

Tema 10

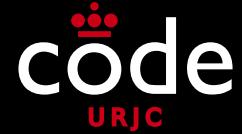
Testing de APIs REST

Micael Gallego
micael.gallego@gmail.com
[@micael_gallego](https://twitter.com/micael_gallego)

Testing de APIs REST

- Pruebas funcionales de APIs REST
- Pruebas de carga de APIs REST

Pruebas funcionales de APIs REST



- Las pruebas funcionales de una API REST consisten en **verificar** que los diferentes **endpoints** tienen el comportamiento esperado en función de los parámetros (URL, query params, body...)
- Existen **librerías** que facilitan la escritura de este tipo de **tests tan específicos**

- ## Supertest

- Super-agent driven library for testing node.js HTTP servers using a fluent API.

```
$ npm install supertest --save-dev
```

<https://github.com/visionmedia/supertest>

Pruebas funcionales de APIs REST



ejem1

- Es conveniente dividir la app en **dos módulos** para que los tests puedan acceder a la app express

app.js

```
var app = require('./server')
app.listen(3000, () => { console.log('Server started in port 3000') });
```

const express = require('express');

server.js

```
const app = express();
app.use(express.json());
```

```
app.get('/ads', (req, res) => { ... });
```

```
module.exports = app;
```

Pruebas funcionales de APIs REST

ejem1

- Superagent proporciona un API para hacer peticiones REST y verificar el resultado obtenido

app.test.js

```
const app = require('./server')
const supertest = require('supertest')

const request = supertest(app)

test('gets the ads endpoint', async () => {
  const response = await request.get('/ads')
    .expect('Content-type', /json/)
    .expect(200)
  expect(response.body.length).toBe(3);
})
```

Expects de superagent

Expects de jest sobre la respuesta

Ejercicio 1

- Implementa tests funcionales de la API REST del Ejemplo 3 del Tema 7 (API REST de anuncios)

Testing de APIs REST

- Pruebas funcionales de APIs REST
- Pruebas de carga de APIs REST

Pruebas de carga de APIs REST

- Las pruebas de carga son una de las **pruebas no funcionales** más habituales
- Se suelen simular **múltiples usuarios** accediendo simultáneamente a la misma **API REST**
- A diferencia de las funcionales, a veces no existe un “**criterio**” para determinar si la prueba debe pasar o no

