

Status 429: Too Many Requests

Módulo 4: REST en producción

Repaso

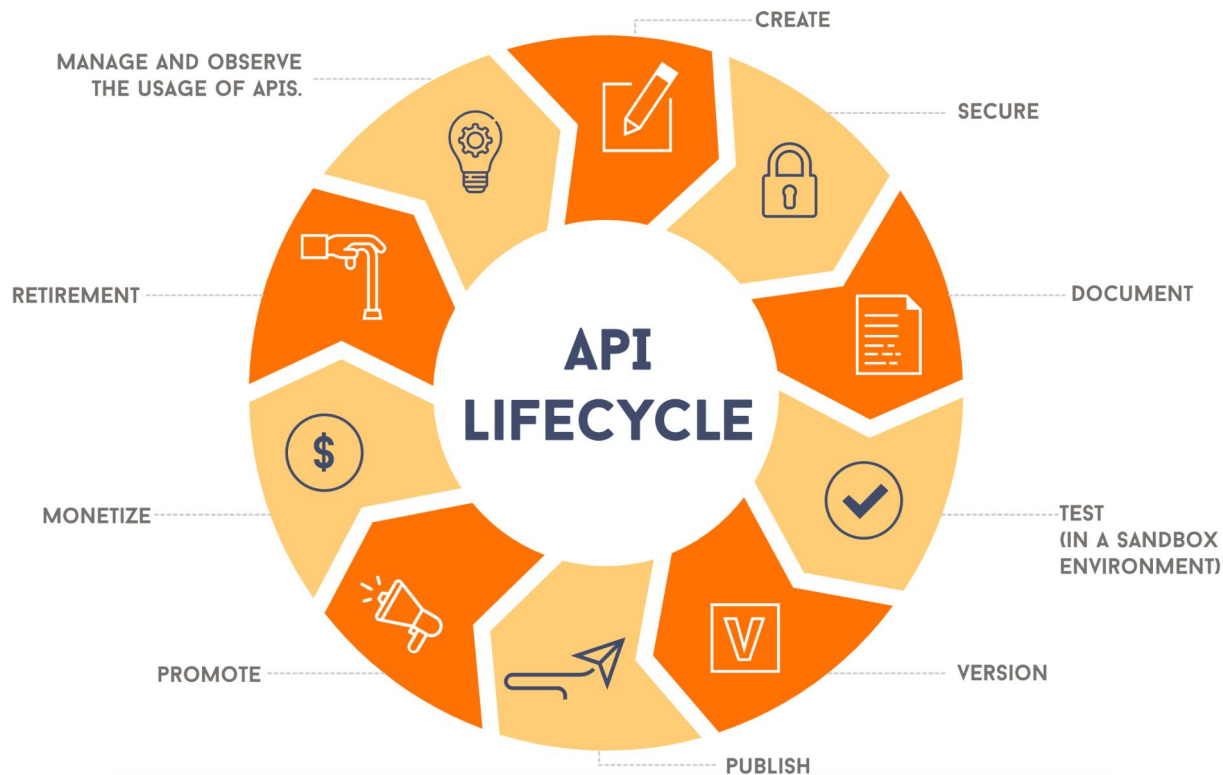
Las APIs REST en teoría:

- Modelo madurez de Richardson
- HATEOAS
- La web como plataforma de aplicaciones

Las API REST en la práctica:

- Buenos patrones de diseño: cómo usar URIs, como usar status codes, ...
- OAS3

Pero diseñar una API es sólo el comienzo...



En la clase de hoy

Nos metemos de lleno en distintas técnicas para poner APIs en entornos productivos:

- Cómo *mockear* una API
- Clientes/SDK para una API
- Cómo implementar nuestra API
- Diffing de APIs
- Cómo testear APIs
- Cómo hacer que nuestra API sea segura
- Cómo monitorear APIs

Mocking

Las APIs permiten desacoplar el trabajo de 2 equipos: típicamente Frontend (FE) y Backend (BE).

Una buena metodología:

1. FE + BE se juntan para diseñar una API en conjunto
2. El equipo de BE se ocupa de construir la implementación de la API
3. El equipo de FE consume la API

Problema: ¿cómo hace el equipo de FE para probar su parte del sistema si la API aún no está implementada?

Mocking

El ***mocking*** de una API nos permite simular su comportamiento.

Veamos varios ejemplos

- 1) Mocking con SwaggerHub
- 2) Mocking con Node.js
 - a) [openapi-mock-express-middleware](#)
 - b) [swagger-mock](#)

La ventaja de los últimos 2 es que podemos extenderlos con más lógica.

Cientes / SDKs

Si bien una API bien documentada alcanza...

