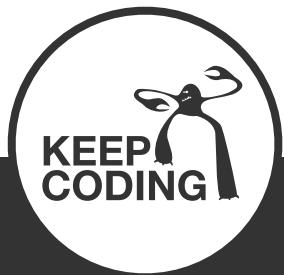
En un API respondemos con formato JSON

```
// agents list handler
app.get('/api/agents', async (req, res, next) => {
  try {

    const agents = await Agent.find()

    res.json({ results: agents })

  } catch (error) {
    next(error)
  }
})
```



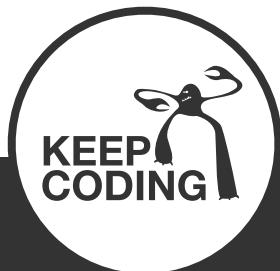
# ■ API

En un API los errores deberían ser devueltos en formato JSON

```
// error handler
app.use((err, req, res, next) => {
  res.status(err.status || 500)

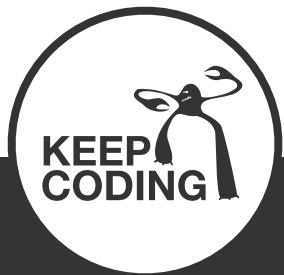
  // API error, send with JSON format
  if (req.originalUrl.startsWith('/api/')) {
    return res.json({ error: err.message })
  }

  ...
  res.render('error')
})
```





# ■ Documentación API



# ■ OpenAPI

Especificación

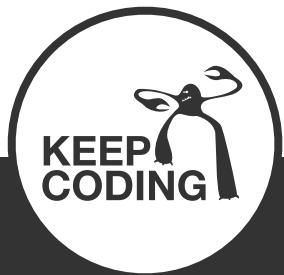
<https://swagger.io/specification/>

Editor

<https://swagger.io/tools/swagger-editor/>

Visor de doc

<https://swagger.io/tools/swagger-ui/>



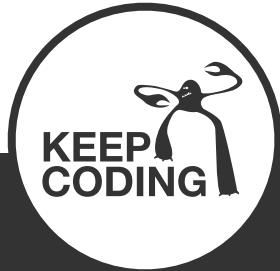
# ■ OpenAPI en Express

Generar especificación desde código o YAML

- <https://github.com/Surnet/swagger-jsdoc>

Middleware de Express con Swagger UI

- <https://github.com/scottie1984/swagger-ui-express>



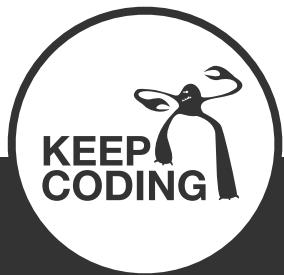
# ■ OpenAPI

Alternativa interesante como visor de doc OpenAPI

- <https://github.com/Redocly/redoc>

Describir y generar un API

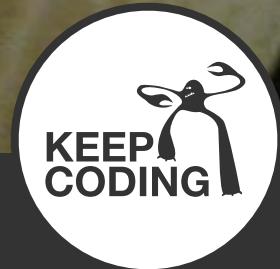
- <https://typespec.io/>





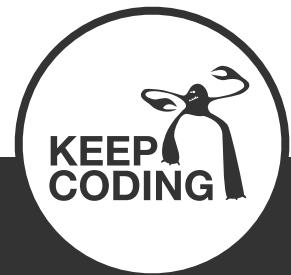
# ■ Autenticación en APIs

## API Key, Tokens, JWT



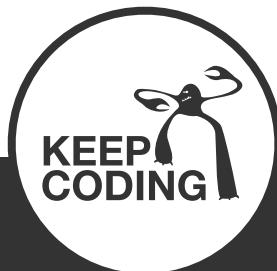


# API Key

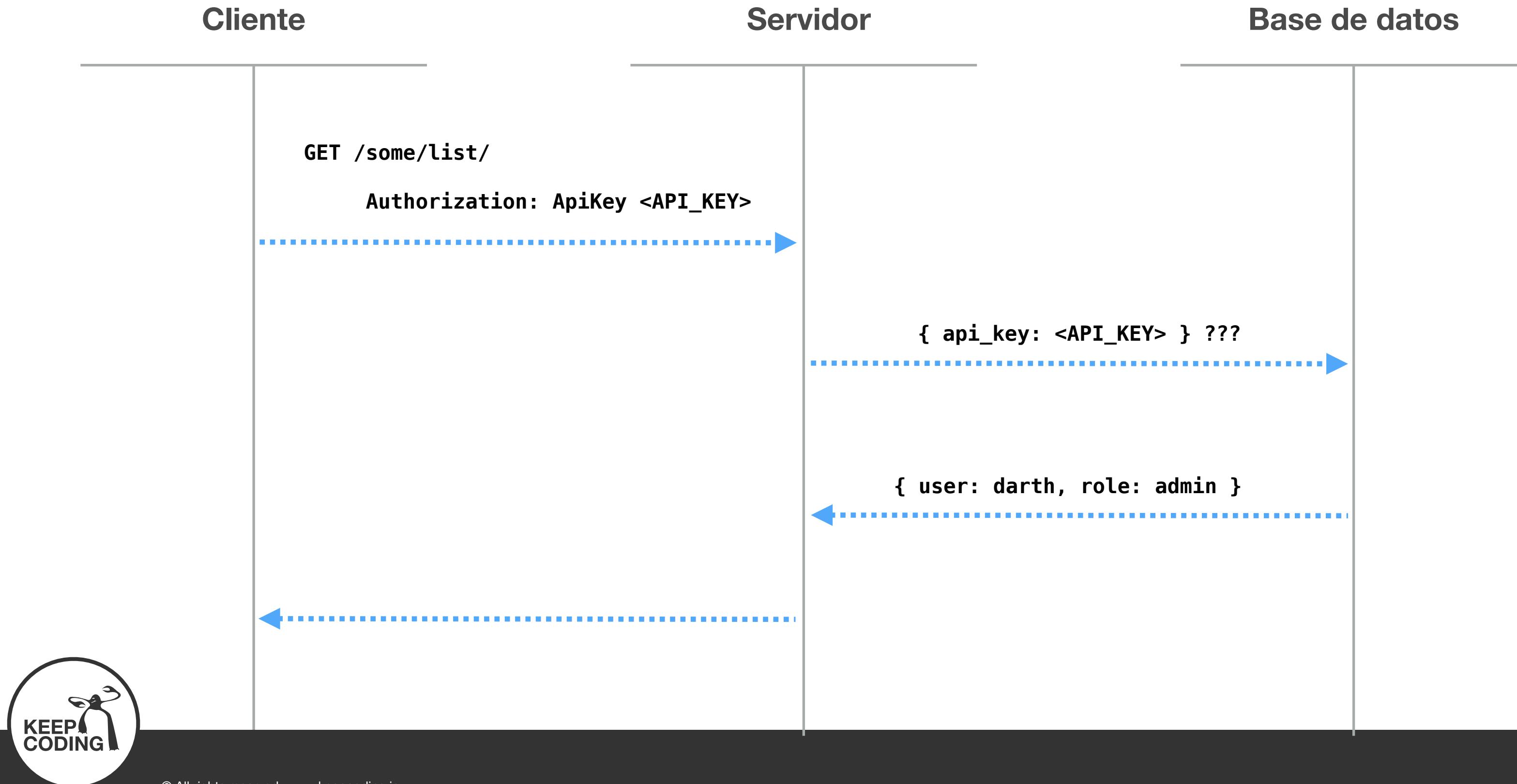


# ■ Autenticación por API Key

- La autenticación con API Key se basa en enviar en todas las peticiones un API Key (como cabecera o parámetro GET)
- Éste API Key suele ser suministrada por el servicio tras el registro de usuario.
- El API Key es único e identifica al usuario de manera única.
- El API Key siempre es la misma, no cambia ni hay que renovarla.

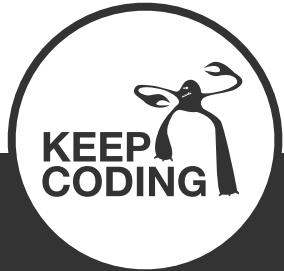


# ■ Autenticación por API Key

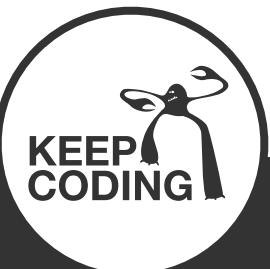




# Tokens (y JWT)



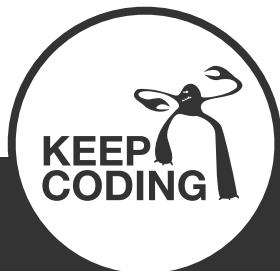
# JWT (JSON Web Tokens)



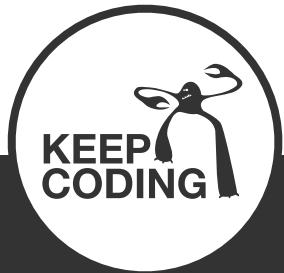
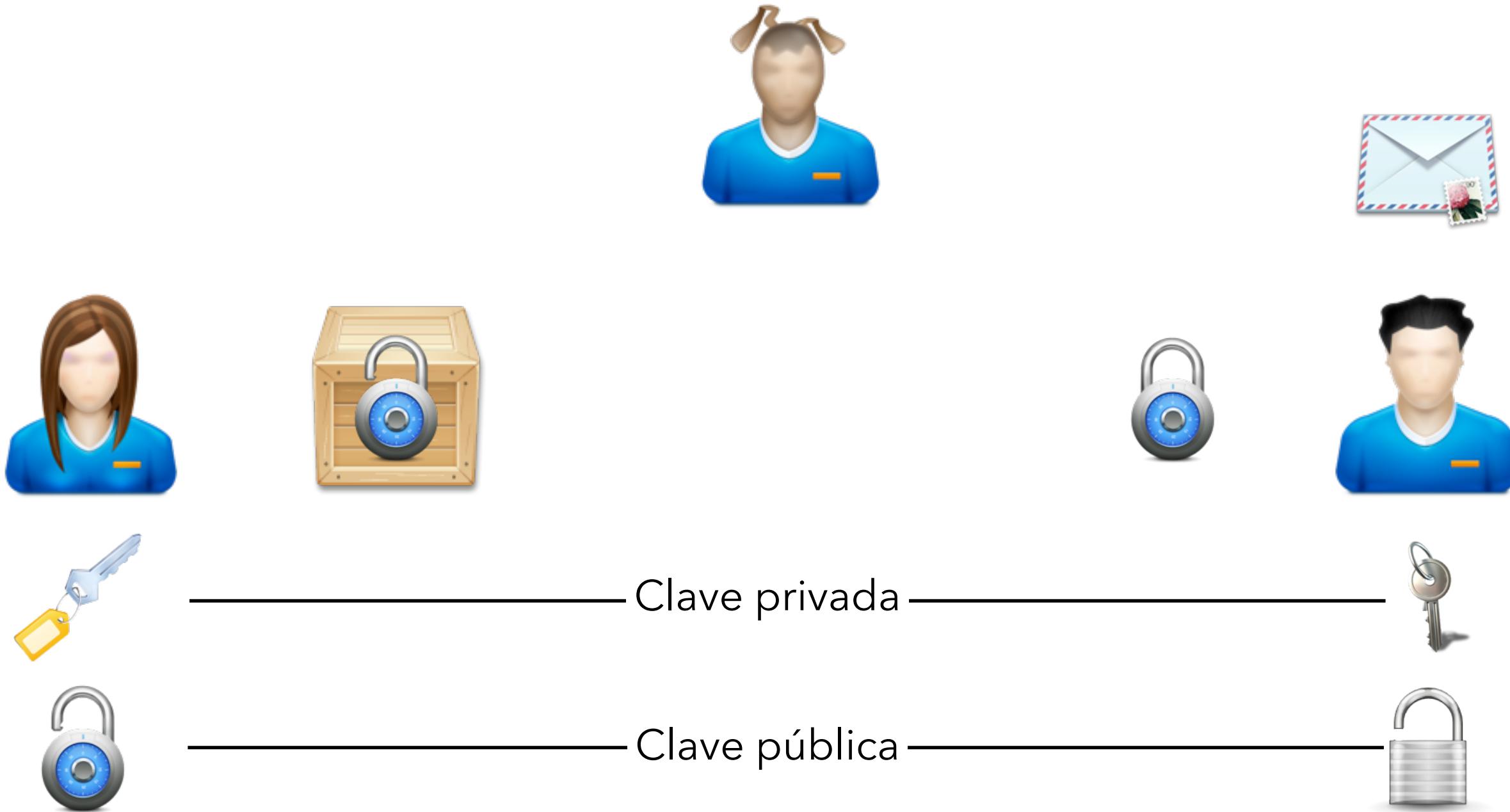
KEEP  
CODING

# ■ JWT (JSON Web Tokens)

- JWT es un estándar abierto ([RFC 7519](#))
- Define una forma segura y compacta de transmitir información entre diferentes partes con un objeto JSON
- La información puede ser verificada porque va firmada digitalmente
- Se pueden firmar los tokens con HMAC (algoritmo secreto) o con un par de claves RSA (pública/privada)



# Cifrado asimétrico



# Estructura de un JWT

Un JWT está formado por tres partes:

- Cabecera: JSON en formato base64 que indica el tipo de token y el algoritmo usado para firmarlo
- Payload: JSON de datos en formato base64. Contiene los claims (claves del diccionario de datos).
- Firma: firma del token resultante de aplicar la siguiente fórmula:

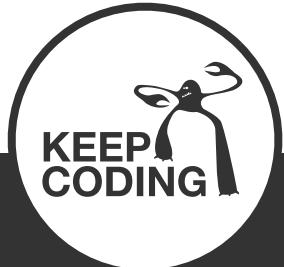
HMACRSA256( base64(header) + base64(payload) + <secret> )



# Ejemplo

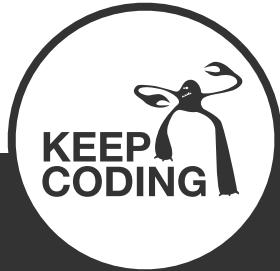
eyJhbGciOiJIUzI1NilsInR5cCl6IkpxVCJ9.eyJ1c2VybmFtZSI6IkRhcnRoliwiX2IkIjoiczZzZDZzZHV5ZHM2N3NkNmRzIwicm9sZSI6ImFkbWluliwiaWF0IjoxNTA4MDAwNTUxfQ.FfEP4sl3Q-Ggp\_HPJaCofpBSO6c6w9\_MKtYuecWkvqA

cabecera.payload.firma



# ■ ¿Por qué usar JWT?

- Evitamos guardar la sesión del usuario en la base de datos.
- Es el usuario quien nos envía su información en todo momento.
- Es seguro, porque podemos validar que el token no ha sido alterado desde que nosotros lo enviamos.



# ■ Autenticación por JWT

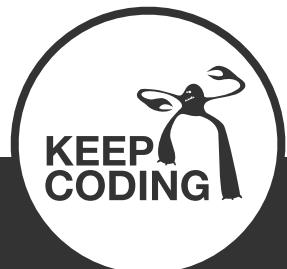
Cliente

Servidor

**POST /login/**

{ token: <JWT> }

**GET /products/in/cart/  
Authorization: <JWT>**



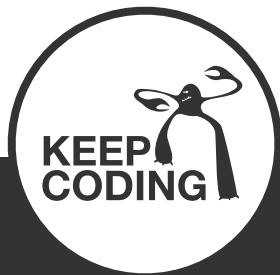


# Tareas en segundo plano

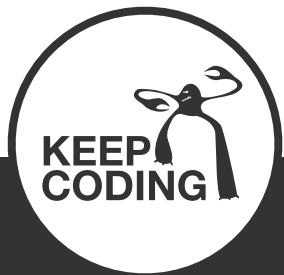
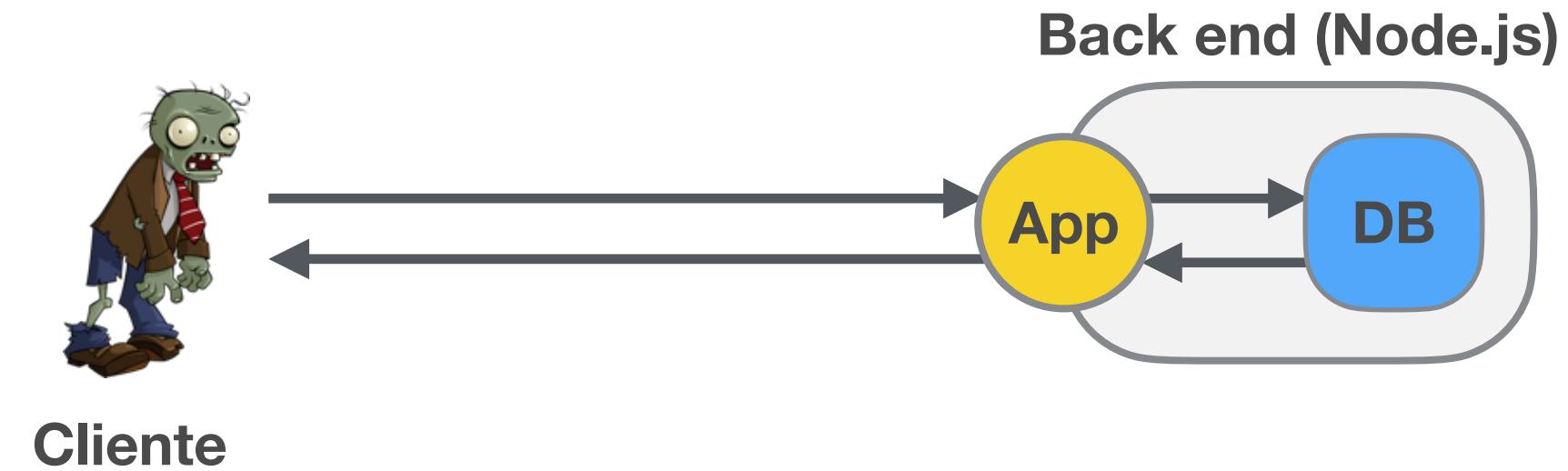


