Pruebas de rendimiento con JMeter. Ejemplos básicos

02/07/2018

JUAN FRANCISCO SÁNCHEZ

QA

#qa #jmeter #rendimiento #pruebas

En todo proyecto de desarrollo software es muy importante la realización de planes de pruebas que garanticen que se cumplen los requisitos funcionales de la aplicación. Pero no todo el trabajo de un QA son las pruebas funcionales. Además de éstas, existen otras que pueden marcar la calidad de una aplicación: de seguridad, de usabilidad, de rendimiento.

Las pruebas de rendimiento son, desde la perspectiva más evidente, las que se realizan para determinar lo rápido que un sistema realiza una tarea en unas determinadas condiciones de trabajo. En este post hablaremos sobre su utilidad, los tipos que existen y cómo realizarlas con JMeter.

¿Para qué sirven las pruebas de rendimiento?

Para alcanzar un buen nivel de rendimiento de un sistema es fundamental que las pruebas comiencen en el inicio del desarrollo del software. Al igual que en las pruebas funcionales, el coste de solucionar defectos se ve aumentado conforme más se tarde en detectarlos.

Además, si queremos que los resultados sean lo más fiables posible, nuestro entorno de pruebas debe ser lo más parecido posible al de producción, y no cruzarlo nunca con el de desarrollo ni el de otras pruebas.

La pruebas de rendimiento sirven, entre otras cosas, para:

fiabilidad, uso de los recursos.
- Comparar dos sistemas para saber cuál de ellos funciona mejor.
- Medir qué partes del sistema o de carga de trabajo provocan que el conjunto rinda mal.

Tipos de prueba de rendimiento

PRUEBA DE CARGA

Éste es el tipo más sencillo de pruebas de rendimiento. Una prueba de carga se realiza generalmente para observar el comportamiento de una aplicación bajo una cantidad esperada de peticiones. Esta carga puede ser el número esperado de usuarios concurrentes, utilizando la aplicación que realizan un número específico de transacciones, durante el tiempo que dura la carga. Esta prueba puede mostrar los **tiempos de respuesta** de todas las transacciones importantes de la aplicación. Si también se monitorizan otros aspectos como la base de datos, el servidor de aplicaciones, etc., entonces esta prueba puede mostrar el cuello de botella en la aplicación.

PRUEBA DE ESTRÉS

Se utiliza normalmente para romper la aplicación. Se va doblando el número de usuarios que se agregan a la aplicación y se ejecuta una

Esto ayuda a los administradores para determinar si la aplicación rendirá lo suficiente en caso de que la carga real supere a la carga esperada.

PRUEBA DE ESTABILIDAD (SOAK TESTING)

Normalmente se hace para determinar si la aplicación puede **aguantar una carga esperada continuada**. Generalmente esta prueba se realiza para determinar si hay alguna fuga de memoria en la aplicación.

PRUEBA DE PICO (SPIKE TESSTING)

La prueba de picos, como el nombre sugiere, trata de observar el comportamiento del sistema variando el número de usuarios, tanto cuando bajan como cuando tiene cambios drásticos en su carga. Esta prueba se recomienda que sea realizada con un software automatizado que permita realizar cambios en el número de usuarios mientras que los administradores llevan un registro de los valores a ser monitorizados.