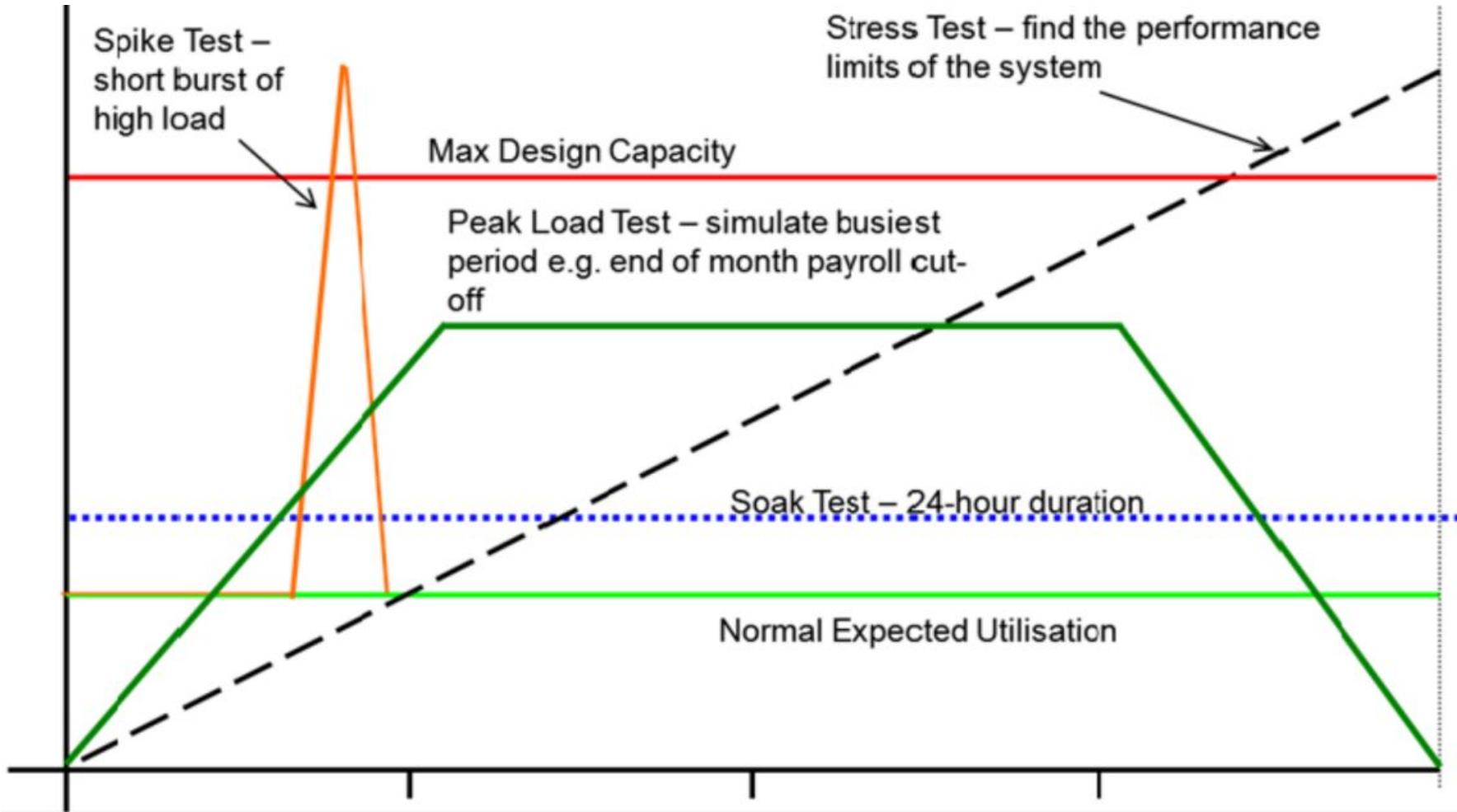
Pruebas de carga de APIs REST

- **Pruebas de Rendimiento:** verifican que el SUT cumple con el rendimiento esperado que se puede medir en función del número de transacciones por segundo, tiempo máximo de generación de respuesta, etc.
- **Prueba de carga o de esfuerzo o escalabilidad:** verifican que el SUT cumple con el comportamiento y rendimiento esperado bajo una pesada carga de los datos, la repetición de ciertas acciones de los datos de entrada, los grandes valores numéricos, consultas grandes a una base de datos o para comprobar el nivel de los usuarios concurrentes

Pruebas de carga de APIs REST

- **Prueba de estabilidad:** es una prueba que se ejecuta durante un tiempo largo y que busca anomalías como pérdidas de memoria u otras degradaciones por la continuidad de la ejecución
- **Pruebas de estrés o volumen:** donde se escala la cantidad de carga con el tiempo hasta que se encuentren los límites del sistema; con el objetivo de examinar cómo falla y vuelve a su funcionamiento normal

Pruebas de carga de APIs REST



Pruebas de carga de APIs REST

- Necesitamos herramientas que nos permitan ejercitar el SUT en **diferentes condiciones de carga/volumen** y tomar las **medidas de su comportamiento**
- Hay que definir **qué métricas** medir y cómo hacerlo

Pruebas de carga de APIs REST

- En ocasiones se definen **umbrales** sobre esas métricas, o se **compara** con los valores de una **versión previa** (para **evitar regresiones**)
- Se pueden hacer en **fases tempranas del desarrollo** para mantener los parámetros de rendimiento en valores aceptables en todo momento

Pruebas de carga de APIs REST

- **Medidas más utilizadas en APIs REST**
 - **Latencia de las peticiones:** Tiempo desde que se envía una petición hasta que se empieza a recibir la respuesta
 - **Número / % de peticiones incorrectas:** Estado http diferente de 2xx
 - **Ancho de banda:** bytes por segundo
 - **Número de peticiones por segundo:** Depende del número de usuarios (hilos) en paralelo ejercitando el SUT

Pruebas de carga de APIs REST

- ¿Cómo se mide la **latencia** de las peticiones?
 - **Min:** Mínima
 - **Max:** Máxima
 - **Median (p50):** Mediana. Si ordenamos los valores de menor a mayor, el que ocupa la posición central (No es el valor medio)
 - **p95:** La latencia mayor del 95% de las peticiones con menor latencia. Sólo el 5% está por encima de ese valor
 - **p99:** La latencia mayor del 99% de las peticiones con menor latencia. Sólo el 1% está por encima de ese valor