... a veces está bueno poder usar funciones de nuestro propio lenguaje de programación

Similar al concepto de RPC:

- Un cliente/SDK nos permite encapsular un llamado a la API como si fuera una función
- Se ocupa de codificar el request y decodificar la respuesta

Cientes / SDKs

Estos clientes se pueden armar a mano:

- Muy trabajoso y difícil de mantener al día

También se pueden generar a partir de la especificación OAS de la API

→ Veamos el caso con Python

<https://github.com/gdecaso/curso-apis/tree/main/heladeria-api-client>

Clientes / SDKs

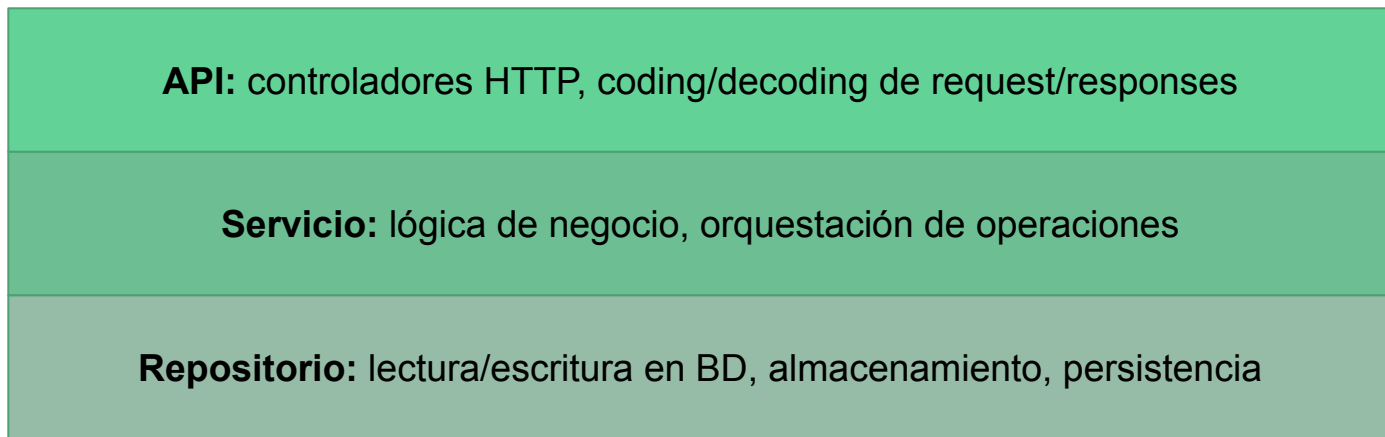
Herramienta para crear clientes automáticamente a partir de OAS3

<https://github.com/OpenAPITools/openapi-generator>

Implementando una API

Hay infinidad de opciones, en todos los lenguajes

Importante separar la lógica de nuestra implementación en 3 capas:



Implementación: capa de API

API
Servicio
Repositorio

Encargada de lidiar con los aspectos del protocolo HTTP:

- Esperar requests, por ejemplo en el puerto 80
- Procesar los headers y el cuerpo de los requests. Por ej: parsear JSON
- Rechazar requests erróneos. Por ejemplo `http://.../ggustos` → 404
- Rutear el request al código/método adecuado
- Codificar la respuesta de nuestra implementación, por ej: escribir JSON
- Capturar excepciones y convertirlas a response codes HTTP adecuados

Una parte importante de esta capa se puede autogenerar a partir de OAS3

Implementación: capa de API

API
Servicio
Repositorio

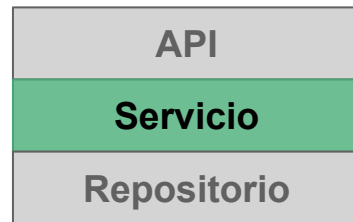
La capa de API lidia con objetos que llamamos **DTOs**: Data Transfer Objects

- Estos son los tipos de datos definidos en la API
- Queremos desacoplar los DTOs de los tipos de datos de nuestro servicio/repositorio
 - Si hacemos un refactor en nuestros tipos, no tenemos que tocar la API
 - Si cambian los tipos de la API, no tenemos que tocar nuestros tipos

¡Veamos un ejemplo!

<https://github.com/gdecaso/curso-apis/tree/main/heladeria-server-springboot/src/main/java/com/heladeriaapilia/api>

Implementación: capa de servicio



Encargada de lidiar con la lógica de negocio de nuestro software.

- Validaciones de negocio
 - Por ej: no se puede agregar un gusto inexistente a un pote
 - Si estas validaciones estuvieran en la capa de API, estamos expuestos si otra parte del código llama a esas funciones
- Conexión con otros módulos.
 - Por ej: sistema de cobranza de tarjetas de crédito
- Nuestra capa de servicio sirve para otros procesos:
 - Por ej: migración de pedidos desde un sistema legacy al nuestro

<https://github.com/gdecaso/curso-apis/tree/main/heladeria-server-springboot/src/main/java/com/heladeriaapilia/service>

Implementación: capa de repositorio

API
Servicio
Repositorio

Encargada de lidiar con la persistencia de nuestros objetos, ya sea en una BD u otros medios.