# Tareas lentes o pesadas

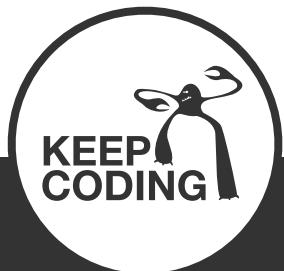
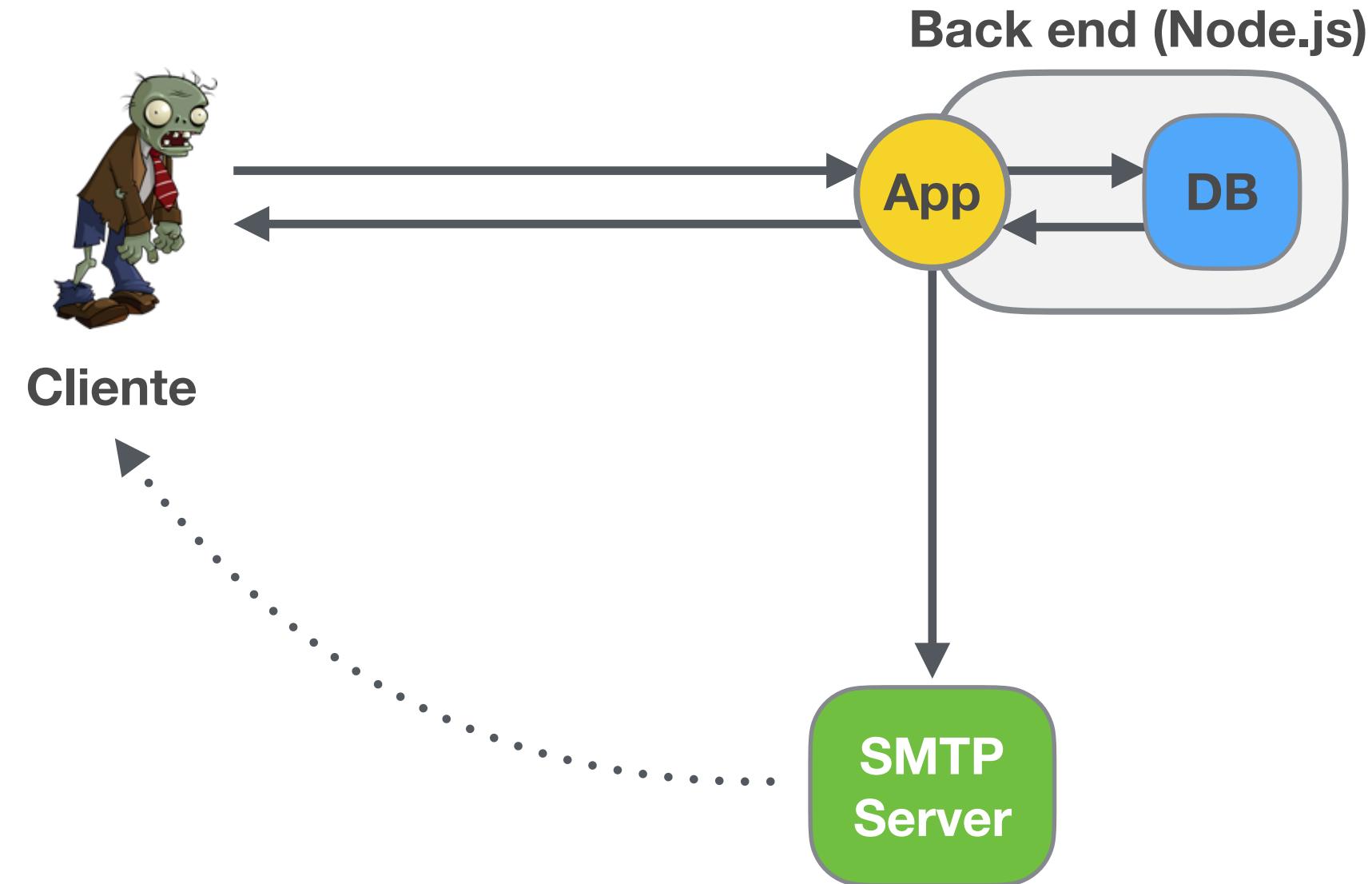
- Las respuestas a las peticiones HTTP deben ser casi inmediatas
- Un backend lento, es un backend mal hecho
- Pero a veces hay que hacer cosas que son lentes como **enviar correo, redimensionar una imagen, leer archivos** gigantes...
- Todas esas tareas lentes, deberemos hacerlas de manera diferida



# ■ Flujo básico de una petición HTTP



# Envío de un e-mail desde un back-end

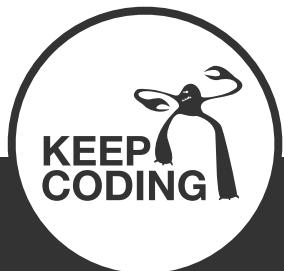
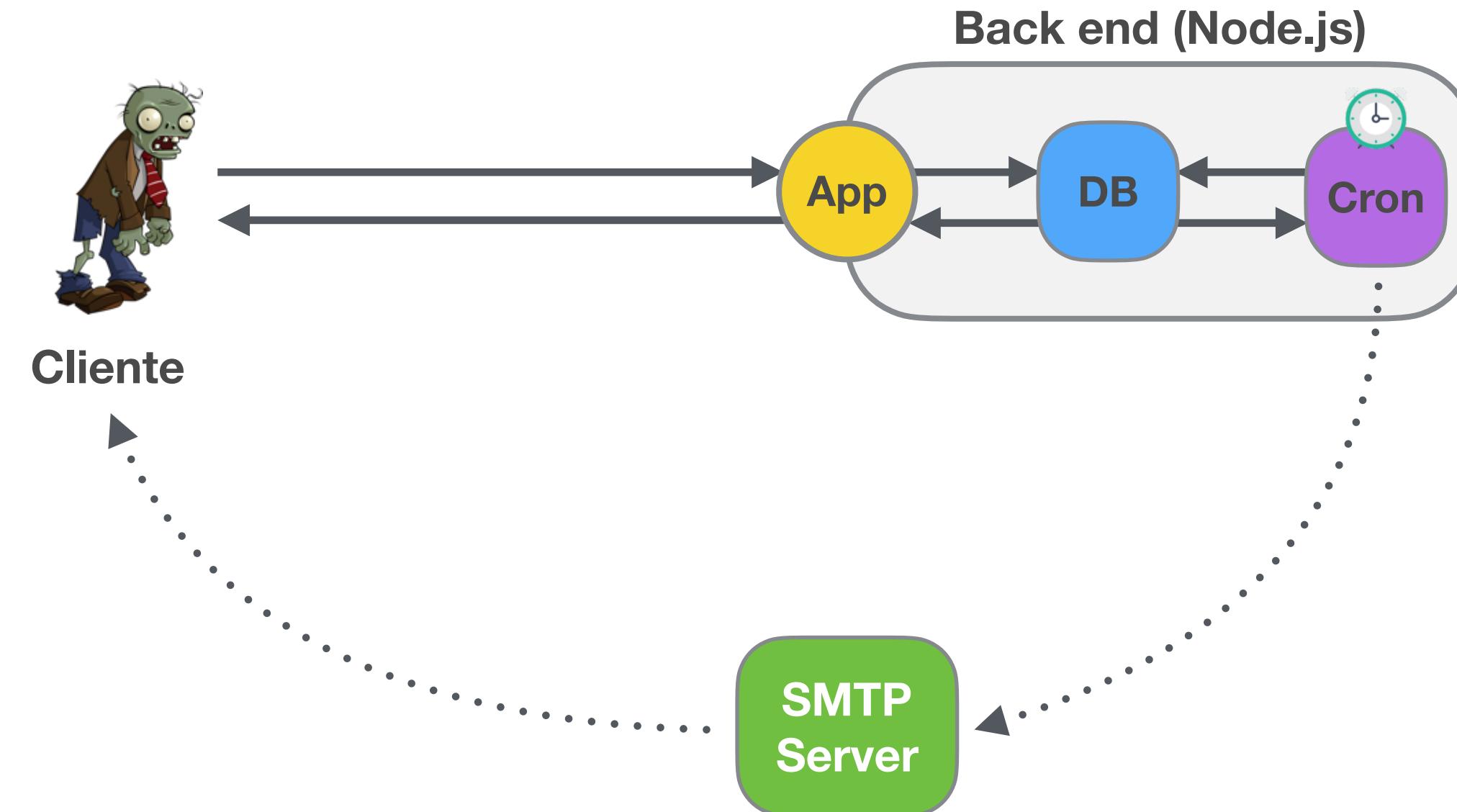




Let's code  
Jesse



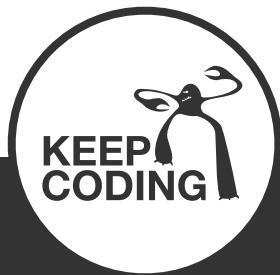
# Envío con tarea programada



# ■ Comandos personalizados en NPM

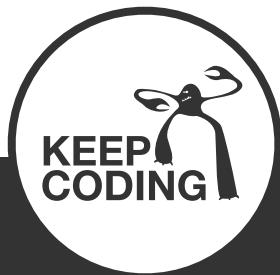
NPM nos permite crear nuestro propios comandos personalizados

Esto es útil para ejecutarlos desde la consola o a través de tareas programadas (con cron)

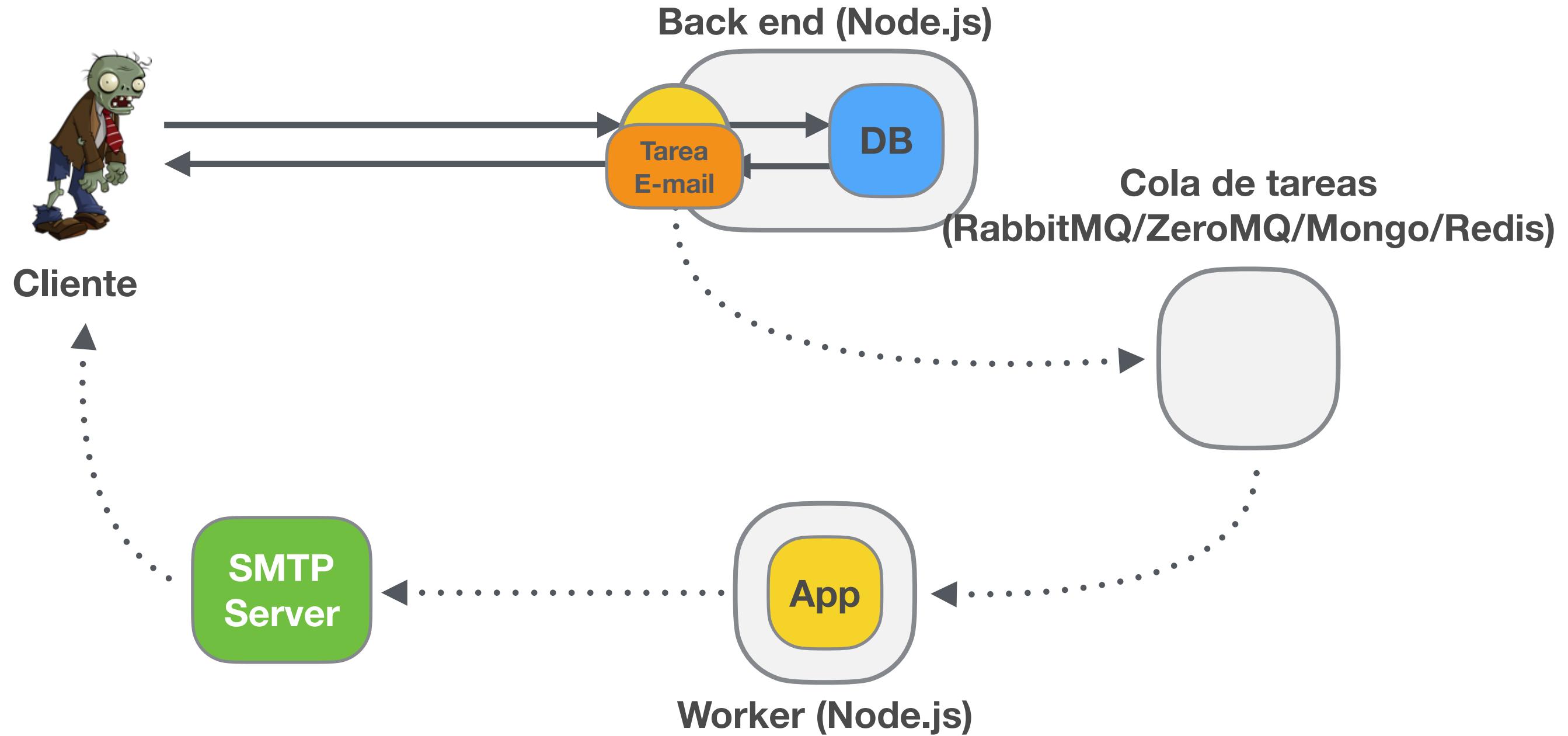


# ■ Comandos personalizados en NPM

Tan sólo hay que crear una entrada en **package.json**,  
apartado **scripts**



# Envío de un e-mail con tareas en background

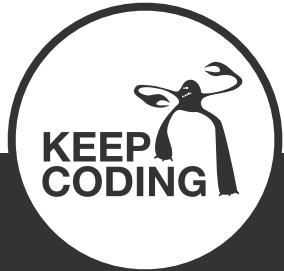


KEEP  
CODING

# Tareas en background con RabbitMQ

RabbitMQ es un software de envío de mensajes que implementa el protocolo AMQP

<https://www.rabbitmq.com>

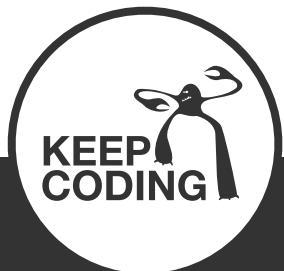


# Tareas en background con RabbitMQ

Podemos usarlo desde:

- **Instalación Win/linux/Mac**
  - <https://www.rabbitmq.com/download.html>
- **Docker**
  - [https://hub.docker.com/\\_/rabbitmq/](https://hub.docker.com/_/rabbitmq/)
- **CloudAMQP**
  - <https://www.cloudamqp.com/>

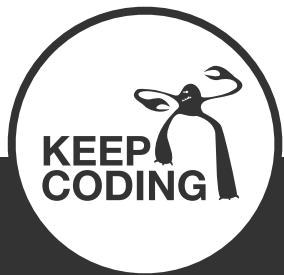
Y usarlo desde amazon, cloud foundry, etc...



# Tareas en background con RabbitMQ

Docker

```
docker run  
-d --hostname= mq --name mq -p 8080:15672 -p 5672:5672  
rabbitmq:3-management
```



# Tareas en background con RabbitMQ

npm i amqplib

- **Uso con Node.js**
  - <http://www.squaremobius.net/amqp.node/>
- **Ejemplos estrategias**
  - <https://github.com/squaremo/amqp.node/blob/master/examples/tutorials/README.md>



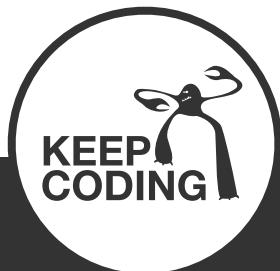
# Tareas en background con RabbitMQ

¿Que tamaño pueden tener los mensajes?

Teóricamente:  $2^{64}$  bytes

En la práctica depende de las máquinas que usemos y la conexión.

Si los mensajes son grandes mejor enviar una ruta a un servidor de ficheros o similar.



# Tareas en background con otros motores

Usando MongoDB:

<https://github.com/chilts/mongodb-queue>

<https://github.com/scttnlsn/monq>

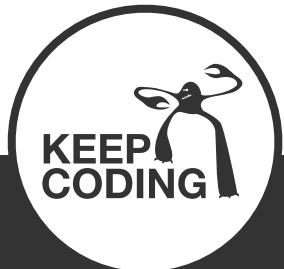
<https://github.com/agenda/agenda>

Usando Redis:

<https://github.com/OptimalBits/bull>

<https://github.com/Automattic/kue>

<https://github.com/bee-queue/bee-queue>



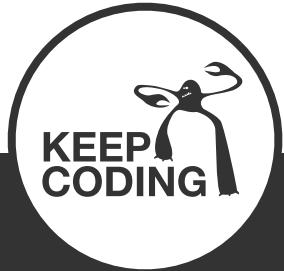


Let's code  
Jesse





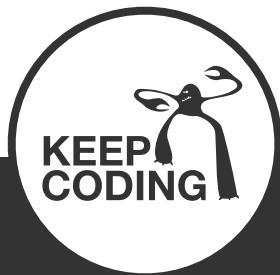
# ■ WebSockets





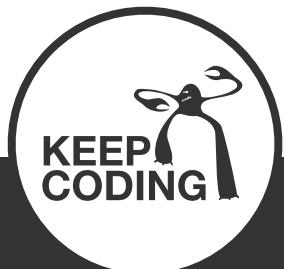
- **WebSockets** es una tecnología avanzada que hace posible abrir una sesión de comunicación interactiva entre el navegador del usuario y un servidor

Fuente: MDN <https://developer.mozilla.org/es/docs/WebSockets-840092-dup>



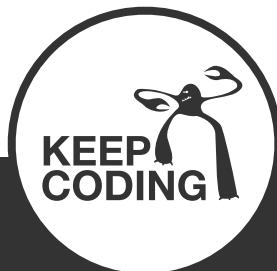
# ■ Compatibilidad navegadores

Protocolo	Fecha versión preliminar	Internet Explorer	Implementation status					
			Firefox <sup>17</sup> (PC)	Firefox (Android)	Chrome (PC, Mobile)	Safari (Mac, iOS)	Opera (PC, Móvil)	Android Browser
<a href="#">hixie-75</a> ↗	February 4, 2010				4	5.0.0		
<a href="#">hixie-76</a> ↗	May 6, 2010		4.0		6	5.0.1	11.00	
<a href="#">hybi-00</a> ↗	May 23, 2010		(disabled)				(disabled)	
<a href="#">7 hybi-07</a> ↗	April 22, 2011		6 <sup>18</sup>					
<a href="#">8 hybi-10</a> ↗	July 11, 2011		7 <sup>19</sup>	7	14 <sup>20</sup>			
<a href="#">13 RFC 6645</a>	December, 2011	10 <sup>21</sup>	11	11	16 <sup>22</sup>	6	12.10 <sup>23</sup>	4.4



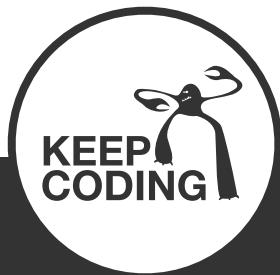
# ■ Características

- El protocolo es un estándar ([RFC 6455](#))
- Está diseñada para ser implementada en navegadores y servidores web, pero puede utilizarse por cualquier aplicación cliente/servidor
- Usa los puertos habituales HTTP (80,443)
- Atraviesa firewalls y proxies



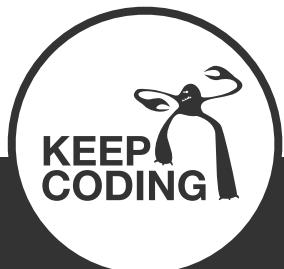
# Casos prácticos

- Juegos online multijugador
- Aplicaciones de chat
- Rotativos de información deportiva
- Actualizaciones en tiempo real de las actividades de tus amigos



# ■ Algunas implementaciones

- **Node.js**
  - Socket.IO
  - WebSocket-Node
  - ws
- **Python**
  - pywebsocket
  - Tornado
- **Ruby**
  - EventMachine
- **Java**
  - Jetty
- **Erlang**
  - Shirasu
- **C++**
  - libwebsockets
- **.NET**
  - SuperWebSocket

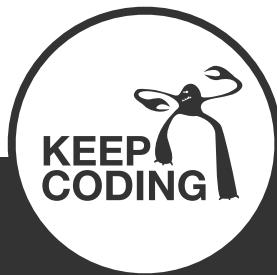


# Con socket.io

```
var app = require('express')();
var server = require('http').Server(app);

app.get('/', (req, res) => res.sendFile(__dirname + '/index.html'));

server.listen(3000, () => console.log('listening on *: 3000'));
```



# Con socket.io

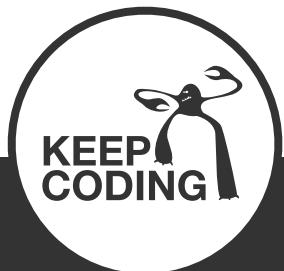
```
var app = require('express')();
var server = require('http').Server(app);

app.get('/', (req, res) => res.sendFile(__dirname + '/index.html'));

server.listen(3000, () => console.log('listening on *: 3000'));

var io = require('socket.io')(server);

io.on('connection', function(socket){
  socket.on('chat message', function(msg){
    io.emit('chat message', msg);
  });
});
```



# Con socket.io

```
var app = require('express')();
var server = require('http').Server(app);

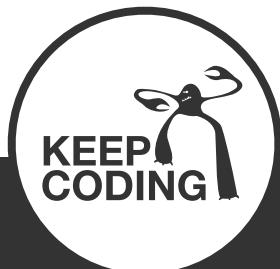
app.get('/', (req, res) => res.sendFile(__dirname + '/index.html'));

server.listen(3000, () => console.log('listening on *: 3000'));

var io = require('socket.io')(server);

io.on('connection', function(socket){
  socket.on('chat message', function(msg){
    io.emit('chat message', msg);
  });
});
```