Pruebas de carga de APIs REST

ARTILLERY.IO

Complete report @ 2019-01-02T17:32:36.653Z

Scenarios launched: 300

Scenarios completed: 300

Requests completed: 600

RPS sent: 18.86

Request latency:

min: 52.1

max: 11005.7

median: 408.2

p95: 1727.4

p99: 3144

Scenario counts:

0: 300 (100%)

Codes:

200: 300

302: 300

Pruebas de carga de APIs REST

ApacheBench



```
Requests per second: 734.76 [#/sec] (mean)
Time per request:   136.098 [ms] (mean)
Time per request:   1.361 [ms] (mean, across all concurrent requests)
Transfer rate:      60645.11 [Kbytes/sec] received
```

Percentage of the requests served within a certain time (ms)

50%	133
66%	135
75%	137
80%	139
90%	145
95%	149
98%	150
99%	151
100%	189 (longest request)

Pruebas de carga de APIs REST

- El entorno debe ser **lo más parecido posible al de producción**
- No se deberían ejecutar otro tipo de procesos o pruebas en este entorno para no afectar a la **fiabilidad** de los resultados
- No se recomienda ejecutar la **herramienta de pruebas de carga** en la misma máquina del SUT

Pruebas de carga de APIs REST

- La mayoría de las veces es necesario **ejecutar estas pruebas de forma distribuida**
 - Varios nodos realizan peticiones sobre el SUT
 - Se pueden simular cargas más parecidas a las reales
 - Vigilar los cuellos de botella
 - Nodos en la misma máquina
 - Ancho de banda de red

Artillery



- Artillery es un **toolkit** para **pruebas de carga** web / REST
- Esta escrita en Node.js y su instalación es muy sencilla.

```
$ npm install -g artillery
```

- Plugins para generar métricas
- Integración con herramientas de visualización: StatsD, Graphana, Graphite, Datadog ..

<https://artillery.io/>

Artillery - Quick

- Podemos hacer una prueba rápida y sencilla a través de la terminal de comandos usando *quick*:

```
$ artillery quick --count 10 -n 20 https://www.urjc.es/
```

- Con **--count <users>** definimos el número de usuarios que van a realizar la conexión de forma simultánea
- Con **-n <requests>** definimos el número de conexiones/peticiones que realizará cada usuario.
- Por defecto, realizará peticiones **GET** sobre la URL que le hemos proporcionado (<https://www.urjc.es/>)

Artillery - Quick

```
Scenarios launched: 10
Scenarios completed: 10
Requests completed: 200
RPS sent: 8.72
Request latency:
  min: 70.7
  max: 1524.5
  median: 456.7
  p95: 732.3
  p99: 1301.7
Scenario counts:
  0: 10 (100%)
Codes:
  200: 200
```

Escenarios
lanzados/completados



Artillery - Quick

```
Scenarios launched: 10
Scenarios completed: 10
Requests completed: 200
RPS sent: 8.72
Request latency:
  min: 70.7
  max: 1524.5
  median: 456.7
  p95: 732.3
  p99: 1301.7
Scenario counts:
  0: 10 (100%)
Codes:
  200: 200
```

Peticiones completadas

Artillery - Quick

```
Scenarios launched: 10
Scenarios completed: 10
Requests completed: 200
RPS sent: 8.72
```

Promedio de peticiones por segundo completadas

```
Request latency:
  min: 70.7
  max: 1524.5
  median: 456.7
  p95: 732.3
  p99: 1301.7
```

```
Scenario counts:
  0: 10 (100%)
Codes:
  200: 200
```

Artillery - Quick

```
Scenarios launched: 10
Scenarios completed: 10
Requests completed: 200
RPS sent: 8.72
Request latency:
```

```
min: 70.7
max: 1524.5
median: 456.7
p95: 732.3
p99: 1301.7
```

Medidas estadísticas de la latencia

```
Scenario counts:
 0: 10 (100%)
Codes:
 200: 200
```

Artillery - Quick

```
Scenarios launched: 10
Scenarios completed: 10
Requests completed: 200
RPS sent: 8.72
Request latency:
  min: 70.7
  max: 1524.5
  median: 456.7
  p95: 732.3
  p99: 1301.7
Scenario counts:
  0: 10 (100%)
Codes:
  200: 200
```