- Mapping de objetos a tablas
 - Ej: relación one-to-many pedido/potes
- Búsqueda de objetos por criterios.
 - Ej: buscar un pedido según su id
- Otras configuraciones relativas a la BD
 - Ej: migraciones de datos/esquema

Spring REST DEMO - agregamos el método GET /pedidos



Estudiando la evolución de una API: Diffing

¿Qué pasa si somos clientes de una API y ...

- ... le quitan un recurso?
- ... le renombran un query parameter?
- ... le agregan un atributo a uno de sus responses?

Algunos de estos cambios son **compatibles hacia atrás** y otros no.

Hay herramientas como [openapi-diff](#) que nos permiten analizar esto de manera automática.

DEMO: openapi-diff



<https://github.com/gdecaso/curso-apis/tree/main/heladeria-api-spec/oas3/diff>

Testeando nuestra API

Lo primero: **test unitarios**



Queremos testear la lógica de nuestros controladores

- **Independientemente** del resto del código

Para esto vamos a *mockear* las capas de abajo.

Veamos: `GustosApiControllerTest.java`

Testeando nuestra API

Lo siguiente: **tests de integración**

API
Servicio
Repositorio

Queremos testear nuestra aplicación en conjunto

- Típicamente con una BD en memoria

En Java contamos con la ayuda de frameworks tales como RestAssured

Veamos: `BasicTest.java` (sin RestAssured) y `RestAssuredTest.java`

Algunos desafíos de los tests de integración

- ¿Qué secuencias de invocaciones a la API voy a ejercitar?
- ¿Con qué parámetros voy a hacer cada llamado?
- ¿Cómo preparo los payloads de cada llamado (e.g. JSON)?
- ¿Cómo elijo los datos para insertar si mi aplicación tiene las bases de datos (SQL/NoSQL) o persistencia?
- ¿Cómo restablezco **eficientemente** el estado inicial de la aplicación luego de cada ejecución?
- Objetivos:
 - Encontrar fallas y vulnerabilidades
 - Maximizar la ejercitación del código

Escribir casos de tests manualmente para cada endpoint lleva mucho esfuerzo y tiempo

Generación automática de tests para nuestra API (Fuzzing)



```
Image timer;  
Thread timerThread;  
public void init() {  
    count = 0;  
    lastcount = 0;  
    pictures = new Image[50];  
    mediatrackerTracker = new MediaTracker(this);  
    for (int a = 0; a < lastcount; a++) {  
        pictures[a] = getImage(a);  
        mediatrackerTracker.addImage(pictures[a], 0);  
    }  
    timerThread = new Thread(timer);  
    timerThread.start();  
}  
  
public void start() {  
    if (timer == null) {  
        timer = new Thread(timerThread);  
        timer.start();  
    }  
}  
  
public void paint(Graphics g) {  
    g.drawImage(pictures[0], 0, 0, 100, 100, this);  
    if (count == lastcount) {  
        run();  
    }  
}
```

Mi API (ej: OpenAPI)

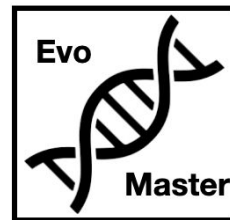
Crea, Ejecuta y Elige
Tests sobre la API

Test Case Generator

```
@Test  
public void test0() throws Exception {  
  
    given().header("Authorization", "ApiKey user")  
        .accept("*/*")  
        .get("www.foo.com/api/v1/media_files/42")  
        .then()  
        .statusCode(200);  
}
```

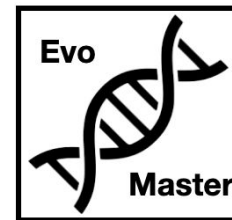
Test Suite Automáticamente
Creado
(ej: Rest-Assured)


EvoMaster






- Tool para generar automáticamente tests para REST-APIs
- Blackbox:
 - Puede ser usada sobre cualquier aplicación que exhiba OpenAPI/GraphQL
- Whitebox:
 - Aprovecha información estructural y de runtime de la aplicación (cobertura de líneas, seeding, instrumentación, dynamic taint analysis, etc.)
 - Actualmente soporta lenguajes JVM (Java & Kotlin)
- Utiliza **algoritmos evolutivos** (Search-based Software Testing), que es una rama de la optimización combinatoria
 - Algoritmo “Many Independent Objective” o MIO