Le pasamos a socket.io nuestro server para que se instale en el mismo puerto



# ■ Con socket.io

```
var app = require('express')();
var server = require('http').Server(app);

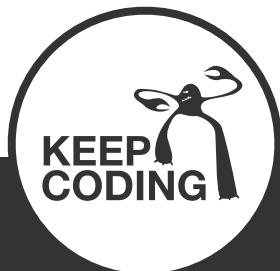
app.get('/', (req, res) => res.sendFile(__dirname + '/index.html'));

server.listen(3000, () => console.log('listening on *: 3000'));

var io = require('socket.io')(server);

io.on('connection', function(socket){
  socket.on('chat message', function(msg){
    io.emit('chat message', msg);
  });
});
```

Ante cada conexión de un cliente  
se crea un socket



# ■ Con socket.io

```
var app = require('express')();
var server = require('http').Server(app);

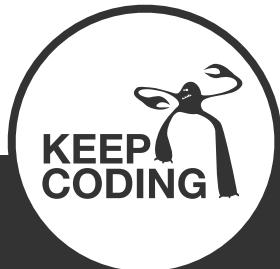
app.get('/', (req, res) => res.sendFile(__dirname + '/index.html'));

server.listen(3000, () => console.log('listening on *: 3000'));

var io = require('socket.io')(server);

io.on('connection', function(socket){
  socket.on('chat message', function(msg){
    io.emit('chat message', msg);
  });
});
```

Escuchamos eventos del socket



# ■ Con socket.io

```
var app = require('express')();
var server = require('http').Server(app);

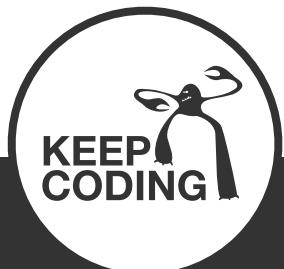
app.get('/', (req, res) => res.sendFile(__dirname + '/index.html'));

server.listen(3000, () => console.log('listening on *: 3000'));

var io = require('socket.io')(server);

io.on('connection', function(socket){
  socket.on('chat message', function(msg){
    io.emit('chat message', msg);
  });
});
```

Emitimos mensajes



# Con socket.io

```
// sending to sender-client only
socket.emit('message', "this is a test");

// sending to all clients, include sender
io.emit('message', "this is a test");

// sending to all clients except sender
socket.broadcast.emit('message', "this is a test");

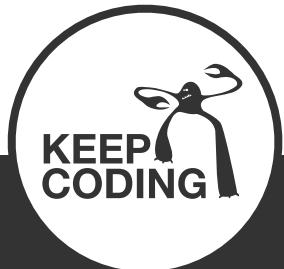
// sending to all clients in 'game' room(channel) except sender
socket.broadcast.to('game').emit('message', 'nice game');

// sending to all clients in 'game' room(channel), include sender
io.in('game').emit('message', 'cool game');

// sending to sender client, only if they are in 'game' room(channel)
socket.to('game').emit('message', 'enjoy the game');

// sending to all clients in namespace 'myNamespace', include sender
io.of('myNamespace').emit('message', 'gg');

// sending to individual socketid
socket.broadcast.to(socketid).emit('message', 'for your eyes only');
```





Let's code  
Jesse



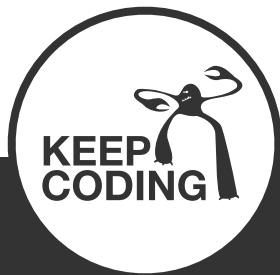


# ■ Microservicios



# ■ Introducción

La arquitectura de microservicios nace como respuesta a la problemática del mantenimiento y evolución de grandes aplicaciones monolíticas.

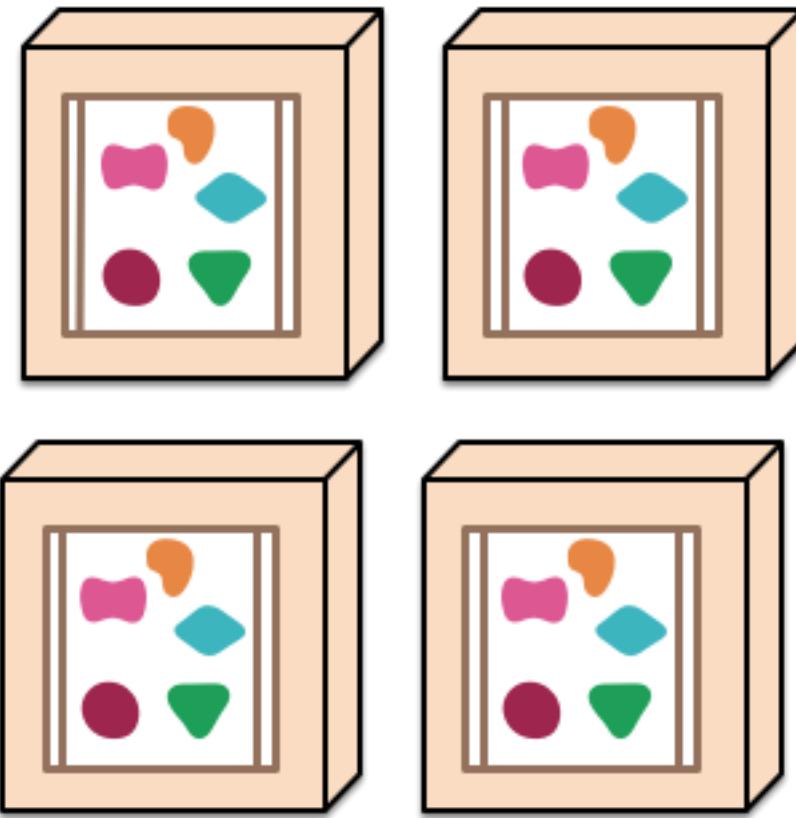




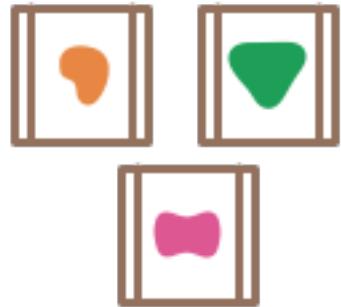
*A monolithic application puts all its functionality into a single process...*



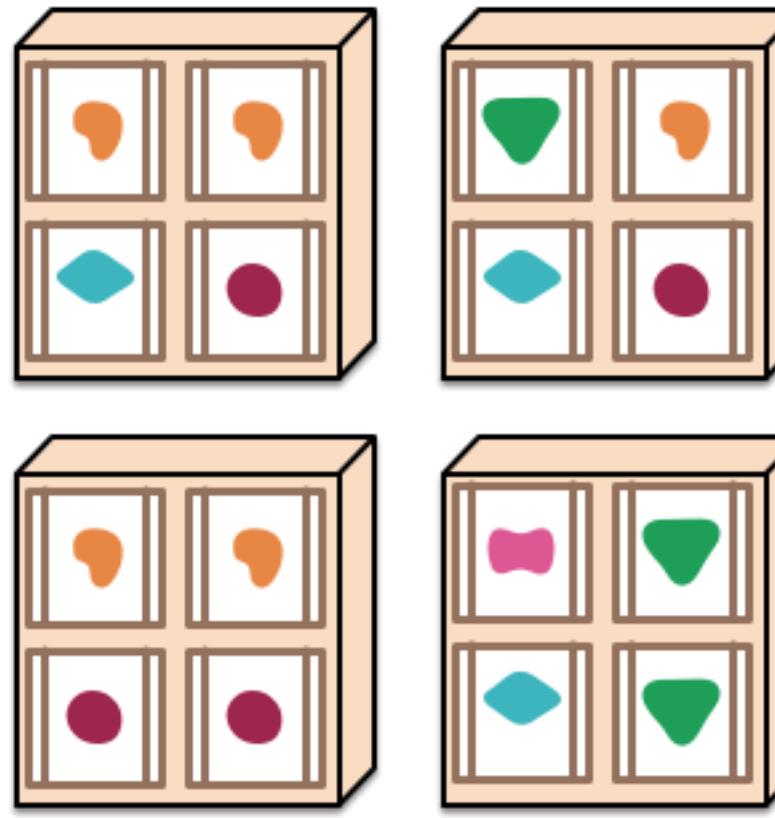
*... and scales by replicating the monolith on multiple servers*



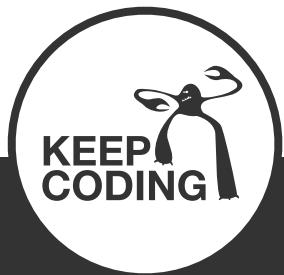
*A microservices architecture puts each element of functionality into a separate service...*

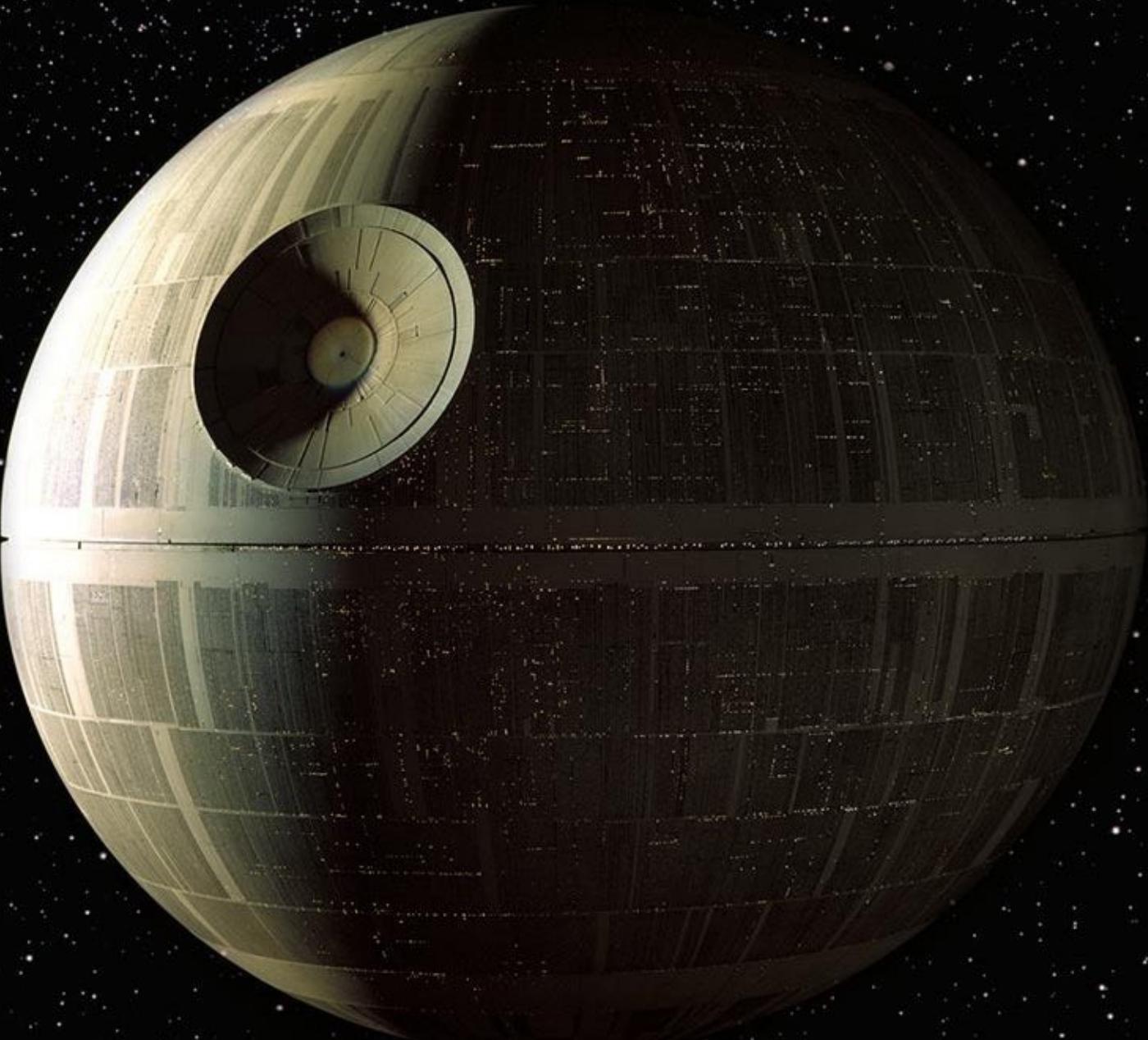


*... and scales by distributing these services across servers, replicating as needed.*

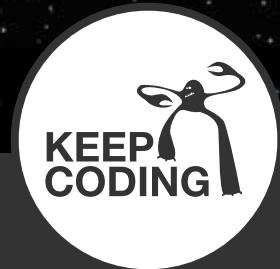


<http://martinfowler.com/articles/microservices.html>



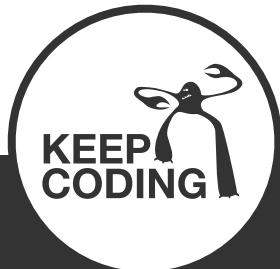


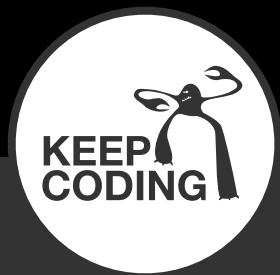
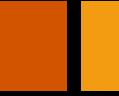
# Aplicación monolítica



# ■ Problemática de las aplicaciones monolíticas

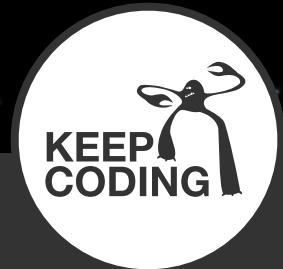
- Si algo falla en la aplicación, todo el sistema se cae
- Son difícil de escalar a nivel de infraestructura
- Es difícil de escalar a nivel de equipo de desarrollo (hasta que todos los desarrolladores saben cómo funciona todo el monolito)
- Es muy difícil mantener la interdependencia de relaciones y un código limpio (spaghetti code, merge conflicts, etc...)
- Es la manera natural de desarrollar (todo software tiende a monolito)





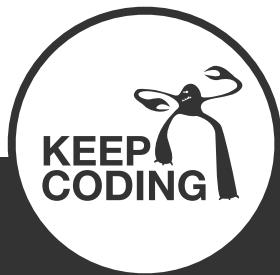


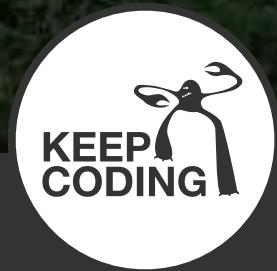
# Microservicios



# Ventajas de los microservicios

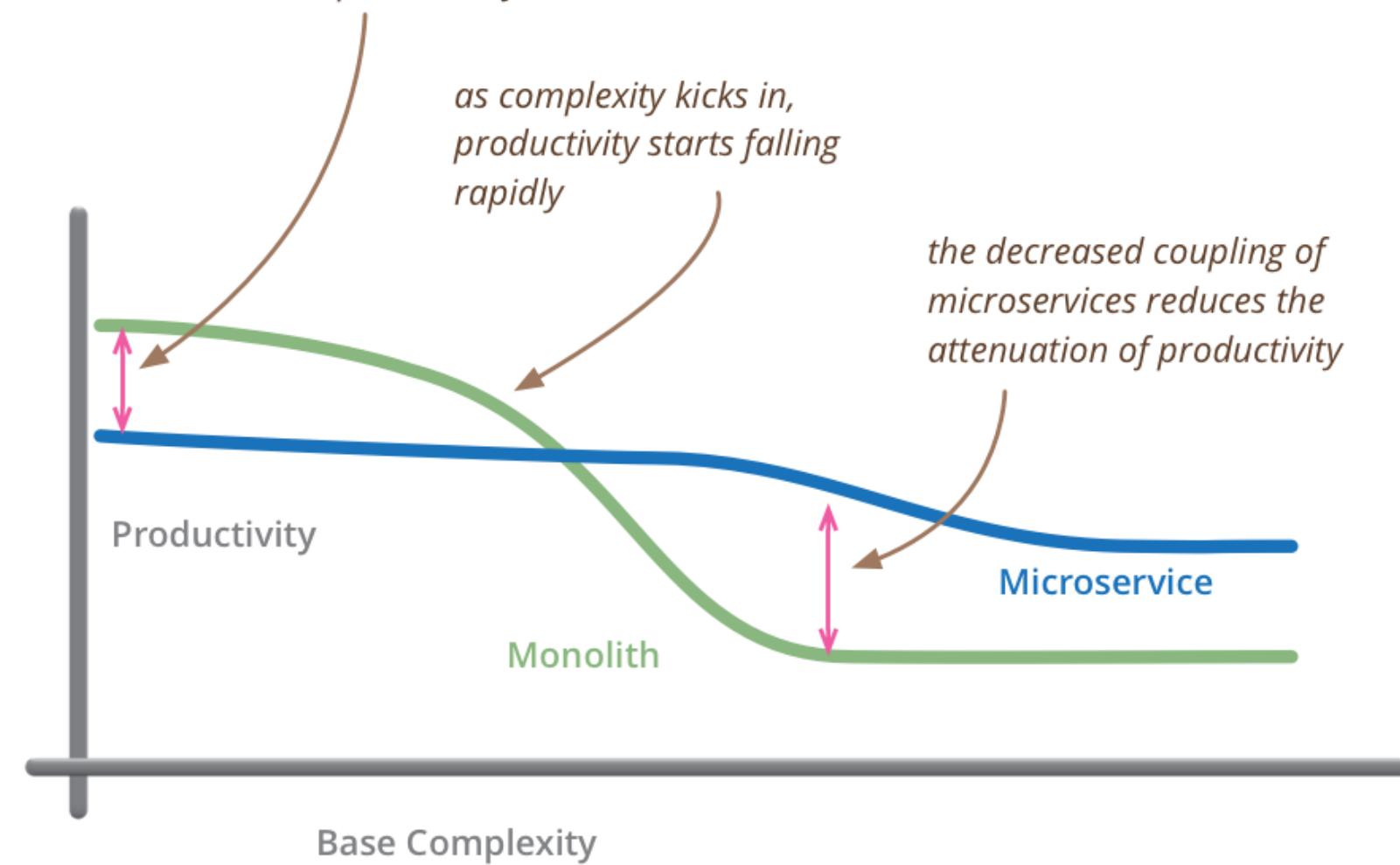
- Si falla un microservicio, no todo el sistema falla
- Su infraestructura es fácilmente escalable
- Fácil de escalar los equipos de desarrollo (cada equipo sólo debe mantener un par de microservicios)
- Código más mantenable y reutilizable (menos interdependencias)
- Permiten utilizar la mejor tecnología para cada problema
- Facilitan externalización a equipos de desarrollo de otras empresas







*for less-complex systems, the extra baggage required to manage microservices reduces productivity*



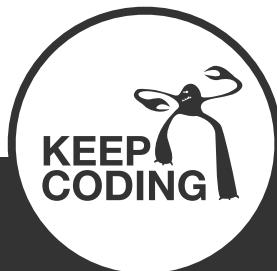
*but remember the skill of the team will outweigh any monolith/microservice choice*