Resultados de escenario

Artillery - Quick

```
Scenarios launched: 10
Scenarios completed: 10
Requests completed: 200
RPS sent: 8.72
Request latency:
  min: 70.7
  max: 1524.5
  median: 456.7
  p95: 732.3
  p99: 1301.7
Scenario counts:
  0: 10 (100%)
Codes:
  200: 200
```

Recuento de códigos HTTP obtenidos

Artillery – Quick

- Opciones del comando *quick*

- `-d, --duration <seconds>` Duración del test
- `-r, --rate <number>` Número de peticiones por segundo
- `-p, --payload <string>` Cuerpo de payload (para peticiones POST)
- `-t, --content-type <string>` Tipo de contenido
- `-o, --output <string>` Fichero de salida
- `-k, --insecure` Desactiva la verificación del certificado TSL
- `-q, --quiet` Modo silencioso (solo muestra resultado final)

Artillery – Test Script

- **Artillery** cuenta con multitud de opciones que podrían ser tediosas de escribir en un solo script con *quick*.
- Nos ofrece un **archivo de configuración** con la extensión **.yml** dónde situar todas las propiedades que conformarán nuestro test de carga.
- Podemos lanzar este fichero de configuración utilizando el comando **run**.

```
$ artillery run script.yml
```

Artillery – Test Script

- Ejemplo básico de un Test Script

ejem2

```
$ artillery run script.yml
```

```
config:  
  target: 'https://www.urjc.es'  
  phases:  
    - duration: 20  
      arrivalRate: 20  
  defaults:  
    headers:  
      x-my-service-auth: 'secret'  
scenarios:  
  - flow:  
    - get:  
      url: "/estudios/grado"
```

Durante 20 segundos

20 usuarios por cada segundo

Acción realizada por cada usuario: Petición GET

Artillery – Test Script

- Los Test Scripts constan de dos secciones:
 - ***config***: Incluye toda la configuración previa a la prueba de carga:
 - *target*
 - *environments*
 - *phases*
 - *payloads*
 - *etc ..*
 - ***scenarios***: Incluye la descripción de las acciones que realizarán los usuarios.

- **Configuración**
 - *target*
 - La URI base de los recursos
 - *environments*
 - Podemos definir una serie de entornos (desarrollo, producción) cada uno con su URL para ejecutar los test de carga.

Artillery – Test Script - Configuración

- Ejemplo target/environments

ejem3

```
$ artillery run -e staging script.yml
```

```
config:  
  environments:  
    production:  
      target: "http://www.ads.es"  
      phases:  
        - duration: 120  
          arrivalRate: 10  
    staging:  
      target: "http://127.0.0.1:3000"  
      phases:  
        - duration: 20  
          arrivalRate: 10  
scenarios:  
  - flow:  
    - get:  
      url: "/ads/"
```

- **Configuración**

- *phases*

- Permite definir distintas fases, cada una de ellas con una **duración** y **ratio de usuarios** por segundo.

```
config:  
  target: "https://staging.example.com"  
  phases:  
    - duration: 120  
      arrivalRate: 10  
      rampTo: 50  
      name: "Warm up the application"  
    - pause: 60  
    - duration: 600  
      arrivalRate: 50  
      name: "Sustained max load"
```

- **Configuración**

- *payload*

- Con esta opción podemos importar variables desde un CSV y usarlas en las peticiones

- *variables*

- Podemos utilizar en los distintos escenarios las variables que obtengamos en el payload

- *defaults*

- Podemos definir las cabeceras por defecto que utilizarán nuestras peticiones

Artillery – Test Script - Configuración

- Ejemplo *payload/variables*

ejem4

```
config:  
  target: "http://127.0.0.1:3000"  
  phases:  
    - duration: 20  
      arrivalRate: 10  
  payload:  
    path: "ads.csv"  
    fields:  
      - "message"  
      - "author"  
scenarios:  
  - flow:  
    - post:  
      url: "/ads/"  
      json:  
        message: "{{ message }}"  
        author: "{{ author }}"
```

Carga de datos

Uso de datos en las peticiones



- **Configuración**
 - *ensure*
 - Podemos realizar aserciones:
 - Comprobar si un % de las peticiones tiene menos de una latencia determinada.
 - Comprobar si las peticiones no sobrepasan una tasa de error determinada.
 - Es útil cuando integramos las pruebas de carga en un sistema de CI (Integración Continua).
 - El comando “*artillery run*” devolverá distinto de cero si no se cumple alguna de las condiciones.