Open source en github: www.evomaster.org










[Pull requests](#) [Issues](#) [Marketplace](#) [Explore](#)





 **EMResearch / EvoMaster** Public

 Edit Pins  Unwatch **19**  Fork **42**  Starred **255**

[Code](#) [Issues](#) **6** [Pull requests](#) **4** [Discussions](#) [Actions](#) [Projects](#) [Wiki](#) [Security](#) 

 **master**  **54 branches**  **13 tags**

[Go to file](#) [Add file](#) [Code](#)

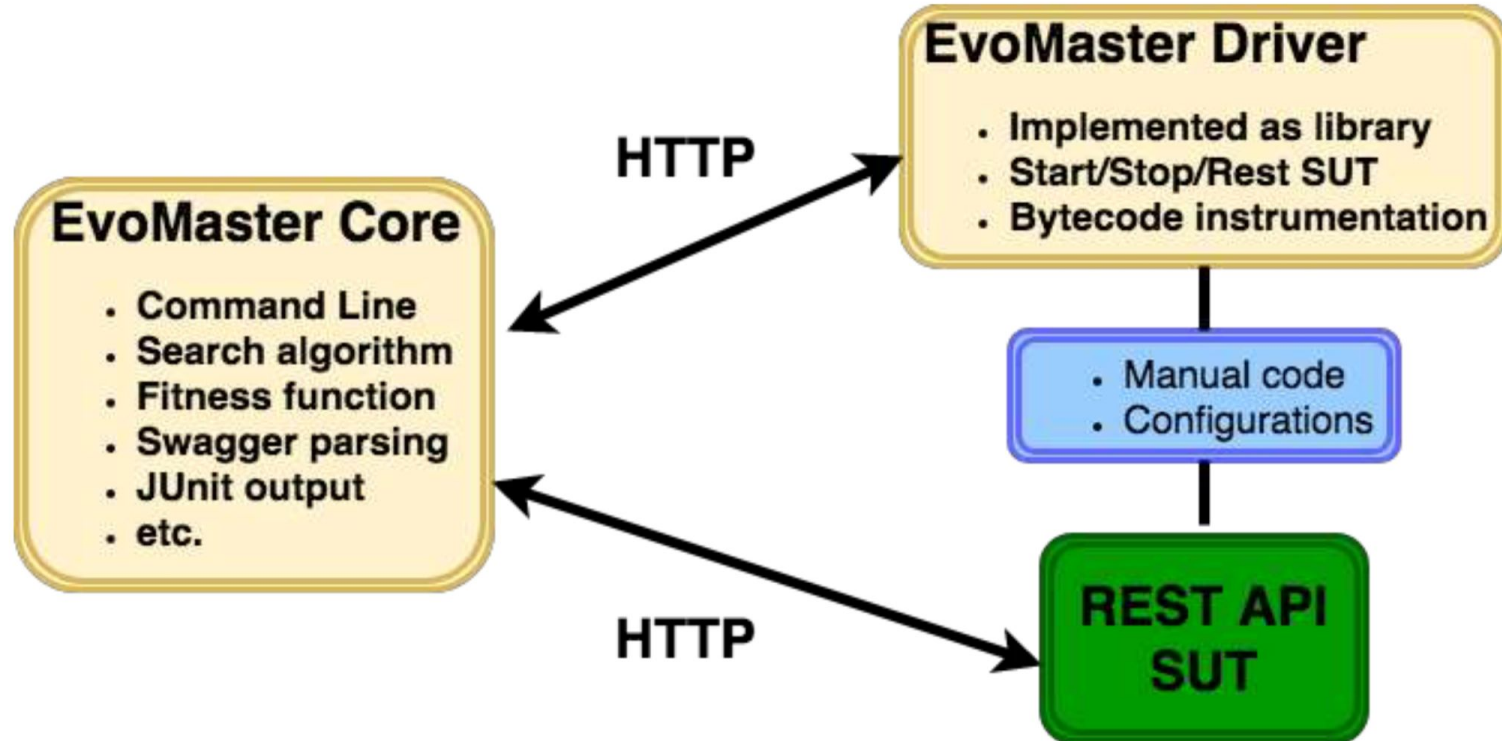
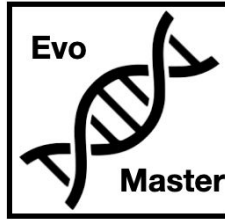
 arcuri82 scripts for comparison tools ✓ 9f63a8f 15 days ago 🕒 6,210 commits
 .circleci clarification 11 months ago
 .github 1.5.1-SNAPSHOT last month
 client-dotnet 1.5.1-SNAPSHOT last month

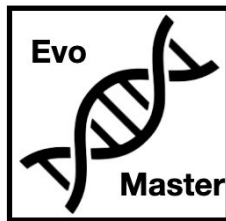
About

The first open-source AI-driven tool for automatically generating system-level test cases (also known as fuzzing) for web/enterprise applications. Currently targeting whitebox and blackbox testing of Web APIs, like REST and GraphQL.