<http://martinfowler.com/microservices/>

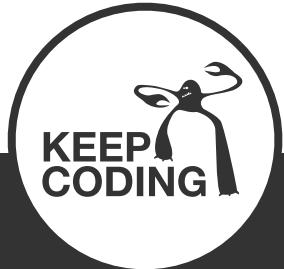
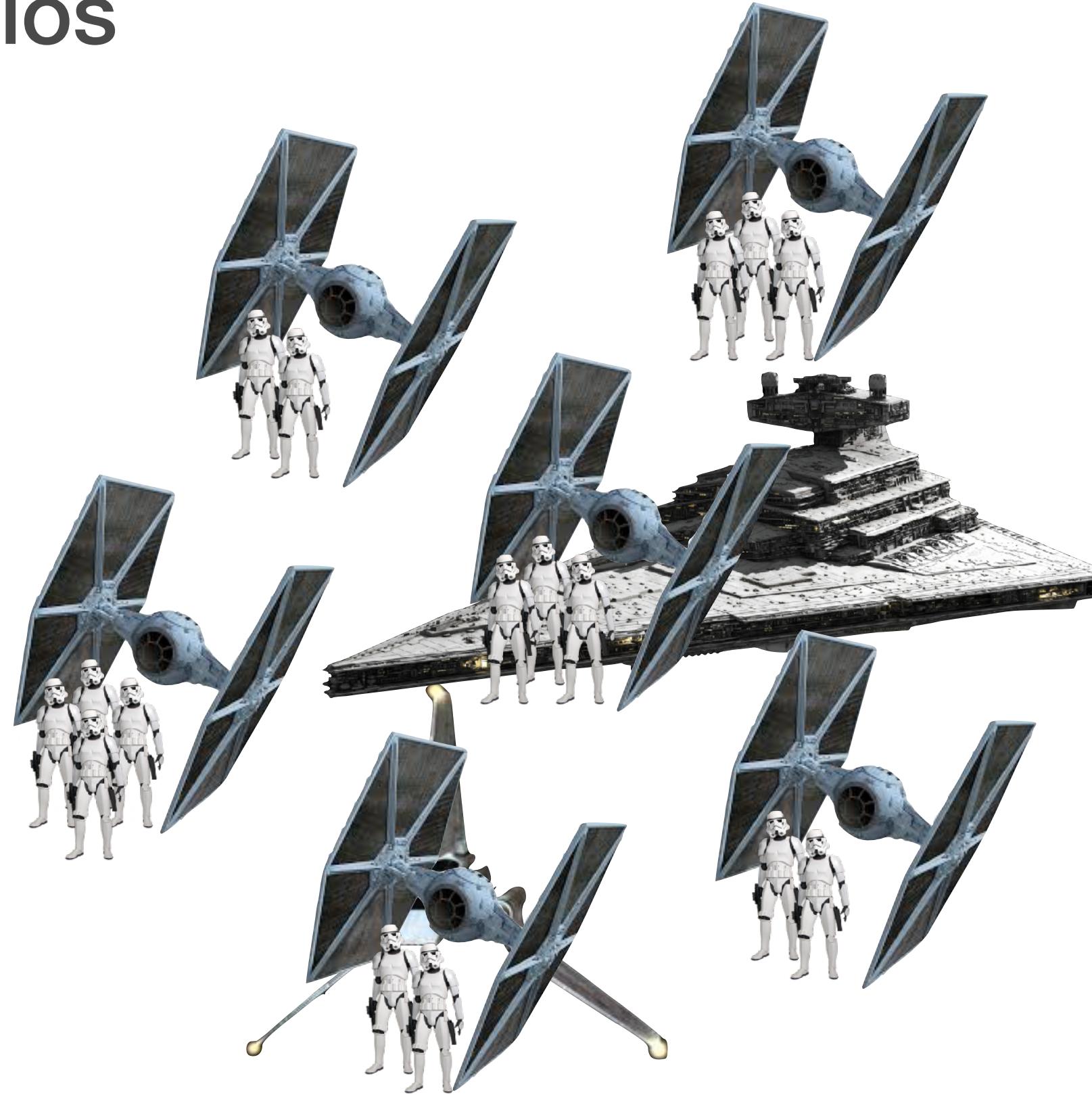
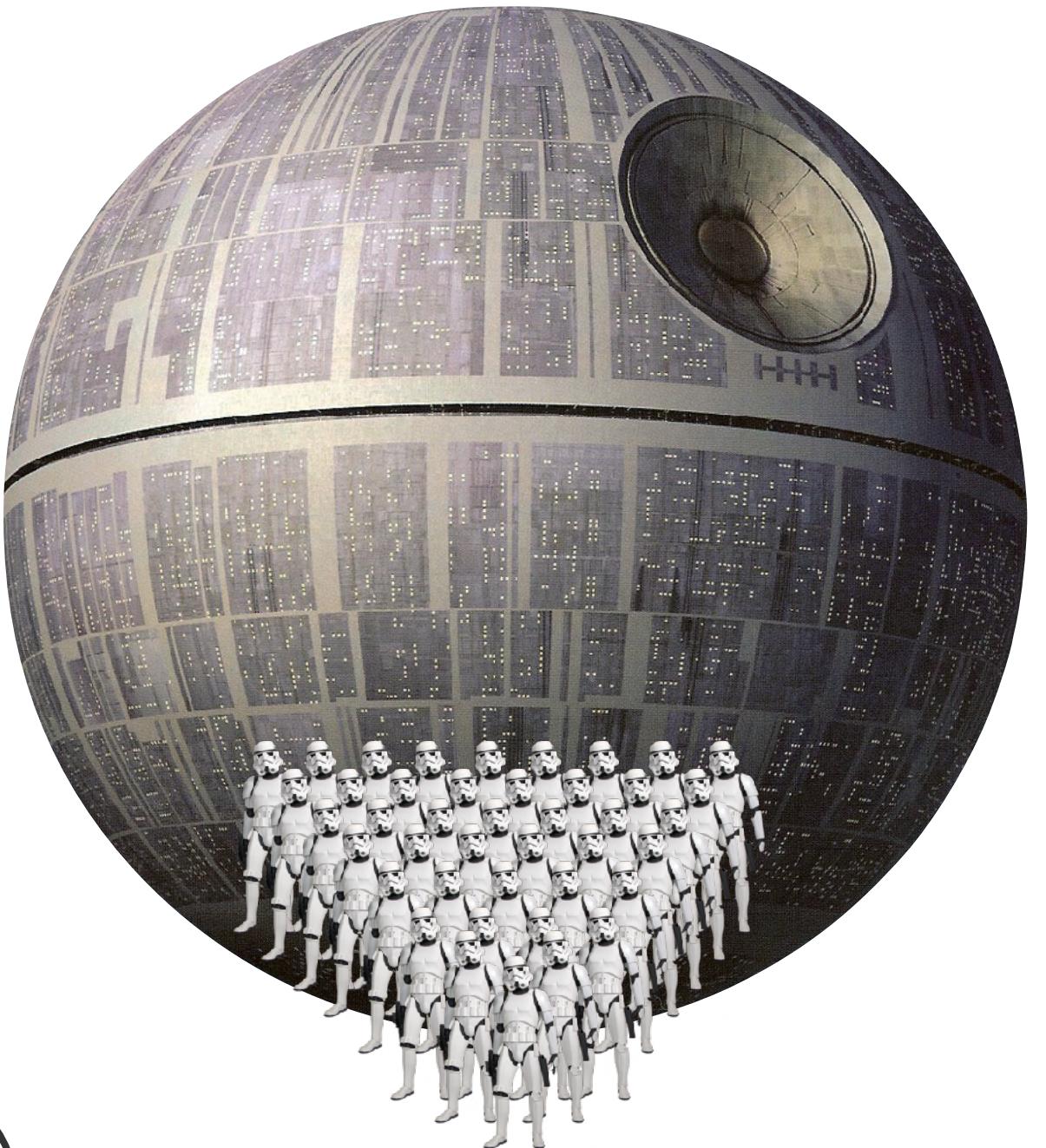


# ■ Problemática de los microservicios

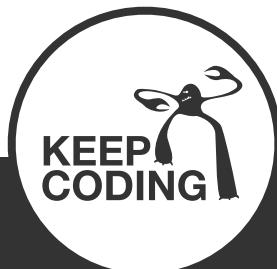
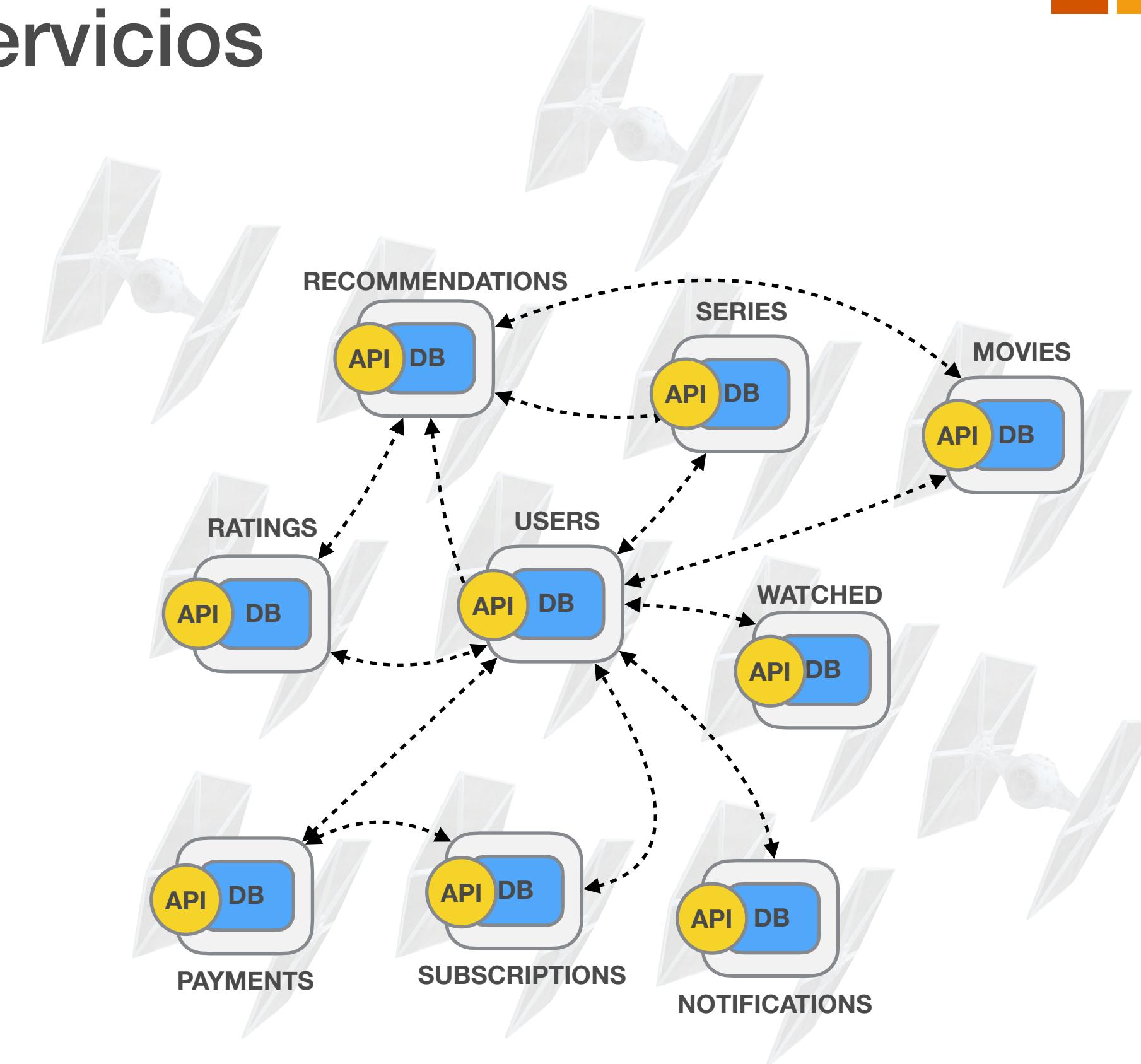
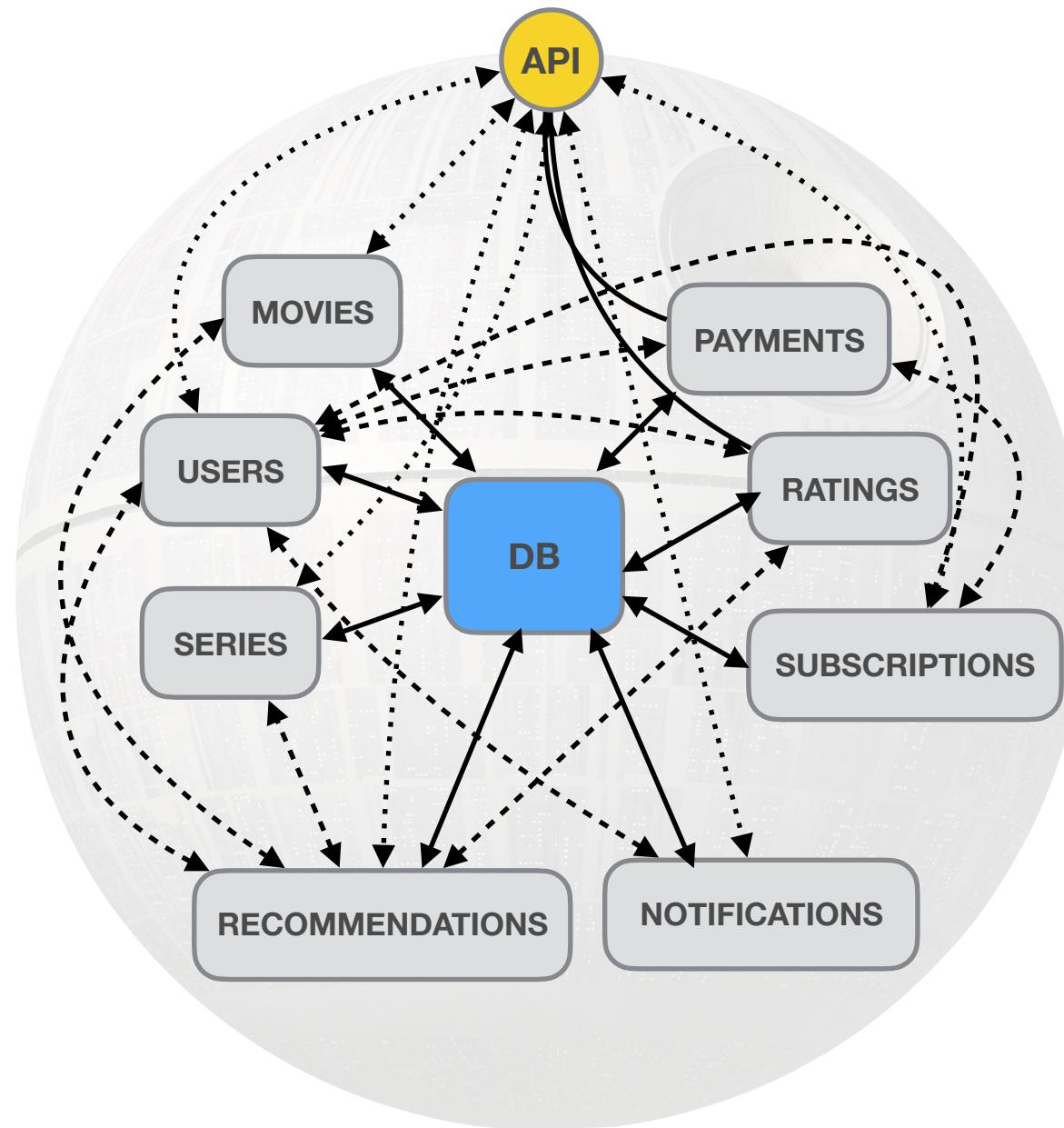
- Orquestación: es más difícil hacer una buena coordinación de todos
- Entorno de desarrollo: arrancar un entorno de desarrollo con 300 microservicios te puede llevar un rato... solución: Docker/Vagrant
- Negociación y compromiso: cada micro servicio debe mantener sus contratos con otros micro servicios que los usan (no podemos cambiar cosas a la ligera).
- Gestión de logs. Están distribuidos, hay que unificarlos para facilitar la búsqueda
- Unión de datos: los JOIN debemos hacerlos a mano
- ¿Quién es el responsable de controlar la autenticación y autorización?



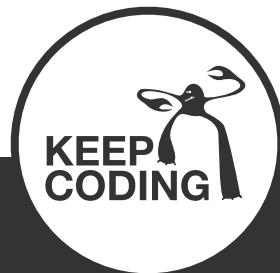
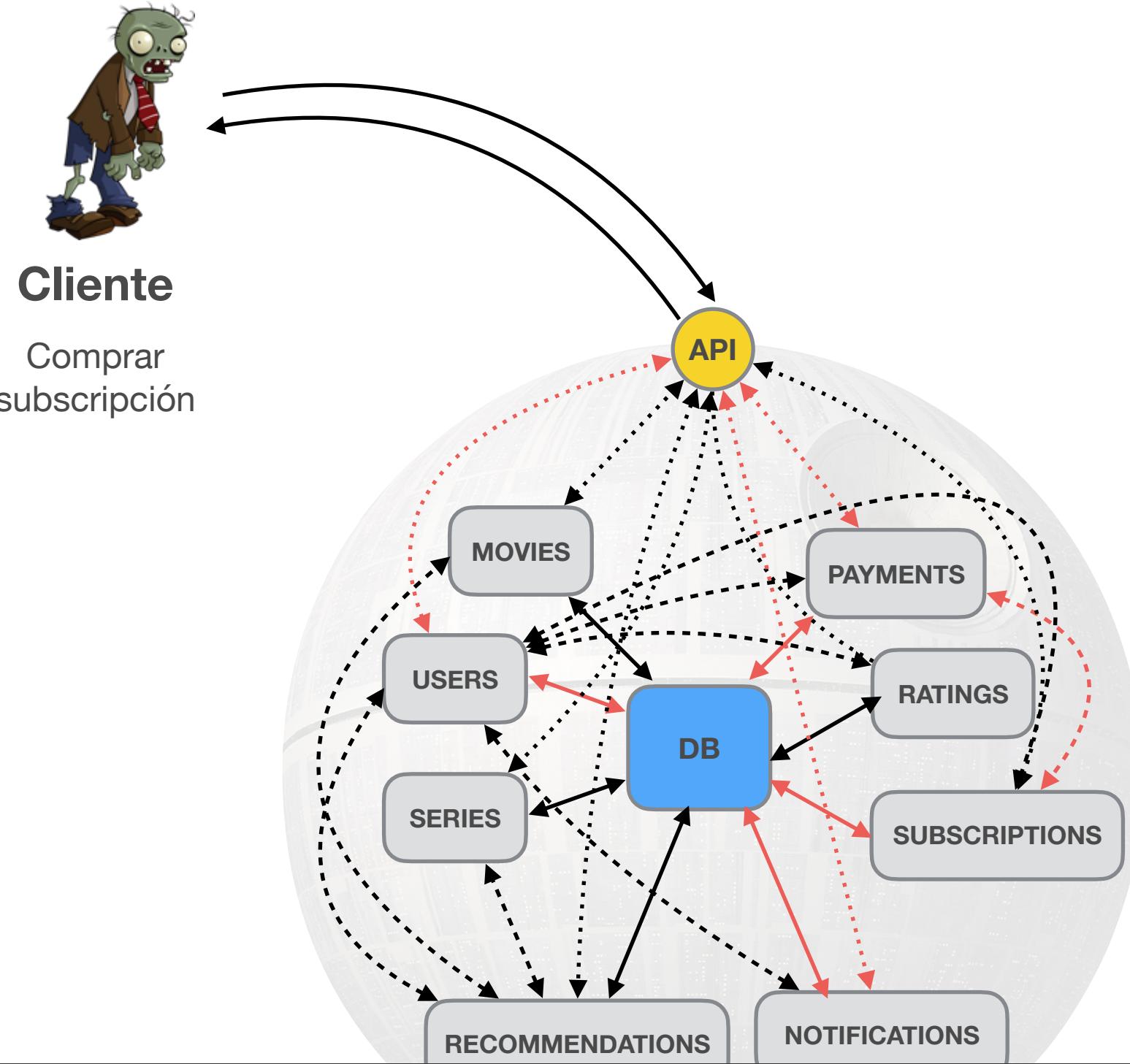
# ■ Del monolito a microservicios