Artillery – Test Script - Configuración

- Ejemplo *ensure*

```
config:  
  target: "http://127.0.0.1:3000"  
  phases:  
    - duration: 20  
      arrivalRate: 10  
  payload:  
    path: "ads.csv"  
    fields:  
      - "message"  
      - "author"  
  ensure:  
    p95: 2  
    maxErrorRate: 1  
scenarios:  
  - flow:  
    - post:  
      url: "/ads/"  
      json:  
        message: "{{ message }}"  
        author: "{{ author }}"
```

Si p95 es mayor de 2ms, falla

Si más del 1% de los usuarios tienen algún error, falla

- **Escenarios**

- En esta sección se pueden declarar **uno o más escenarios** para los usuarios.
- Los escenarios se conforman de una **secuencia de peticiones** o mensajes enviados por el usuario.
- Cada escenario debe contener un atributo ***flow***, un array de operaciones que realiza el usuario (POST, GET, etc).
- Podemos asignar una probabilidad de que un usuario escoja un escenario u otro a través de la propiedad ***weight***.

Artillery – Test Script - Escenarios

- Ejemplo scenario

ejem6

```
config:  
  target: 'https://www.urjc.es'  
  phases:  
    - duration: 20  
      arrivalRate: 20  
scenarios:  
  - name: "Escenario 1"  
    weight: 1 # 1/8 casos  
    flow:  
      - get:  
        url: "/estudios/grado"  
  - name: "Escenario 2"  
    weight: 2 # 2/8 casos  
    flow:  
      - get:  
        url: "/estudiar-en-la-urjc/pruebas-de-acceso"  
  - name: "Escenario 3"  
    weight: 5 # 5/8 casos  
    flow:  
      - get:  
        url: "/intranet-urjc"
```

Artillery – HTTP

• TLS/SSL

- Artillery rechaza por defecto los certificados que no pueda validar.
- Se puede configurar que no verifique los certificados (generalmente usado en desarrollo cuando son auto generados e inseguros)

```
config:  
  target: "https://items.staging:8443"  
  tls:  
    rejectUnauthorized: false
```

Artillery – HTTP

- **Request timeout**
- Latencia máxima de las peticiones para considerarlas fallidas.

```
config:  
  target: "http://localhost:8080"  
  http:  
    timeout: 10
```

Definimos un timeout de 10 segundos



Artillery – HTTP

- **Flow actions**
- Configuración de las acciones flow:
 - **url** → Completa la URI proporcionada en el target
 - **json** → Podemos incluir el cuerpo de la petición en JSON
 - **body** → Datos enviados en el cuerpo de la petición
 - **headers** → Cabeceras de la petición en formato JSON
 - **cookie** → Cookies de la petición en formato JSON
 - **capture** → Nos permite capturar valores de la respuesta para realizar nuevas peticiones.

Artillery - HTTP

ejem7

```
config:  
  target: "http://127.0.0.1:3000"  
  phases:  
    - duration: 20  
      arrivalRate: 10  
  payload:  
    path: "ads.csv"  
    fields:  
      - "message"  
      - "author"  
scenarios:  
  - flow:  
    - post:  
      url: "/ads/"  
      json:  
        message: "{{ message }}"  
        author: "{{ author }}"  
      capture:  
        json: "$.id"  
        as: "id"  
    - get:  
      url: "/ads/{{ id }}"
```

Capturamos la variable **id** de la respuesta y la usamos para **encadenar** una nueva petición

Artillery - HTTP

• Basic Auth

ejem8

```
config:
  target: "https://127.0.0.1:3443"
  tls:
    rejectUnauthorized: false
  phases:
    - duration: 10
      arrivalRate: 5
scenarios:
  - flow:
    - get:
      url: "/ads/"
      auth:
        user: "admin"
        pass: "pass"
```

Artillery – Distribuido

- Artillery permite utilizar un **cluster de nodos** para poder repartir a los usuarios entre distintas máquinas.
- Solo está disponible en la versión PRO (comercial)
- Despliega a los usuarios ficticios en **Amazon Web Service**
- Ejemplo de ejecución sobre un cluster de EC2

```
$ artillery run-cluster --cluster my-ecs-cluster --region us-east-1 --count 1 hello-world.yml
```