[kotlin](#) [java](#) [testing](#) [graphql](#) [rest](#)

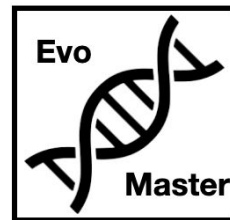
EvoMaster: Arquitectura





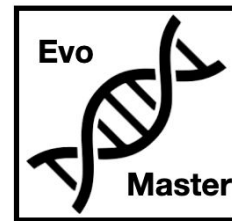
Many-Independent Objectives Algorithm (MIO)

- Dado z testing targets **no cubiertos**, mantiene una lista de tests por cada testing target
 - Si un target es cubierto, se queda con el test que lo hizo y descarta los otros de la lista
- n = cantidad de tests por cada testing target z
- Pr = Probabilidad de generar un nuevo test al azar o mutar un test existente
- F = Porcentaje de tiempo Exploración/Explotación
- Ck = cantidad de veces que se mutó un test de esta lista sin que produzca mejoras de cubrimiento (Feedback-Directed Sampling)



EvoMaster en nuestro ejemplo

- Ejecutamos EvoMaster sobre heladeriaapilia durante 10 min
- EvoMaster produjo 28 Casos de Tests REST-assured
 - 11 Tests con status codes exitosos (200, 201)
 - 12 Tests con status codes no exitosos pero esperados (400, 401, 403, 404)
 - 5 Tests encontraron status code 5xx (fallan)
 - El Test Suite alcanza una cobertura de Line 49%, Method 49%, Class 86%
- El controller para la heladeriaapilia que usamos y los tests generados se pueden encontrar en el branch “evomaster” de curso-api
 - <https://github.com/gdecaso/curso-apis/blob/evomaster/heladeria-server-springboot/src/test/java/com/heladeriaapilia/EmbeddedEvoMasterController.java>

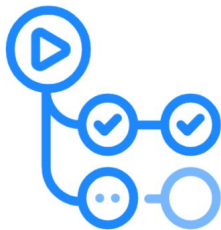


Muchos desafíos abiertos

- Campo muy activo tanto en la academia (doctorado) y en la industria (I+D)
- Algunos desafíos actuales:
 - ¿Cómo ejercitamos escenarios más complejos?
 - ¿Cómo mejoramos la performance del setup/teardown de cada ejecución?
 - ¿Cómo sabemos si una ejecución está exhibiendo una falla en la lógica de la API (más allá de los 5xx)?
 - ¿Por cuánto tiempo nos conviene ejecutar el generador de tests?
 - ¿Cómo integramos la generación de tests en el proceso de desarrollo?
 - ¿Cómo integramos la generación de tests con **integración continua**?

Testeando nuestra API - Continuous Integration (CI)

Con [GitHub Actions](#) (u otro CI) podemos hacer que los tests corran automáticamente.



GitHub Actions

<https://github.com/gdecaso/curso-apis/actions>

```
name: Java CI with Maven

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  build:

    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout @v3
      - name: Set up JDK 11
        uses: actions/setup-java @v3
        with:
          java-version: '11'
          distribution: 'temurin'
          cache: maven
      - name: Build with Maven
        working-directory: heladeria-server-springboot
        run: mvn -B package --file pom.xml
```

Otros tipos de tests para nuestra API

Performance/Carga: ver qué tan rápida es nuestra API y qué tanta carga soporta

- JMeter (¡demo! )

Usability:

- Darle nuestra API a developers y estudiar qué tan fácil les resulta usarla

Security:

- Darle nuestra API a un “red team” y ver qué vulnerabilidades encuentran

Brindándole seguridad a nuestra API

Seguridad es un tema MUY amplio

- Cómo aportar seguridad a una API sería un curso entero
- Pero no podemos dejar de mencionar al menos algunos conceptos básicos

Para empezar: **no alcanza con HTTPS**



Autenticación vs. autorización

Autenticación

La API verifica que el cliente es quien dice ser.

Por ej: mediante el uso de usuario y contraseña

En caso de que falla la autenticación usamos **401 UNAUTHORIZED***



*: sí, es muy confuso que “unauthorized” se use para algo que no tiene que ver con autorizar

Autenticación vs. autorización

Autorización

La API verifica que el cliente ya autenticado tiene derecho a realizar cierta tarea.

Por ej: un administrador de la heladería puede quitar gustos de helado. Otros usuarios no.