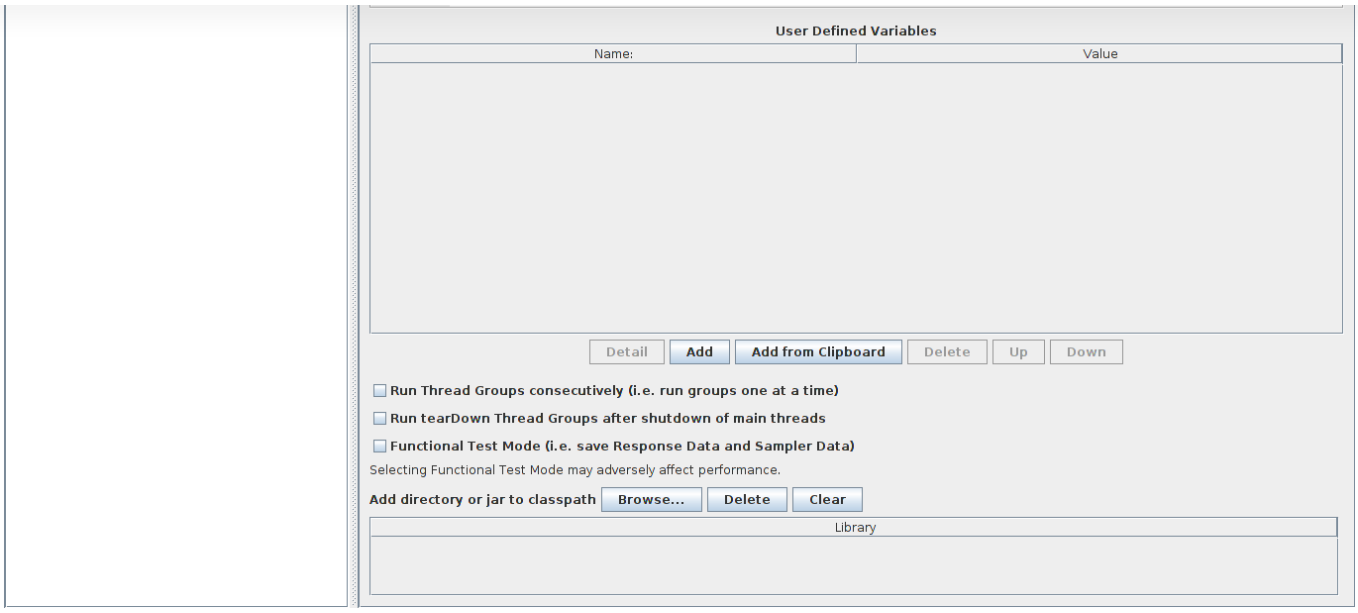
Metodología

1. **Identificar el entorno de pruebas**. Identificar el entorno físico de pruebas y el entorno de producción, así como las herramientas y recursos de que dispone el equipo de prueba. El entorno físico incluye hardware, software y configuraciones de red. Tener desde el principio un profundo conocimiento de todo el entorno de prueba permite diseños de pruebas más eficientes. Facilita también la planificación y ayuda a identificar problemas en las pruebas en fases tempranas del proyecto. En algunas situaciones, este proceso debe ser revisado periódicamente durante todo el ciclo de vida del proyecto.
2. **Identificar los criterios de aceptación de rendimiento**. Determinar el tiempo de respuesta, el rendimiento, la utilización de los recursos y los objetivos y limitaciones. En general, el tiempo de respuesta concierne al usuario, el rendimiento al negocio, y la utilización de los recursos al sistema. Identificar cuáles serían criterios de éxito de rendimiento del proyecto para evaluar qué combinación de la configuración da lugar a un funcionamiento óptimo.
3. **Planificar y diseñar las pruebas**. Identificar los principales escenarios, determinar la variabilidad de los usuarios y la forma de simular esa variabilidad, definir los datos de las pruebas y establecer las métricas a recoger. Consolidar esta información en uno o más modelos de uso del sistema a implantar, ejecutarlo y analizarlo.
4. **Configurar el entorno de prueba**. Preparar el entorno de prueba, las herramientas y recursos necesarios para ejecutar cada una de las estrategias, así como las características y componentes disponibles para la prueba. Asegurarse de que el entorno de prueba se ha preparado para la monitorización de los recursos según sea

6. **Ejecutar la prueba**. Ejecutar y monitorizar las pruebas. Validar las pruebas, los datos de las pruebas y recoger los resultados. Ejecutar pruebas válidas para analizar, mientras se monitoriza la prueba y su entorno.
7. **Analizar los resultados, realizar un informe y repetirlo**. Consolidar y compartir los resultados de la prueba. Analizar los datos, tanto individualmente como con un equipo multidisciplinario. Volver a priorizar el resto de las pruebas y a ejecutarlas en caso de ser necesario. Cuando todas las métricas estén dentro de los límites aceptados, ninguno de los umbrales establecidos hayan sido rebasados y toda la información deseada se ha reunido, las pruebas han acabado para el escenario definido por la configuración.

JMeter

Existen numerosas herramientas, tanto open source como privativas, para realizar las pruebas de rendimiento: NeoLoad, LoadRunner, LoadUI, WebLOAD, etc. Una de las más populares es **JMeter (open source)**, en la cual nos centraremos en este post.



En JMeter, un plan de pruebas es una jerarquía de componentes en forma de árbol (panel de control de la izquierda). Cada nodo del árbol es un componente. A su vez, un componente es una instancia de un tipo de componente en la que quizás se han configurado algunas de sus propiedades (en el panel de control de la derecha).