# ■ Del monolito a microservicios

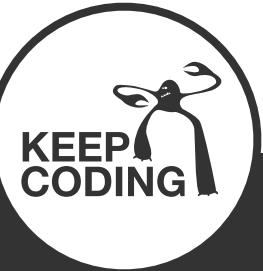
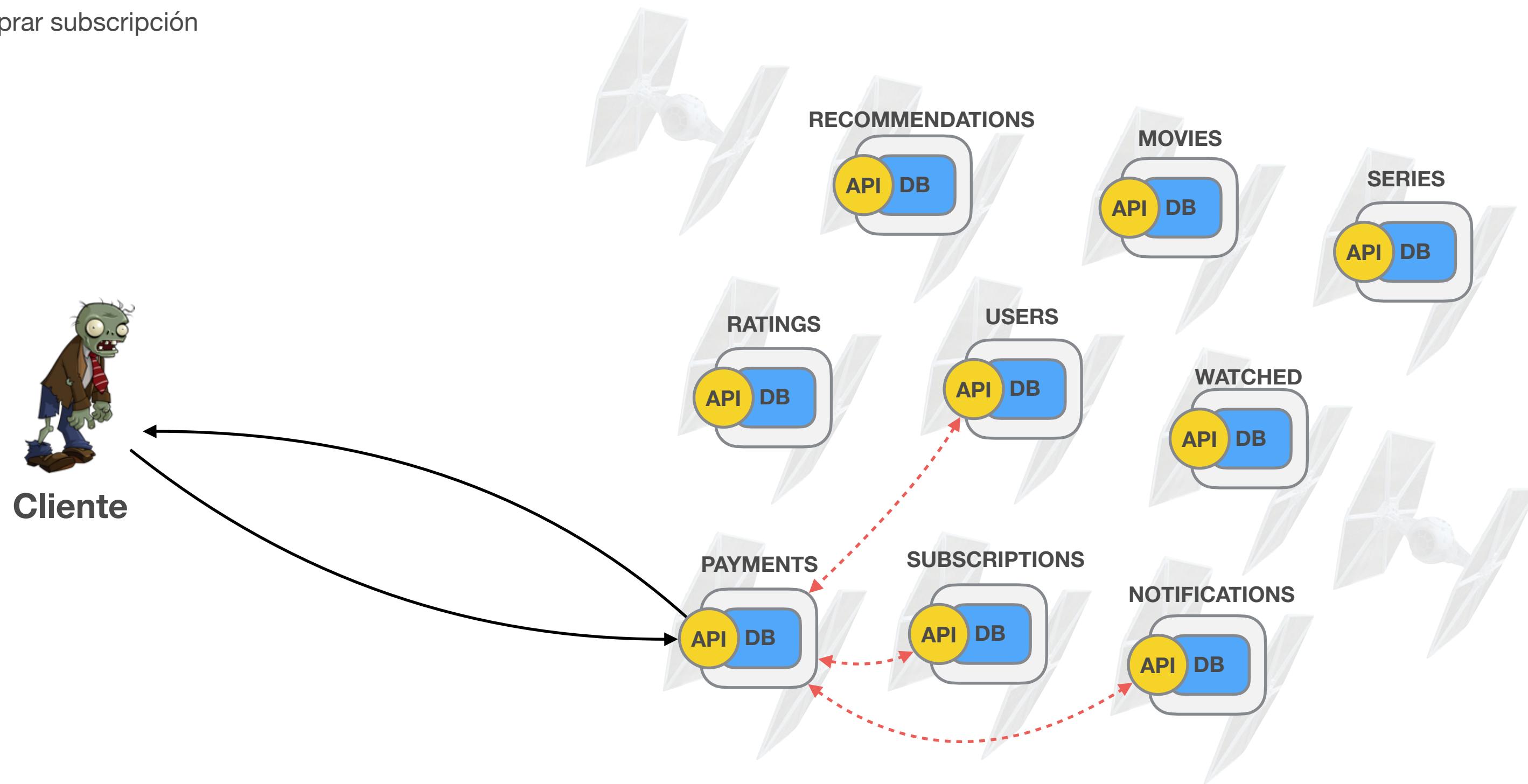


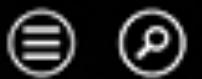
# ■ Cliente-Servidor con monolito



# Conexión directa a microservicios

Acción: Comprar subscripción



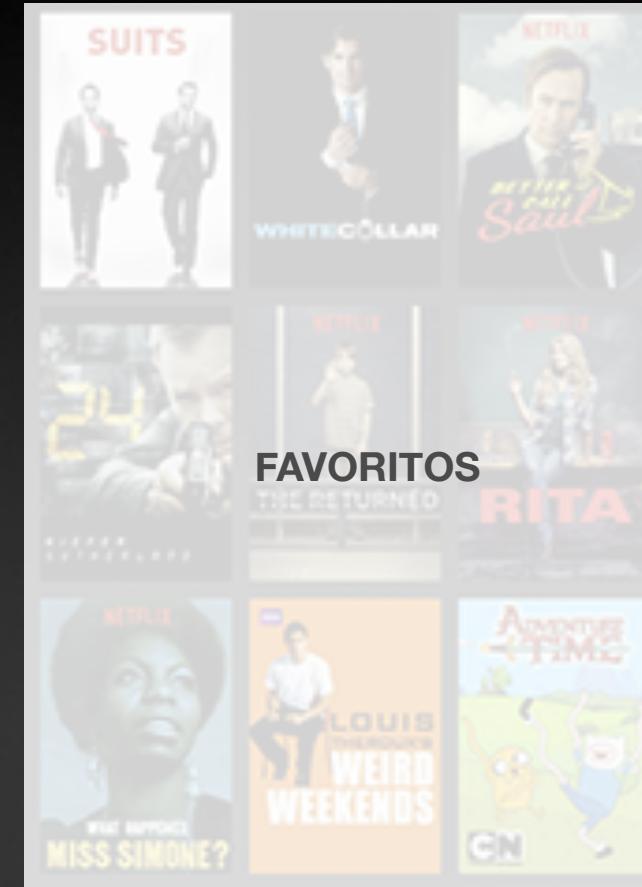


Helen

Continue Watching



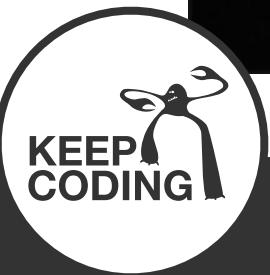
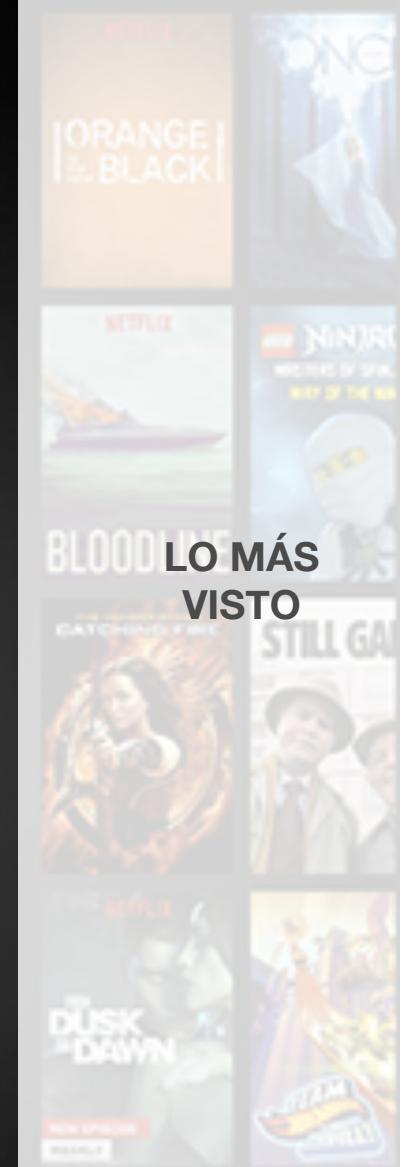
My List



Top Picks for Helen

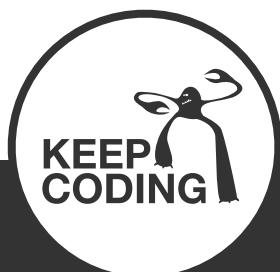
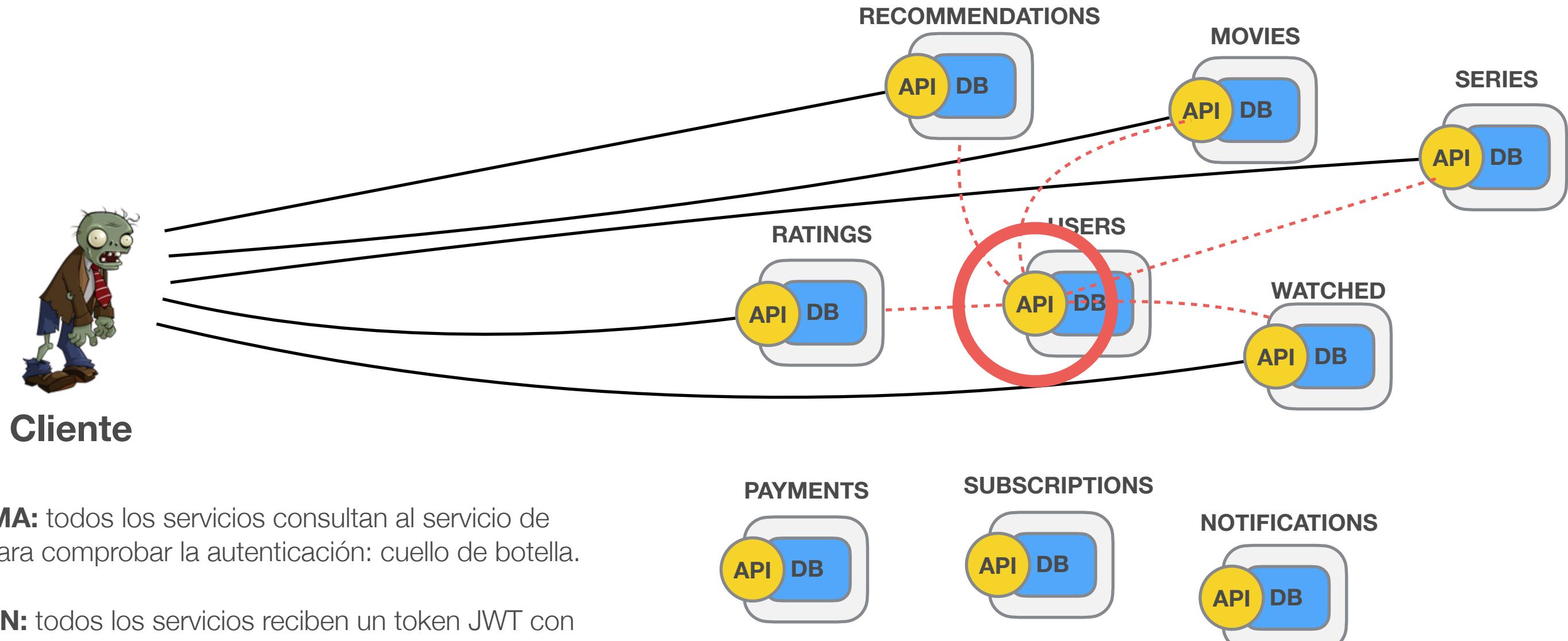


Popular on Netflix



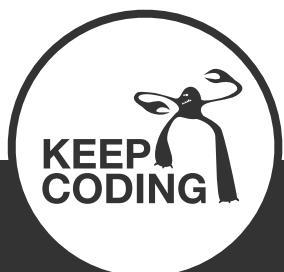
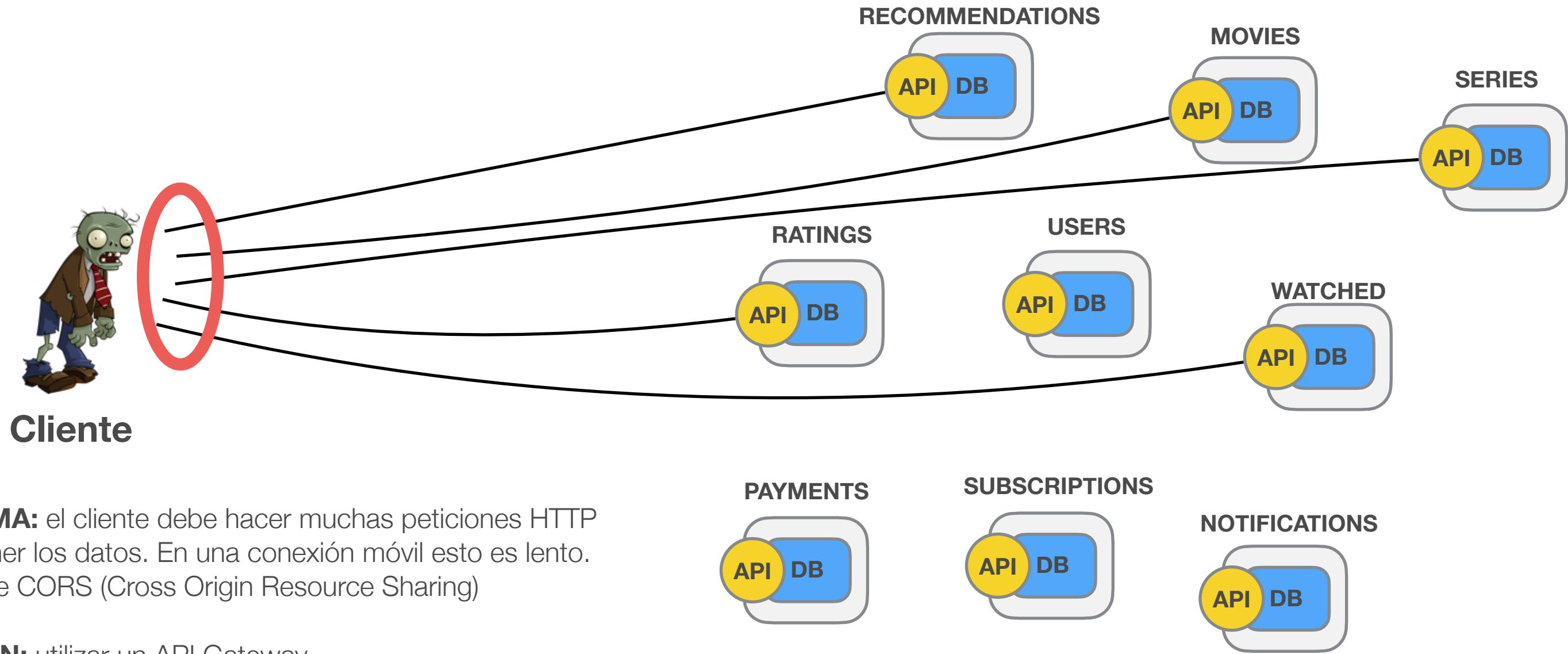
# Conexión directa a microservicios

Acción: Recuperar datos para la página principal



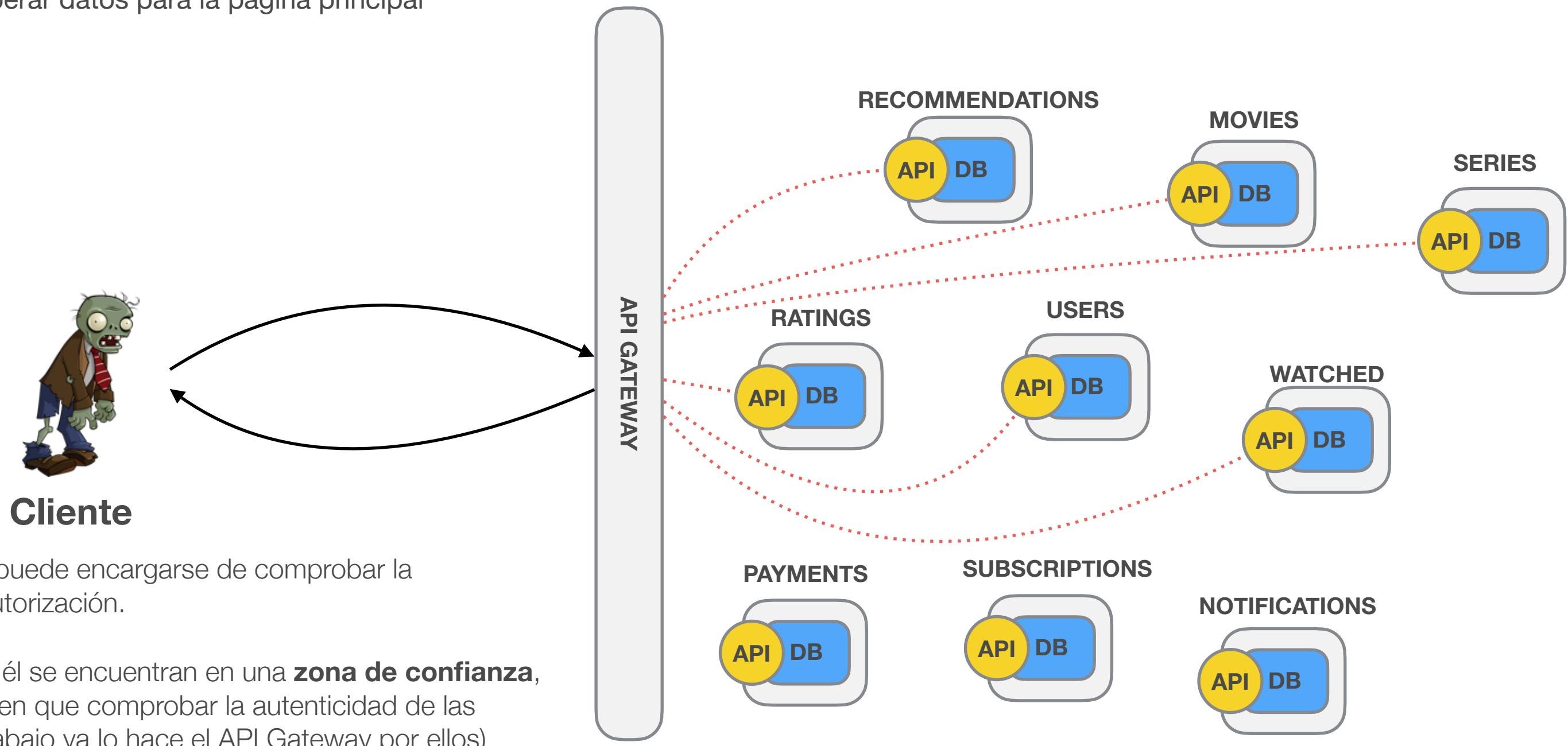
# Conexión directa a microservicios con JWT

Acción: Recuperar datos para la página principal



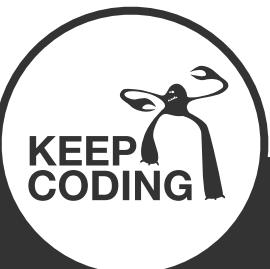
# Conexión a API Gateway

Acción: Recuperar datos para la página principal



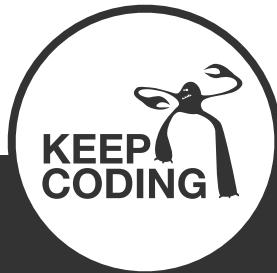
El **API Gateway** puede encargarse de comprobar la autenticación y autorización.