En caso de que falla la autorización usamos **403 FORBIDDEN**



Autenticación vs. autorización

Ejemplo en la vida real

En la ventanilla del banco nos piden nuestro DNI para comprobar que somos quien decimos ser.

→ Autenticación

Cuando pedimos un préstamo nos lo pueden denegar o no en función de nuestra historia crediticia.

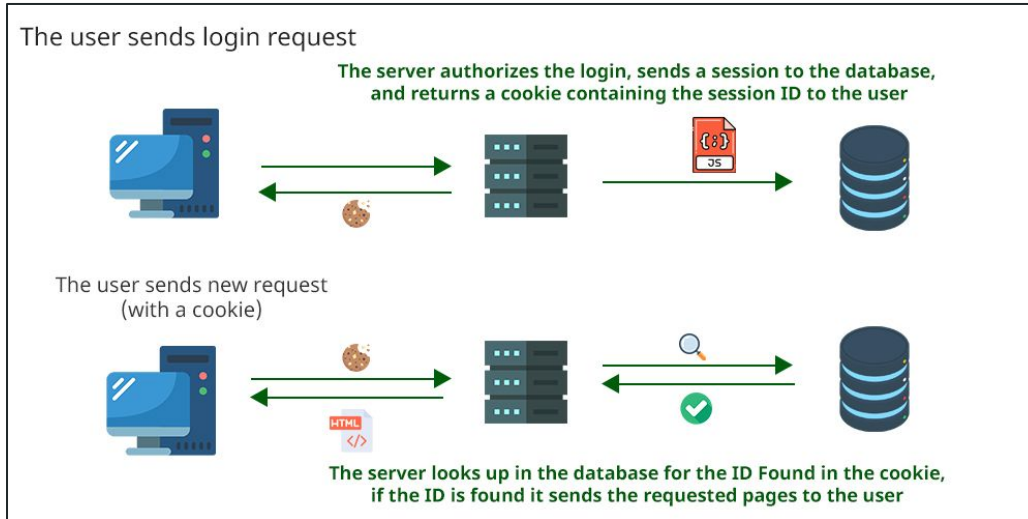
→ Autorización

Autorización basada en sesiones

El servidor almacena una **sesión** por cada usuario logueado.

El usuario recibe un **id de sesión**.

- Futuros requests incluyen el id de sesión.

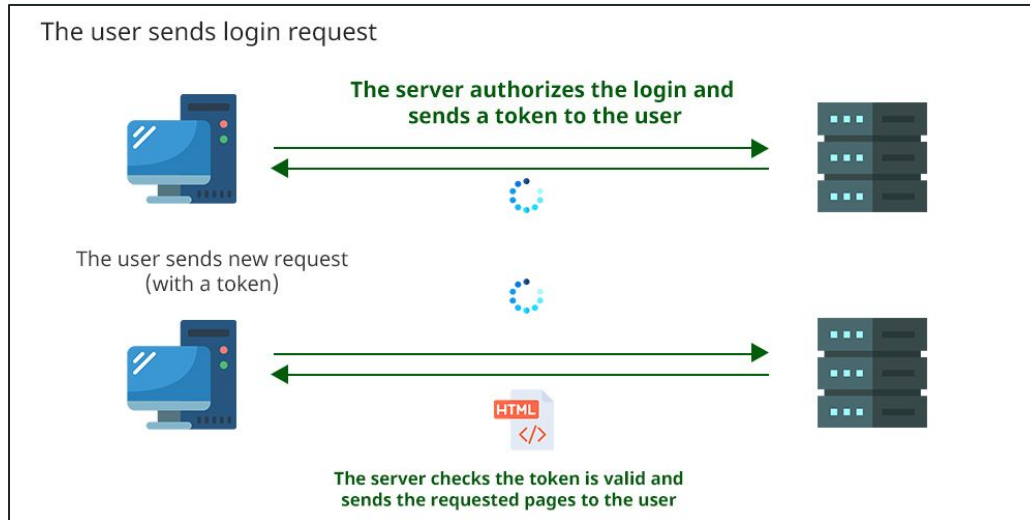


Autorización basada en tokens

El servidor genera un **token** que codifica la información de autorización de forma cifrada.

El usuario recibe un el token y lo guarda localmente.

- Futuros requests incluyen el token



Comparativa pros/cons

Debería llamarse Autorización por token



Autenticación por sesión	Autenticación por token
El servidor almacena el estado	El cliente almacena el estado (cifrado)
Más difícil de escalar el servidor	El servidor puede ser <i>stateless</i>
El servidor puede invalidar sesiones por ejemplo si cree que algún cliente fue comprometido	El token se usa para autorizar, pero se confía ciegamente que quien lo tiene es quien dice ser
Ideal para páginas web con <i>login</i>	Ideal para comunicación server-to-server, es decir APIs



Susceptible a ataques CSRF

Más simple: autenticación basada en API keys

Para APIs también podemos hacer algo bien sencillo:

1. Cuando un cliente se registra solicita una API key
2. El servidor genera una API key unívoca y secreta y se la envía al cliente
 - Tiene que ser por un canal **secundario** seguro.
Por ej: registración en una página web encriptada con HTTPS
3. Cuando el cliente llama a la API, incluye el API key (en query param o header)

A veces el sistema se complejiza un poco con una API key y uno o varios API secrets.

Seguridad de APIs en OAS3

Veamos un ejemplo para nuestra heladería, con una simple API key.

```
components:
  securitySchemes:
    ApiliaApiKey:
      type: apiKey
      in: header
      name: x-api-key
```

```
openapi: 3.0.0
info:
  description: ...
  version: "1.0.0"
servers:
  - url: 'https://api.heladeria-apilia.com'
security:
  - ApiliaApiKey: []
```

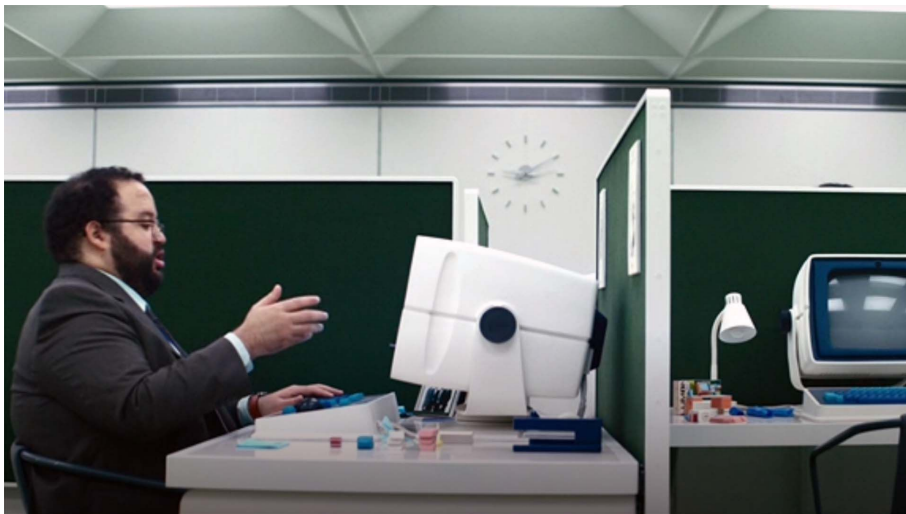
Por defecto, las URIs requieren API key

```
paths:
  /gustos:
    get:
      description: Listar los gustos de helado
      parameters:
        - in: query
          name: tipo
          required: false
          schema:
            $ref: '#/components/schemas/TipoDeGusto'
      security: []
      responses: . .
```

Esta URI no requiere API key

Seguridad de API en nuestra implementación

1. Primero descomentamos el código en `com.heladeriaapilia.security.SecurityConfig`
2. Ahora probamos requests sin x-api-key y con x-api-key a `/gustos` y `/pedidos`



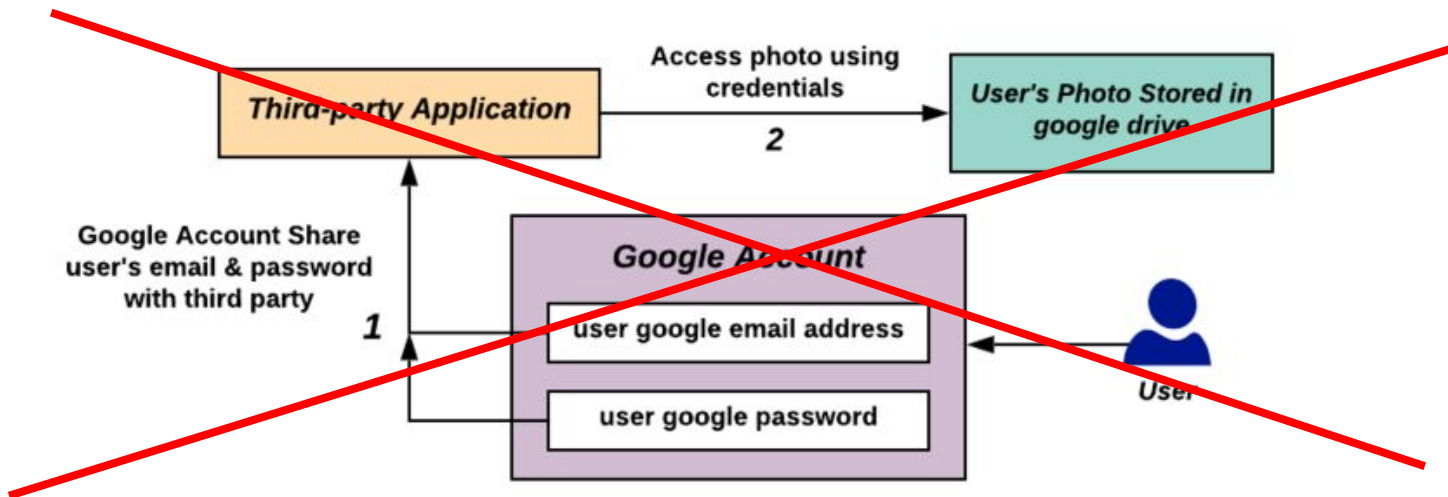
<https://github.com/gdecaso/curso-apis/tree/main/heladeria-server-springboot/src/main/java/com/heladeriaapilia/security>