Los servicios tras él se encuentran en una **zona de confianza**, por lo que no tienen que comprobar la autenticidad de las peticiones (ese trabajo ya lo hace el API Gateway por ellos)



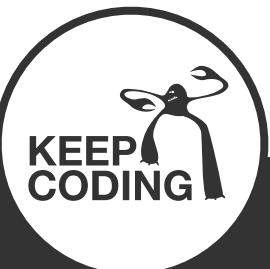
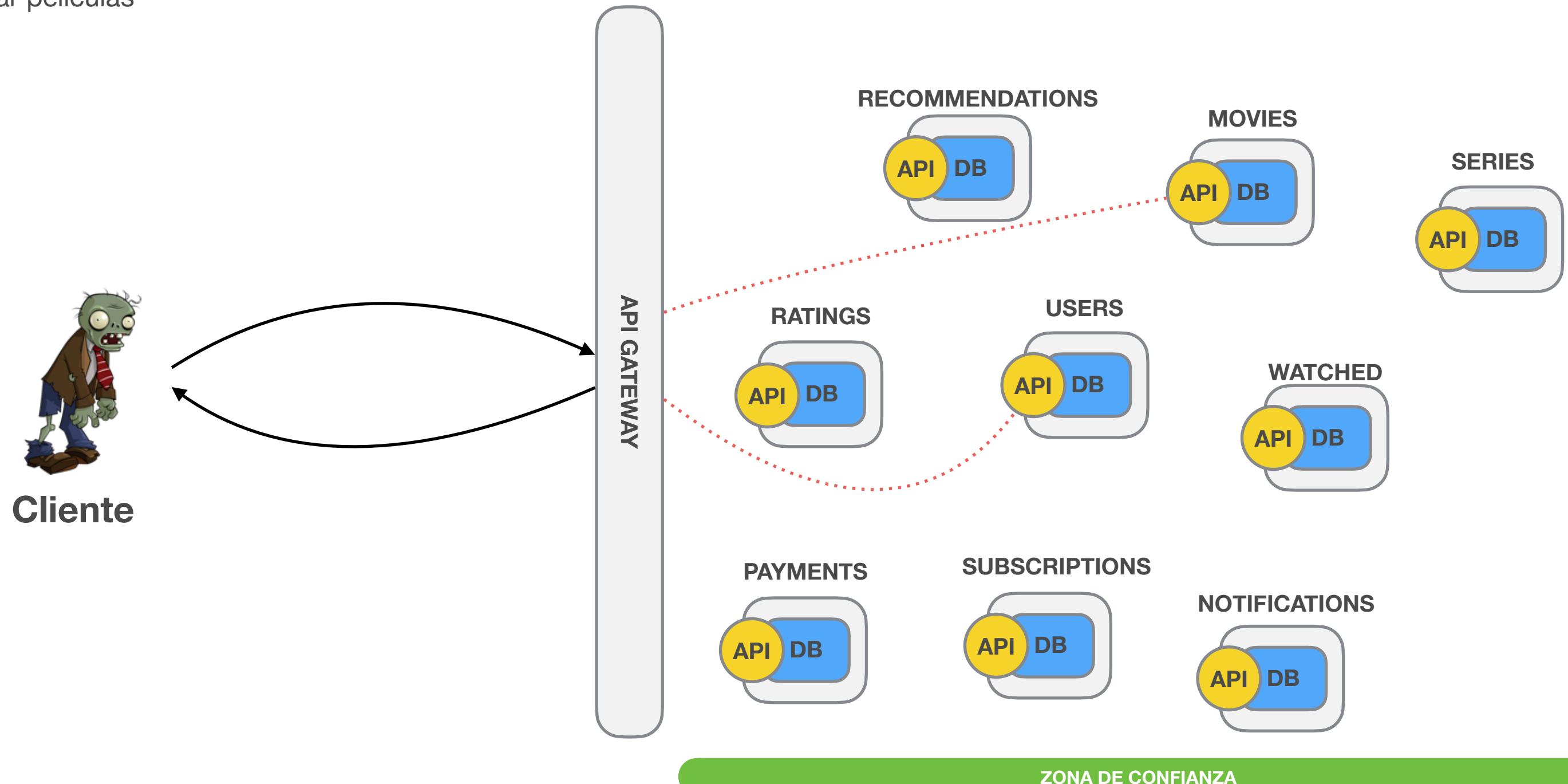
# ■ API Gateway

- Permite ser el único punto de entrada al sistema
- Comprueba la autenticación y autorización de los usuarios
- Los servicios tras él se desprecocupan de la autenticación y autorización
- Puede actuar como proxy/router: enrutando la petición al servicio y devolviendo la misma respuesta que el servicio
- O puede actuar como gestor: haciendo varias peticiones y mezclando el resultado



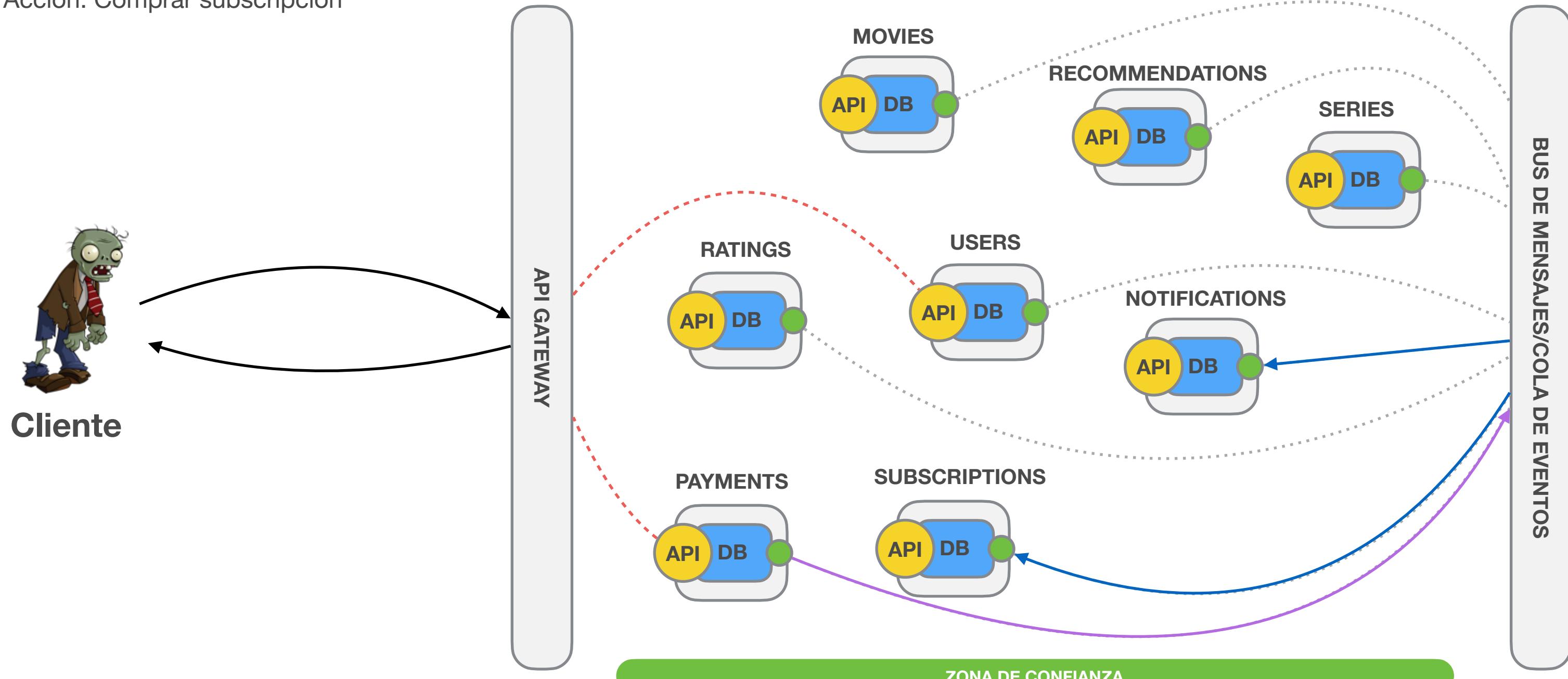
# ■ API Gateway como router/proxy

Acción: buscar películas

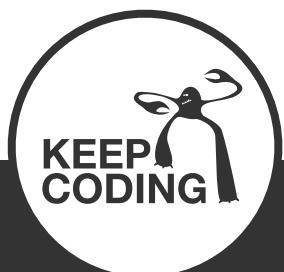


# ■ Arquitectura basada en eventos

Acción: Comprar subscripción

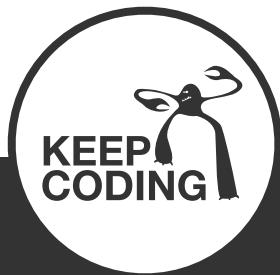


Gracias a la arquitectura basada en eventos podemos coordinar eventos sin tener que conectarlos síncronamente

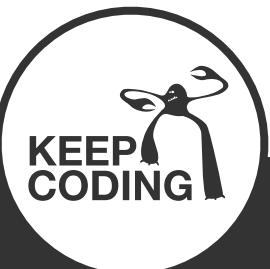
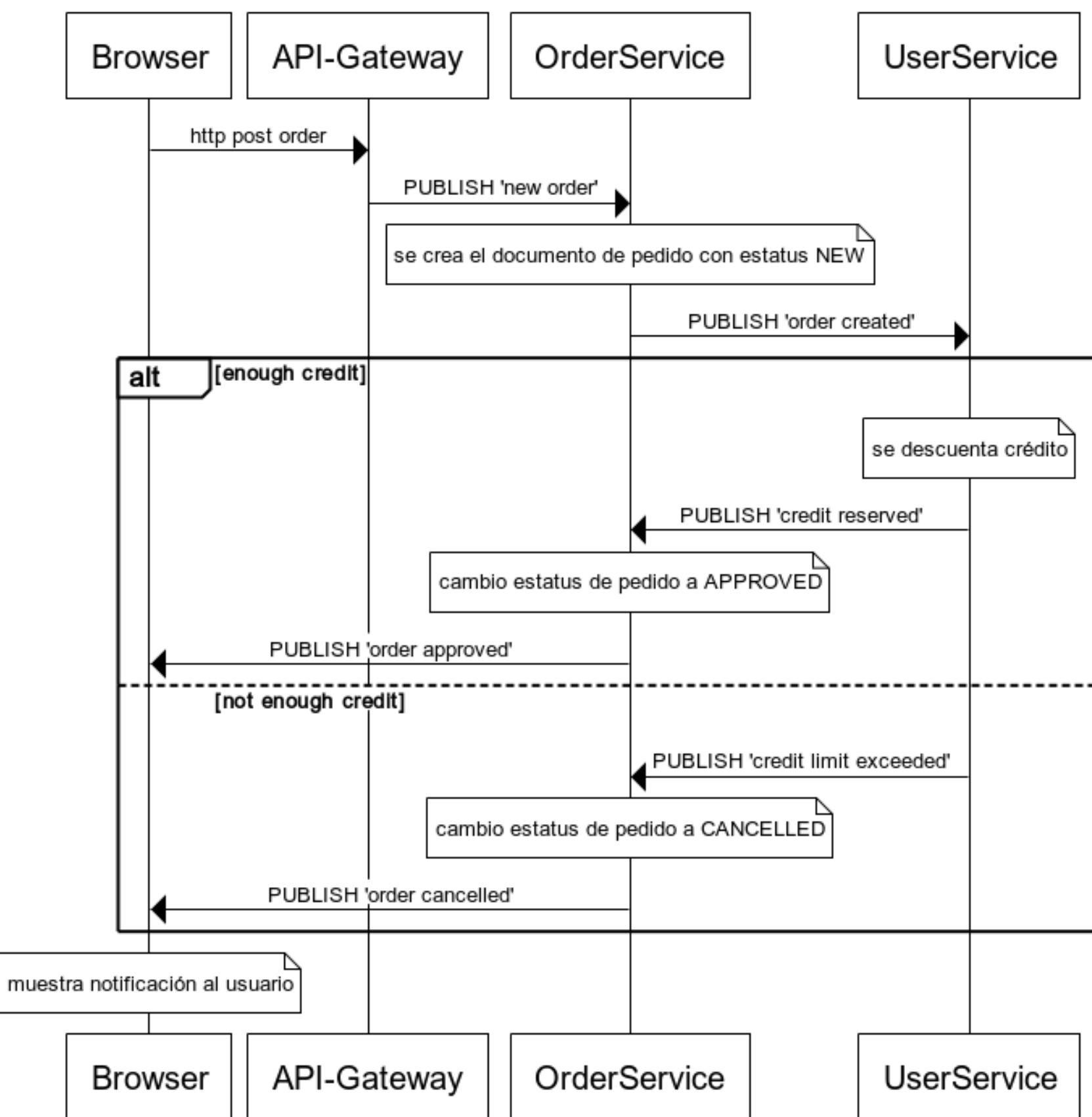


# ■ Características de un microservicio

- Resolver un sólo problema y hacerlo muy bien
- Utilizar la mejor tecnología posible para resolver ese problema
- Exponer un API con endpoints para interactuar con él
- Base de datos propia (a ser posible, se aceptan BD compartidas)
- Conexión al bus de mensajes/cola de eventos (si es necesario)
- Librería cliente del API (a ser posible)

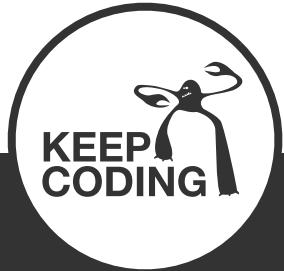


KEEP  
CODING



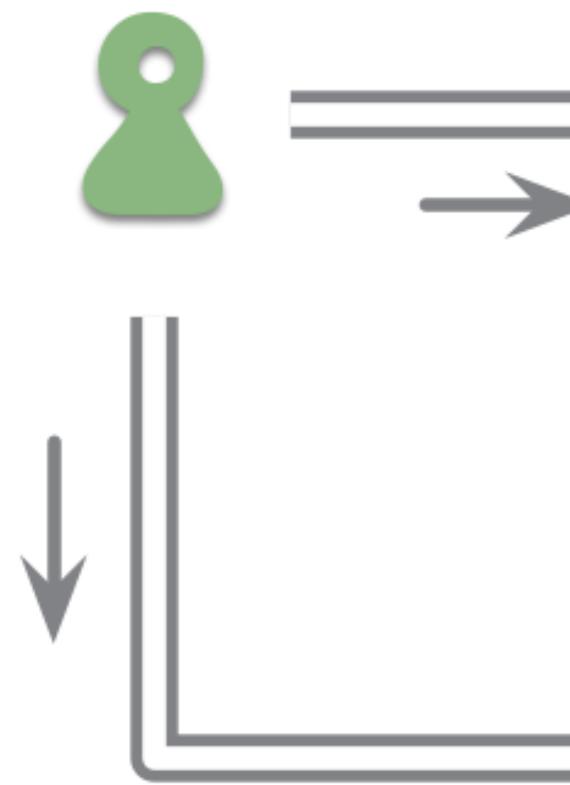


# Monolith First

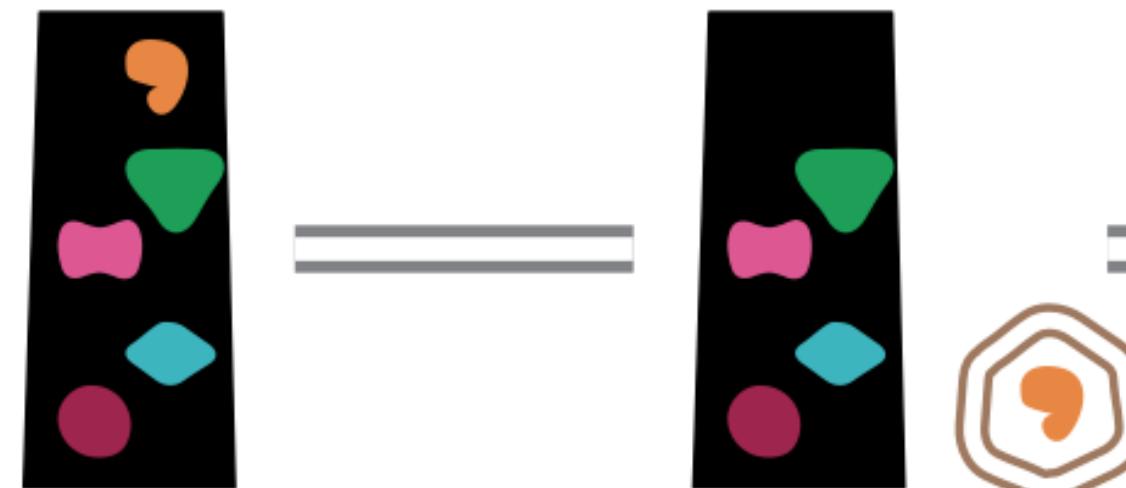




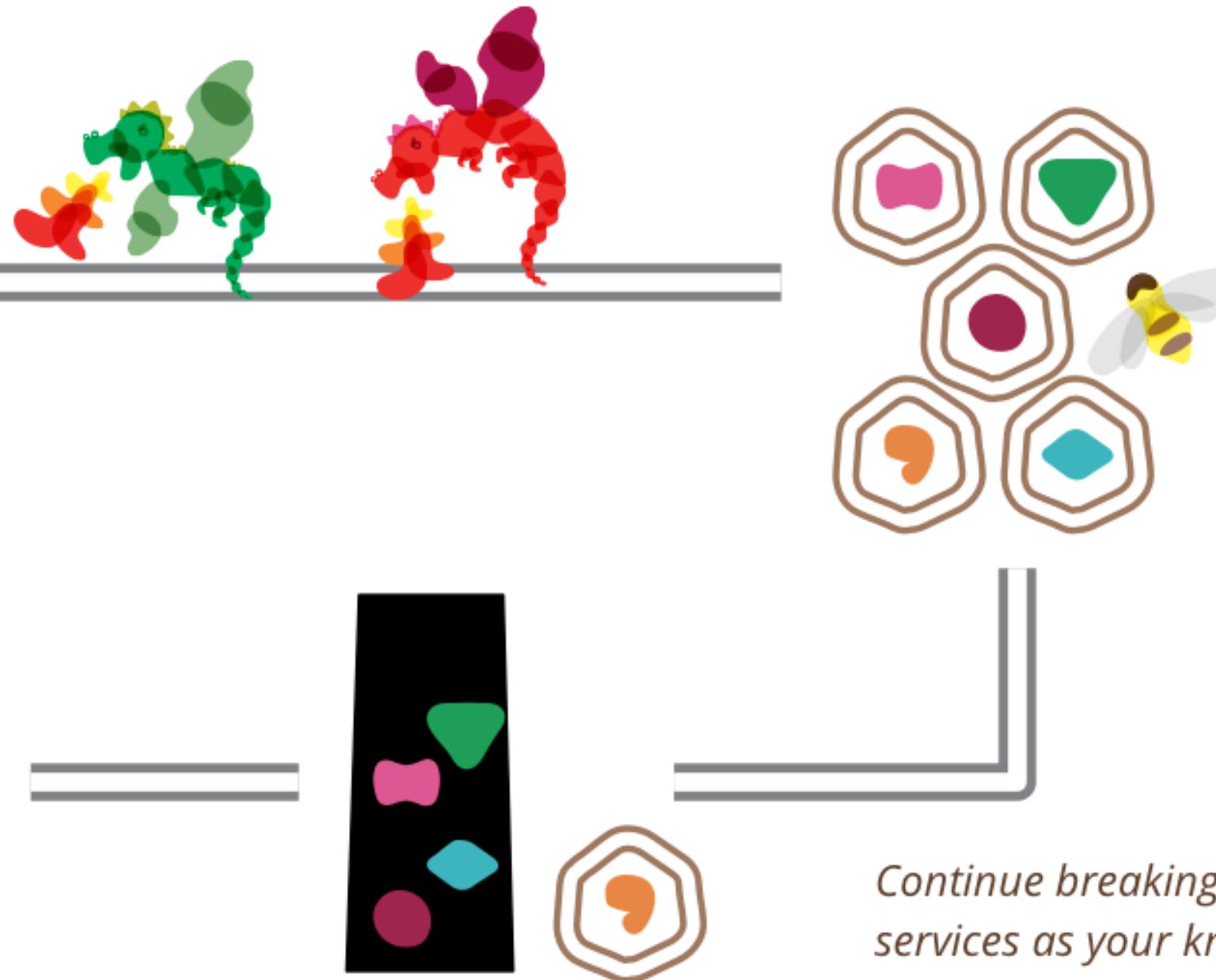
*Going directly to a microservices architecture is risky*



*A monolith allows you to explore both the complexity of a system and its component boundaries*

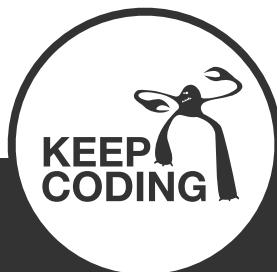


*As complexity rises start breaking out some microservices*



*Continue breaking out services as your knowledge of boundaries and service management increases*

<https://martinfowler.com/bliki/MonolithFirst.html>





<https://pm2.keymetrics.io/docs/usage/quick-start/>

PM2 is a daemon process manager that will help you manage  
and keep your application online 24/7

# ■ PM2

pm2 start ./bin/www

# para configurar en boot

pm2 startup

pm2 stop [all]

pm2 delete [all]

pm2 reload all

pm2 ecosystem

pm2 logs

pm2 logs app

pm2 logs app --lines 100

pm2 logs --json

pm2 logs --format

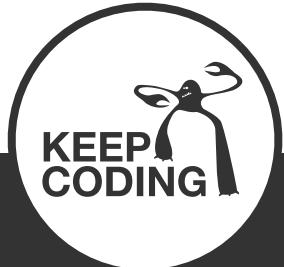
pm2 list --watch

pm2 monit

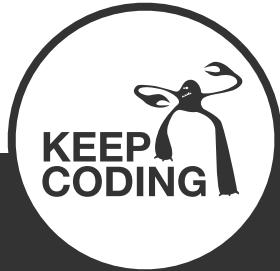
pm2 start ecosystem.pm2.config.js

pm2 stop ecosystem.pm2.config.js

pm2 delete ecosystem.pm2.config.js



# HTTPS en local



# Como poner HTTPS en mi entorno local (en 5 minutos o menos)

manual

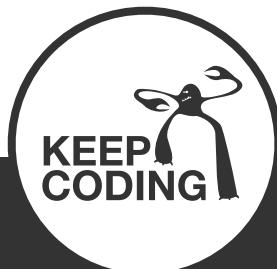
<https://www.freecodecamp.org/news/how-to-get-https-...>

shell

<https://github.com/dakshshah96/local-cert-generator...>

mkcert

<https://github.com/FiloSottile/mkcert>



# También podemos usar:

<https://ngrok.com/>

```
# https://dashboard.ngrok.com/ Authentication / Your Authtoken  
ngrok authtoken <token>
```

```
ngrok http 3000
```

```
ngrok http -auth="username:password" 3000
```

<http://localhost:4040/status>

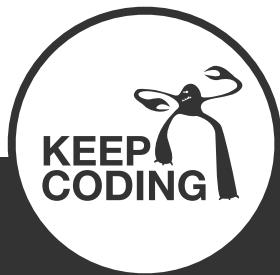


# Mi preferido:

<https://github.com/localtunnel/localtunnel>

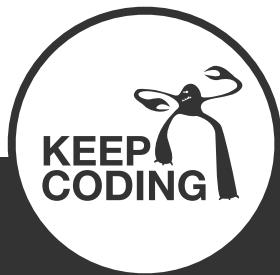
```
npx localtunnel -p 3000
```

```
npx localtunnel -p 3000 --subdomain jamg44
```



# Comparados

- ngrok: http/https/tcp tunnels
- Localtunnel: solo http/https, pero tiene custom domains



# Otra forma más!

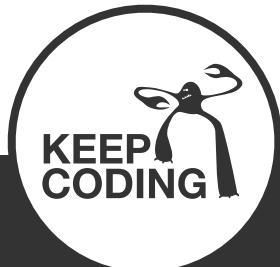
```
nodeapp.localhost {  
    reverse_proxy localhost:3000  
}
```

`<- Caddyfile`

```
"scripts": {  
    "dev": "cross-env DEBUG=app:* nodemon",  
    "dev:https": "concurrently 'npm:dev' 'npm:caddy'",  
    "caddy": "caddy run",  
    "start": "node ./bin/www",  
    ...  
},
```

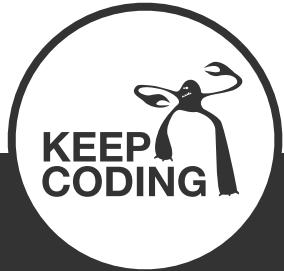
`<- package.json`

También: `caddy reverse-proxy --from nodeapp.localhost --to localhost:3000`





# Cluster

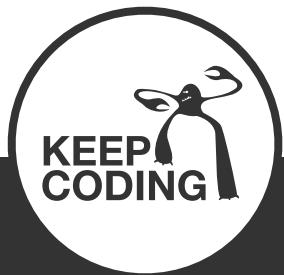


# ■ Cluster

Desde hace algún tiempo cluster es un módulo core de Node.js.

Podemos requerirlo y pedirle que cree nuevos procesos (fork) compartiendo el mismo puerto de escucha.

Esto debemos hacerlo solo al inicio cuando sabemos que somos el master (`cluster.isMaster`).



# Cluster

```
var cluster = require('cluster');

if (cluster.isMaster) {

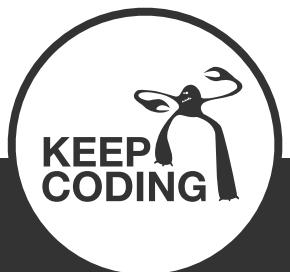
    // hacer forks creando workers, uno por core lógico
    var numCPUs = require('os').cpus().length;
    for (var i = 0; i < numCPUs; i++) {
        cluster.fork();
    }

} else {

    // arranque normal de nuestro server

}
```

ejemplos/cluster



# ■ Cluster

Esto arrancará n procesos adicionales, a los que el master les irá dando peticiones según lógica round robin.

Cluster emite eventos a los que podemos suscribirnos, como por ejemplo:

- online
- exit
- listening
- disconnect
- message

ejemplos/cluster



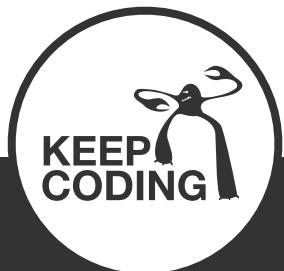
# Cluster

Los eventos que emite los escucharemos así:

```
// al arrancar un worker...
cluster.on('online', function(worker) {
  console.log('Worker ' + worker.id +
    ' is online with pid ' + worker.process.pid);
});

// al terminar un worker...
cluster.on('exit', function(worker, code, signal) {
  console.log('worker ' + worker.process.pid + ' died');
});
```

ejemplos/cluster

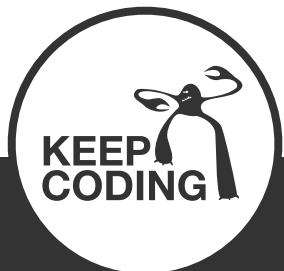


# Cluster

Cuando un worker muere emite el evento exit. En este momento podemos reaccionar y crear otro que le sustituya:

```
// al terminar un worker...
cluster.on('exit', function(worker, code, signal) {
  console.log('Worker ' + worker.process.pid +
    ' died with code: ' + code +
    ', and signal: ' + signal);
  console.log('Starting a new worker');
  cluster.fork();
});
```

ejemplos/cluster



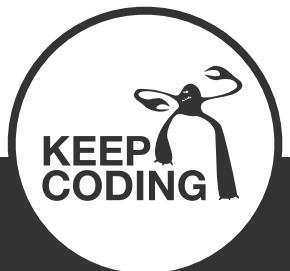
# Cluster

El proceso master puede mandar mensajes a los workers.

```
// master
// cluster.workers nos da una lista de los que tenemos
worker.send('hello from the master');

// worker
process.on('message', function(message) {
  console.log(message);
});
```

ejemplos/cluster



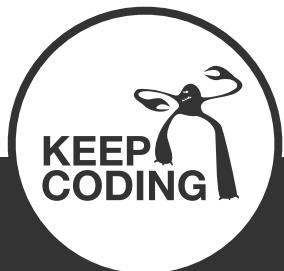
# Cluster

Los workers pueden mandar mensajes al master.

```
// worker
process.send('hello from worker with id: ' + process.pid);
```

```
// master
worker.on('message', function(message) {
  console.log(message);
});
```

ejemplos/cluster

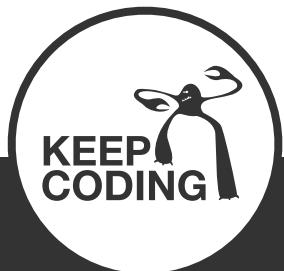


# ■ Cluster

Con cluster aumentaremos notablemente el rendimiento de nuestro servidor, usando más recursos del sistema.

Concurrent Connections	1	2	4	8	16
Single Process	654	711	783	776	754
8 Workers	594	1198	2110	3010	3024

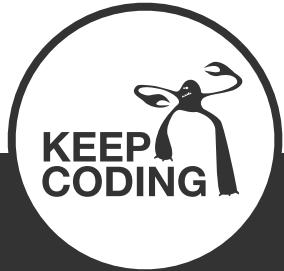
\* Ejemplo de medición con peticiones por segundo.



# ■ Cluster

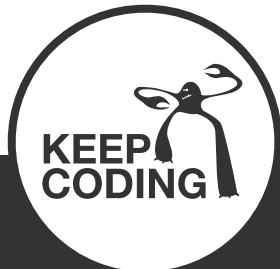
Es recomendable leer la documentación!

<https://nodejs.org/api/cluster.html>



# Inspiración

- [node-api-boilerplate](#) un buen ejemplo de DDD
- [dev-mastery comments-api](#) gran traducción de clean architecture a Node.js, viene con un [video de explicación](#)
- [TypeScript-starter](#) ejemplo con TypeScript y librerías modernas
- [nodejs-api-starter](#) starter con GraphQL
- [bulletproof-nodejs](#) conocido starter con estructura en 3 capas
- [Hackathon-starter](#) aplicación para arranque con autenticación y mucho más



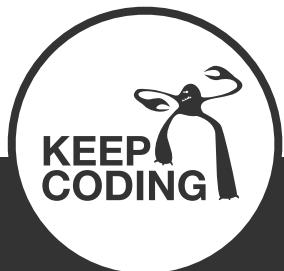
# ■ Arquitectura

<https://alexkondov.com/tao-of-node/>

Tiempo de lectura ~2h

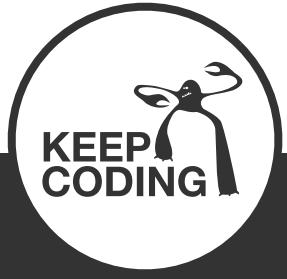
<https://www.freecodecamp.org/news/nodejs-api-best-practices-for-scaling>

Tiempo de lectura ~15m



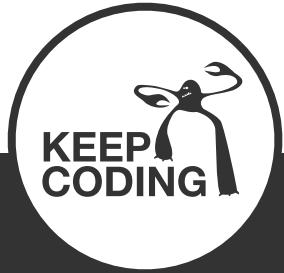


# Consejos





# Node.js

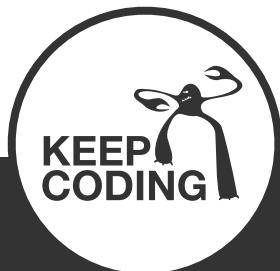


# Buenas prácticas - Node.js

Formato de los callbacks con el error siempre como primer parámetro.

```
var resizeImage(imagePath, callBack) {  
  
    // hacer el procesado que queramos  
    var resizedImagePath = resizer(imagePath, ...);  
  
    // devolvemos en primer lugar error y luego los resultados  
    return callBack(null, resizedImagePath);  
}
```

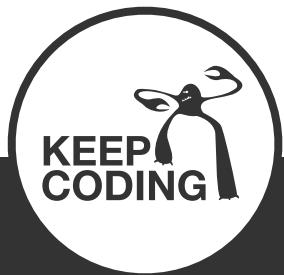
Y mejor usar promesas!



# Buenas prácticas - Node.js

Al recibir un callback, comprobar primero si ha habido error para devolverlo y luego hacer el proceso previsto.

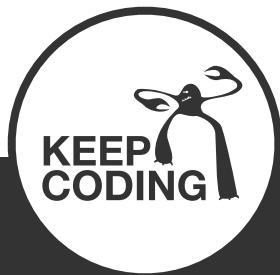
```
function getConfig(callBack) {  
  
    leeFichero('./config.json', function(err, fileContent) {  
        // primero comprobamos y retornamos posibles errores  
        if (err) {  
            return callBack(err);  
        }  
  
        return callBack(null, JSON.parse(fileContent));  
    });  
  
}
```



# Buenas prácticas - Node.js

Usar return al invocar callbacks, reduce la posibilidad de errores por olvidar hacer return y llamar varias veces al callback de la función.

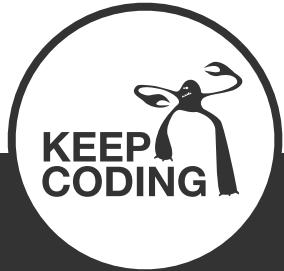
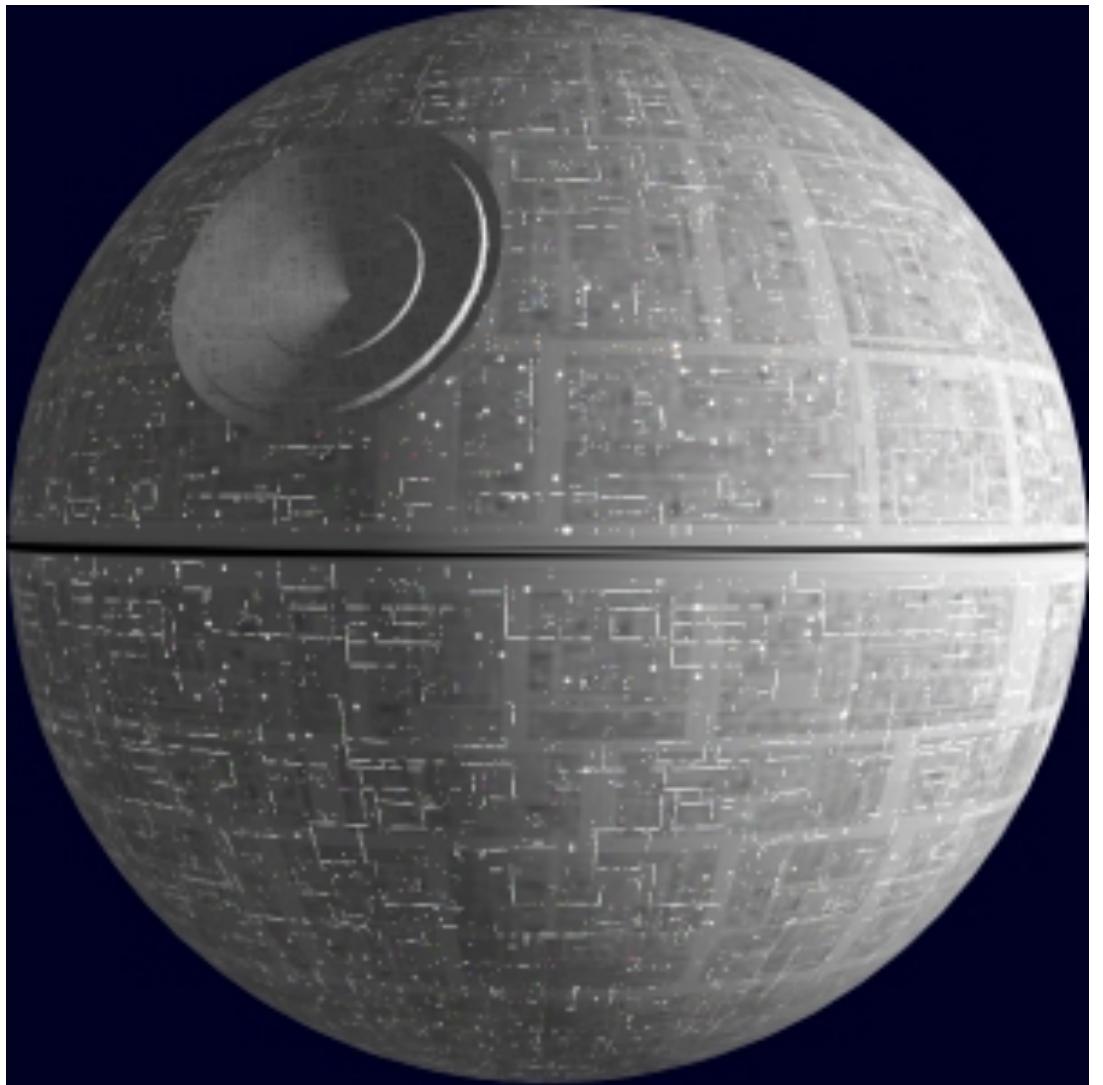
```
function saveUpload(res, callBack) {
  leeFichero('./config.json', function(err, fileContent) {
    if (err) {
      callBack(err);
      // aquí deberíamos parar! --> RETURN
      // para no olvidarlo y mejorar la legibilidad usar return callBack(err);
    }
    callBack(null, JSON.parse(fileContent));
    // aquí mejor return callBack(null, JSON...
  });
}
```



# Buenas prácticas - Node.js

Dividir la funcionalidad en varios módulos especializados, cada uno haciendo solo su trabajo y hacerlo bien.

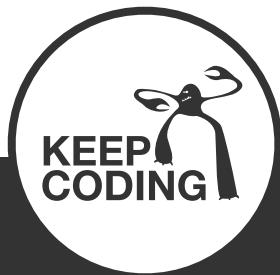
No construyamos estrellas de la muerte que hacen varias cosas, serán difíciles de mantener!



# Buenas prácticas - Node.js

Gestionar lo mejor posible los errores:

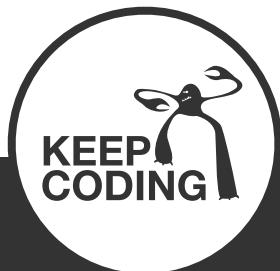
- Hay errores operacionales que se pueden reintentar (una petición a un API de terceros que dio timeout) y errores ante los cuales debemos parar la aplicación (no se pudo leer el fichero de configuración)
- Hagamos log de lo que ocurre en nuestra aplicación, con varios niveles (debug, info, warning, error)



# Buenas prácticas - Node.js

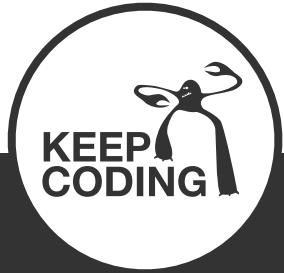
Y algunos consejos más...

- Antes de usar propiedades de un objeto pensar si ese objeto puede no existir
- Usar `npm init` al iniciar una aplicación o un módulo nuevo
- Usar patrones reconocibles al procesar de forma asíncrona (p.e. `async`, promesas, `async/await`)
- Antes de crear un módulo nuevo mirar si ya existe en [npm](#)
- Usar ayudantes de estilo como [JSCS](#), [JSHint](#), etc
- Documentar las partes de nuestra aplicación con `jsdoc` o similar
- No poner datos sensibles en el código, usemos ficheros de configuración o variables de entorno





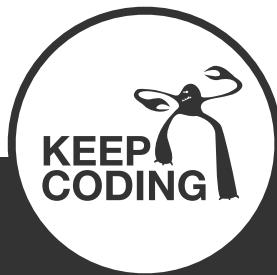
# APIs



# Buenas prácticas en APIs - versionado

Pongamos versión a nuestro API desde el principio.