Delegando acceso

Supongamos que tenemos una API que permite acceder a ciertos recursos de cada usuario

- Por ej: /emails, /contactos, /documentos

Y supongamos que **otro sistema** quiere acceder a esos recursos por parte de sus usuarios



Delegando acceso: OAuth

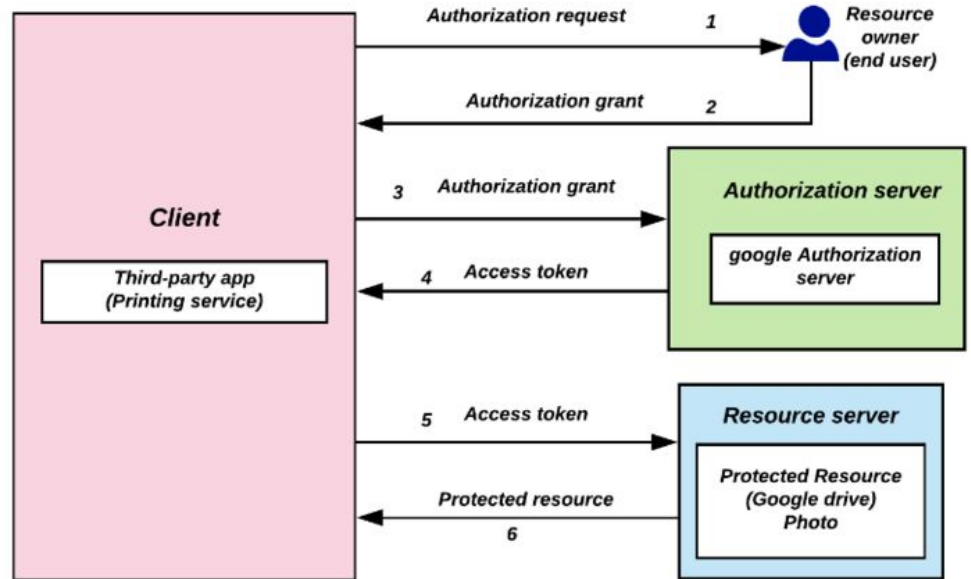


OAuth es un mecanismo para delegar acceso **sin compartir contraseñas**.

- 2006: propuesta en Twitter
- 2010: estandarizado RFC v1.0
- 2012: estandarizado RFC v2.0

Basado en tokens

Hoy es soportado por Amazon, Google, Facebook, Twitter, Microsoft, ...





Delegando acceso: OAuth - Demo

En el browser:

1. Navegamos al primer request para tener el ok del usuario
 - a. Obtenemos **access code**

En Postman:

2. Canjeamos access code por **token**
3. Accedemos a la API de Google Drive



Monitoreando nuestra API

¿Cómo sabemos qué tanto uso está teniendo nuestra API?

¿Y la latencia?

¿Y cuántos errores y qué tipo de errores son más comunes?

⇒ para responder todo esto tenemos que agregar alguna solución de **monitoring**

(Paréntesis: desplegamos nuestra API en Railway)

Railway es una nube que nos permite fácilmente y gratuitamente desplegar nuestro código.



Veamos el status de la app deployada

Y podemos hacer algunos requests sobre la app misma:

- UI: <https://curso-apis-production.up.railway.app/guidodecaso/heladeria/1.0.0/swagger-ui/>
- O directo
<https://curso-apis-production.up.railway.app/guidodecaso/heladeria/1.0.0/gustos>

Vamos a pegarle duro a la API



Ahora sí: monitoreando nuestra API

Infinidad de soluciones de monitoring de apps y algunas con foco en APIs:

Por ejemplo: Moesif

- Configuración → MonitoringConfig.java
- [Dashboard](#)

Esta solución de monitoring requiere cambios de código.

- Pero también se puede obtener monitoring a través de API Gateways (clase que viene)