Cuando tengamos clientes consumiéndola y tengamos que cambiarla tendremos que hacer una nueva versión.

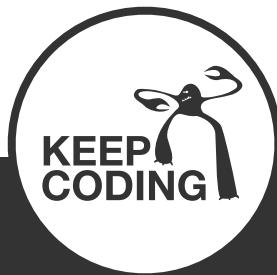


# Buenas prácticas en APIs - versionado

Podemos versionar de múltiples formas, las dos más usadas son:

1. Versión en la ruta base /apiV1/users o api/V1/users
2. Versión en la cabecera. El cliente pone la versión que quiere consumir en el header.

La **opción 1** es más visible y por tanto más segura.

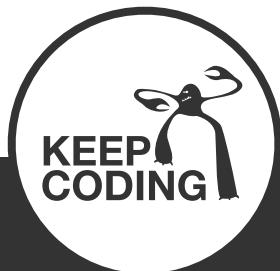


# Buenas prácticas en APIs - nombres

Hacer un API rest implica exponer recursos que son invocados con GET, POST, PUT, etc.

Los recursos deberían ser nombres, no verbos.

GET /users	(mejor que GET /getUserList)
POST /users	(mejor que POST /insertUser)



# Buenas prácticas en APIs - nombres

GET /users - Devuelve lista de usuarios

GET /users/12 - Devuelve un usuario concreto

POST /users - Crea un usuario

PUT /users/12 - Actualiza usuario #12

PATCH /users/12 - Actualiza parcialmente el usuario #12

DELETE /users/12 - Elimina usuario #12

**Solo tenemos que acordarnos del nombre del recurso (users) y la lista de métodos ya nos la sabemos!**



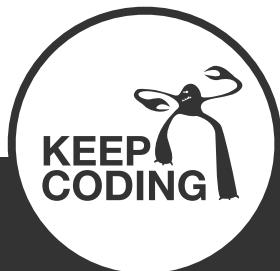
# Buenas prácticas en APIs - nombres

Singular o plural?

- GET /user
- GET /users

Decisión nuestra, pero usémoslo igual para todos los recursos.

El plural es comúnmente más usado.



# Buenas prácticas en APIs - nombres

¿Como hacemos con las relaciones?

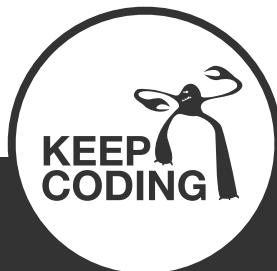
GET /users/12/emails - Devuelve lista de emails de usuario

GET /users/12/emails/7 - Devuelve un email concreto

POST /users/12/emails - Crea un email al usuario

PUT /users/12/emails/7 - Actualiza email #7 del usuario

...



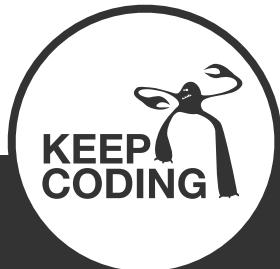
# Buenas prácticas en APIs - nombres

¿Como hacemos con las acciones distintas a GET, POST, etc?

Si la acción podemos usarla como un campo: PATCH /users/5/active

O podemos usarlo como un sub-recurso. Por ejemplo como hace github PUT /gists/:id/star pone una estrella y DELETE /gists/:id/star la quita.

A veces lo que queremos hacer no encaja con nada o implica múltiples recursos como por ejemplo una búsqueda sobre mensajes, usuarios y teléfonos, para estos casos /search tendrá mucho sentido!

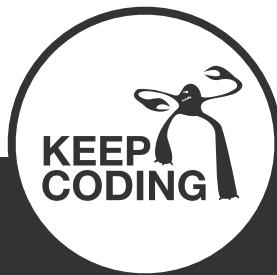


# Buenas prácticas en APIs - browser explorability

Facilitar las cosas al consumidor de nuestro API es bueno para que le guste y lo maneja bien.

Tratemos de que la lectura de todos los recursos sea posible desde un navegador.

Por ejemplo, usando el versionado en la ruta, haciendo un índice de los recursos con URLs clickables.



# Buenas prácticas en APIs - documentación

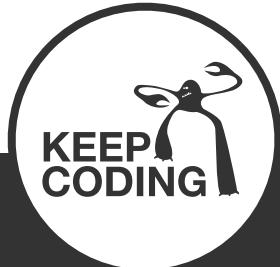
Podemos usar herramientas como apidoc, iodocs o swagger para ayudarnos.

Un par de ejemplos de con iodocs:

- <https://developers.egnyte.com/io-docs>
- <http://developer.klout.com/io-docs>

Actualmente el más usado es OpenAPI.

- <https://swagger.io/specification/>

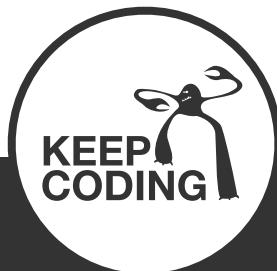


# Buenas prácticas en APIs - errores

Un buen API debe devolver errores en un formato estandarizado o previsible.

Los errores de cliente (tipo 400) deberían devolver un objeto JSON con propiedades similares en los distintos casos.

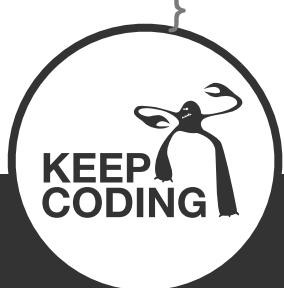
Los errores de servidor (tipo 500) no podrán hacerlo en todos los casos (un balanceador caído es difícil que devuelva JSON), pero al menos los que estén en nuestra mano deberían hacerlo.



# Buenas prácticas en APIs - errores

Para los errores de validación donde están involucrados multiples campos pueden devolver algo como:

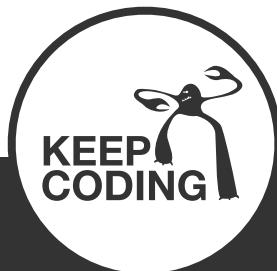
```
{  
  "code" : 1024,  
  "message" : "Validation Failed",  
  "errors" : [  
    {  
      "code" : 5432,  
      "field" : "first_name",  
      "message" : "First name cannot have fancy characters"  
    },  
    {  
      "code" : 5622,  
      "field" : "password",  
      "message" : "Password cannot be blank"  
    }  
  ]  
}
```



# Buenas prácticas en APIs - cambios

Cuando un cliente solicita un cambio en un recurso (con PUT o PATCH) es bastante común que luego tenga que hacer un GET para recuperar los datos guardados y mostrarlos.

Muchos APIs facilitan esto devolviendo siempre una representación del objeto modificado.



# Buenas prácticas en APIs - filtros, paginación, etc

Usemos la query-string. Por ejemplo:

Para filtros GET /messages?state=sent&archived=false

Para ordenación GET /messages?sort=-priority,created\_at

Para paginación GET /users?skip=30&limit=10&returnTotal=true

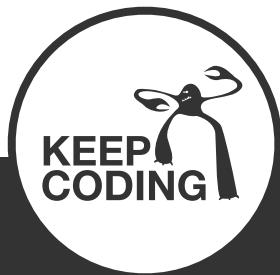
Para búsquedas GET /messages?q=frr149&sort=created\_at



# Buenas prácticas en APIs - alias

Para consultas típicas a nuestro API, quien lo consuma agradecerá que le hagamos alias.

`GET /messages/recently_sent`

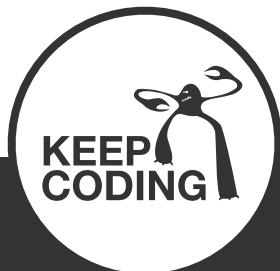


# Buenas prácticas en APIs - reducción de campos

Es muy útil permitir al consumidor que elija si quiere menos o más campos de los que devuelve el recurso por defecto.

```
GET /messages?fields=id,subject,send_date,updated_at&state=sent&sort=-updated_at
```

Algunos usan una cabecera como por ejemplo x-extended o similar para obtener todos los campos disponibles. Cuando no se especifica esta cabecera se devolverían los campos de mayor uso.

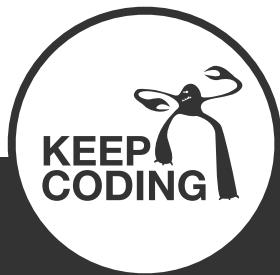


# Buenas prácticas en APIs - HATEOAS o no?

Hypermedia As The Engine Of Application State

Consiste en que nuestro API devuelva links a recursos o acciones que se pueden hacer con los recursos.

```
{  
  "content": [ {  
    "price": 499.00,  
    "description": "Apple tablet device",  
    "name": "iPad",  
    "links": [ {  
      "rel": "self",  
      "href": "http://localhost:8080/product/1"  
    } ],  
    "attributes": {  
      "connector": "socket"  
    }  
  }  
}
```

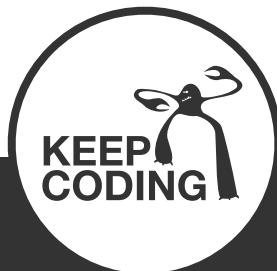


# Buenas prácticas en APIs - timestamps

Permitir solicitar al API los elementos que hayan cambiado desde la última vez que los descargamos es muy bueno para que las apps usen el menor ancho de banda posible y la experiencia sea mejor.

Esto lo conseguimos devolviendo el timestamp actual en cada lista, e implementando un filtro opcional al que pasarle dicho timestamp para que el API nos de solo los elementos que hayan cambiado desde el momento del timestamp.

Los elementos que nos dará serán o nuevos (tendrán un ID que nosotros no tenemos), actualizados (los localizamos por el ID) o borrados (localizados por ID y llevarán una marca de 'DELETED' para que los borremos).



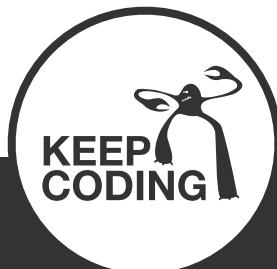
# Buenas prácticas en APIs - method override

Hay clientes que solo pueden usar GET y POST por ejemplo.

Por tanto no podrán llamar a nuestros PUT, PATCH o DELETE.

Para solucionar esto habilitamos una cabecera llamada '*X-HTTP-Method-Override*' (por convención) con un string que contendrá uno de los métodos PUT, PATCH or DELETE.

Usemos esto solo para POST! (los métodos GET nunca deben hacer modificaciones!)

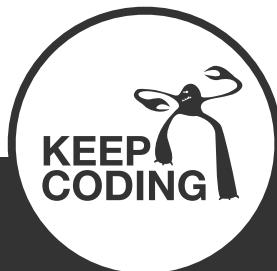


# Buenas prácticas en APIs - rate limiting

Para evitar abusos del API se suele implementar algún sistema que controle e informe al consumidor sobre los límites de llamada al API.

Hay muchos módulos de terceros que nos darán estas habilidades, como por ejemplo:

- <https://github.com/jhurliman/node-rate-limiter>
- <https://github.com/ded/express-limiter>

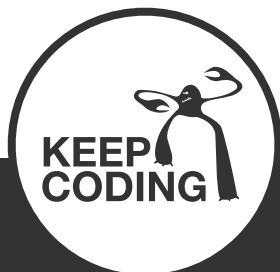


# Buenas prácticas en APIs - CORS

Aunque los primeros clientes de nuestro API sean aplicaciones móviles nativas, es bastante posible que más adelante algún cliente web tenga que consumirlo. Para facilitarlo, habilitemos CORS en nuestro servicio.

[http://enable-cors.org/server\\_expressjs.html](http://enable-cors.org/server_expressjs.html)

```
app.use(function(req, res, next) {  
  res.set("Access-Control-Allow-Origin", "*");  
  res.set("Access-Control-Allow-Methods", "GET, POST, PUT, DELETE");  
  res.set("Access-Control-Allow-Headers", "Origin, X-Requested-With,  
Content-Type, Accept");  
  next();  
});
```

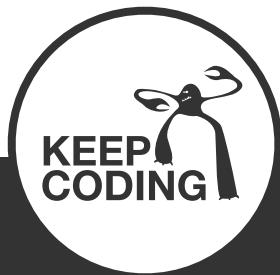


# Buenas prácticas en APIs - HTTPS

## HTTPS everywhere - all the time!

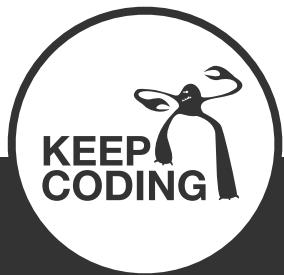
Nuestra API va a ser consultada desde wifis de aeropuertos, bares, bibliotecas, etc.

A veces se implementa que una petición HTTP redirige a la misma con HTTPS. No hagamos esto, en vez de eso devolvamos un error, hagamos saber claramente **que nuestra API no usa métodos inseguros**.

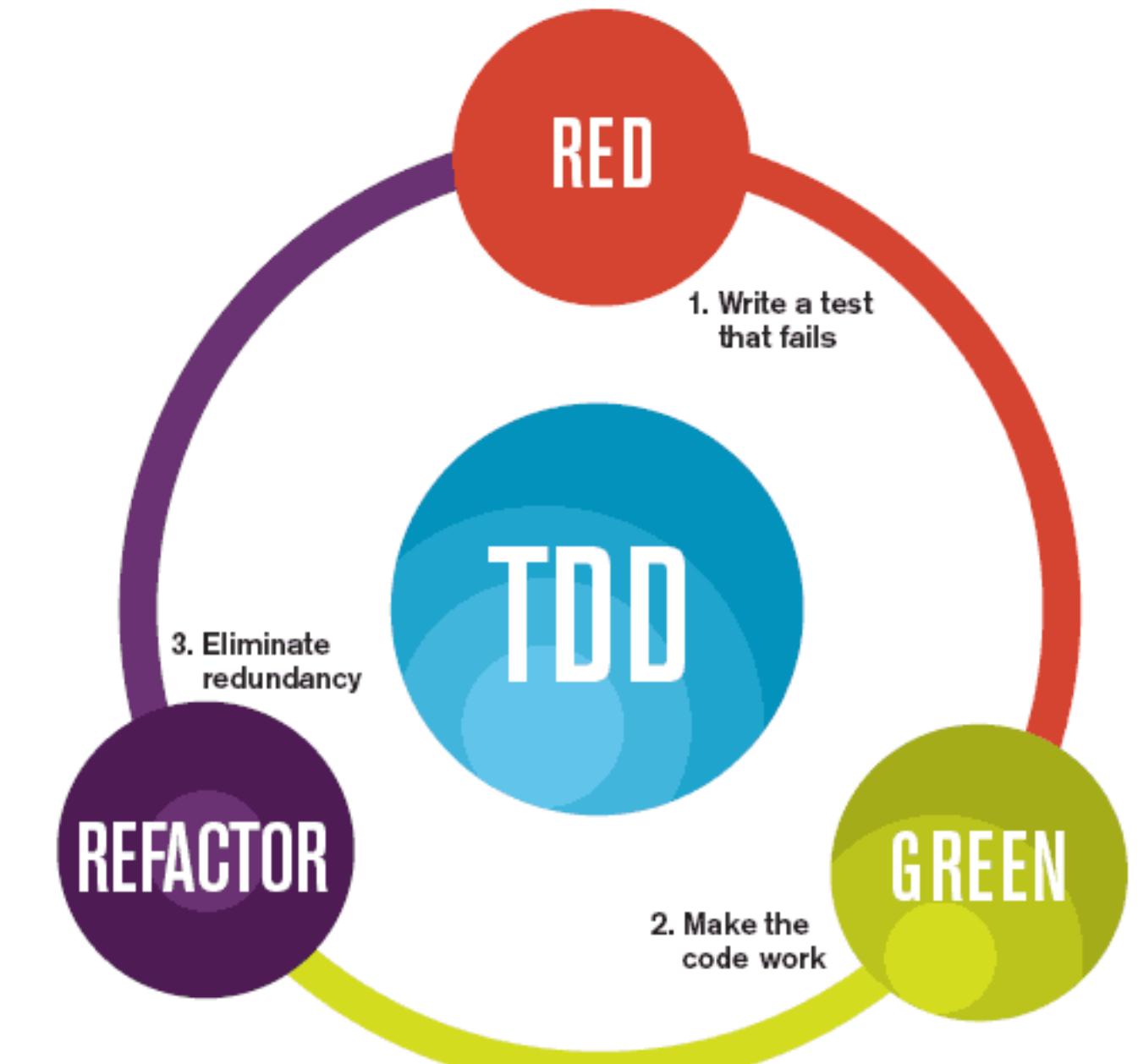


# ■ ¿Qué es TDD?

- Es un proceso de desarrollo de software. Una manera de desarrollar software.
- La base de TDD es escribir programas que prueban nuestro programa para que, cuando introduzcamos cambios, las pruebas estén automatizadas (y no tengamos que volver a ejecutar todas las pruebas manualmente).
- Al principio puede parecer que hace nuestro desarrollo más lento: no es así. A la larga, ahorras mucho tiempo (y disgustos).



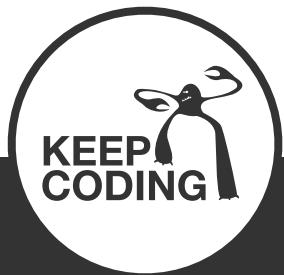
# ■ El mantra de TDD: Red, Green, Refactor



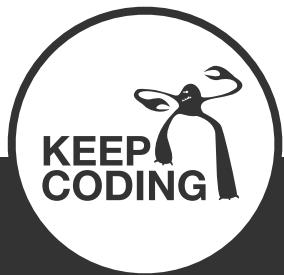
The mantra of Test-Driven Development (TDD) is “red, green, refactor.”



- Debemos escribir un test que falle.
- Es un **requisito obligatorio: el test debe fallar** al menos una vez.
- La idea es hacer que los bugs nos exploten en la cara (y no en producción).

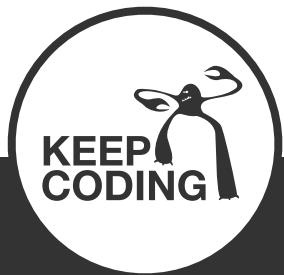


- Debemos **hacer que el test pase.**
- Incorporando el menor código posible.
- Se admiten chapuzas (ya las limpiaremos luego)



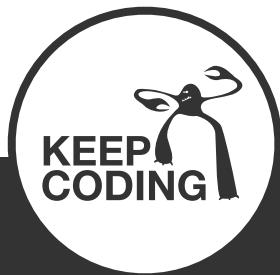
# ■ Refactor

- Una vez el test pasa, debemos **refactorizar** el código
- Eliminar las chapuzas
- Eliminar duplicidades de código



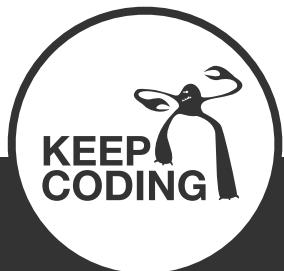
# ■ Características de un buen test

- Es rápido
- Es repetible: siempre obtiene el mismo resultado
- Tiene un nombre claro y explícito
- Sólo comprueba una cosa (es atómico)
- No debe depender del entorno (internet, bases de datos, etc.)



# ■ Tipos de tests

- **Unit tests.** Aseguran que partes individuales de la app, como funciones y clases, funcionan como se espera. Las aserciones prueban que esas partes retornan el resultado esperado para distintas entradas.
- **Integration tests.** Aseguran que la colaboración entre partes de la app es como se espera. Las aserciones pueden probar un API, UI, o interacciones que pueden tener efectos colaterales (como BD, I/O, logging, etc)
- **End-to-end tests.** Aseguran que el software en su conjunto funciona como se espera desde el punto de vista del usuario. Las aserciones principalmente prueban el interfaz de usuario.





Gracias  
Javier Miguel - @javiermiguelg