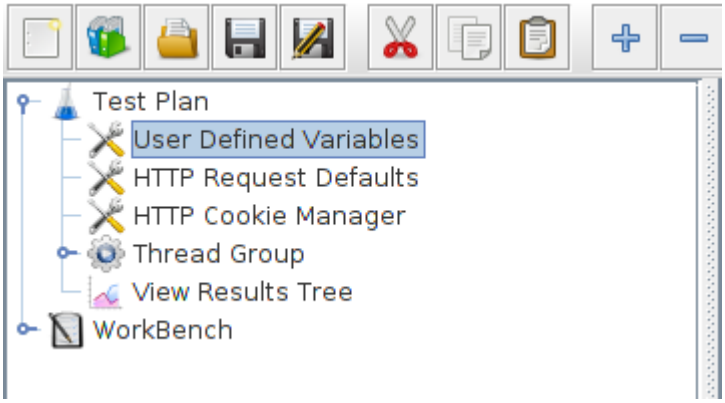
Los diferentes **componentes** de los que puede constar un plan de pruebas son:

- **Test Plan**. Es el tipo de componente que representa la raíz del árbol.
- **Thread Group**. Representa un grupo de usuarios. En JMeter cada thread es un usuario virtual.
- **Controllers (Sampler, Logic Controler)**. Los *samplers* realizan peticiones contra la aplicación y los *logic controlers* establecen el orden en que se ejecutan éstos.
- **Config Element**. Establecen propiedades de configuración que se aplican a los samplers a los que afectan.
- **Assertion**. Comprueban condiciones que aplican a las peticiones

- **Timer**. Añaden tiempo extra a la ejecución de las peticiones que realizan contra la aplicación los samplers a los que afectan
- **Pre-Processor element**. Realizan acciones o establecen configuraciones previa a la ejecución de los samplers a los que afectan.
- **Post-Processor element**. Realizan acciones o establecen configuraciones posteriormente a la ejecución de los samplers a los que afectan.

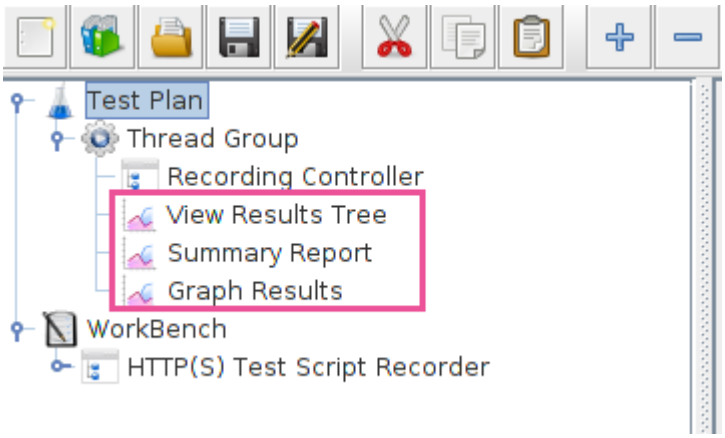
Ejemplo básico con JMeter

Para preparar el escenario de las pruebas en JMeter **cargamos una plantilla**. Vamos a File>Template. Se abre una pequeña ventana nueva para elegir qué plantilla deseamos cargar. Elegimos la primera ('*Recording*'). Ésta sirve para □grabar□ la navegación de un sitio web.

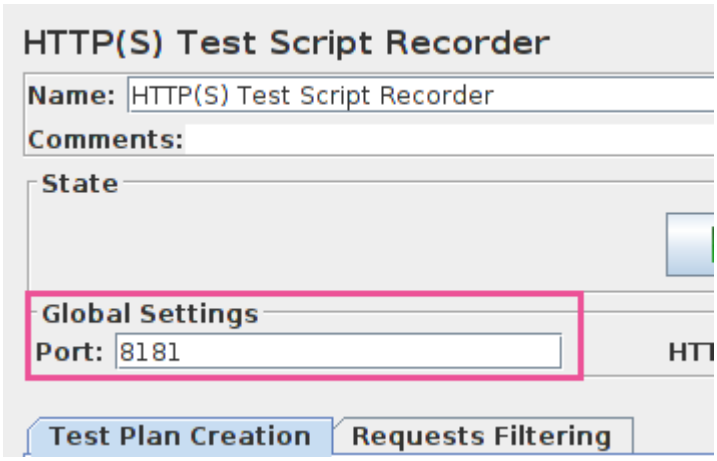


Para nuestro ejemplo básico, podemos eliminar algunos componentes como '*User Defined Variables*', '*HTTP Request Default*' y '*HTTP Cookie*

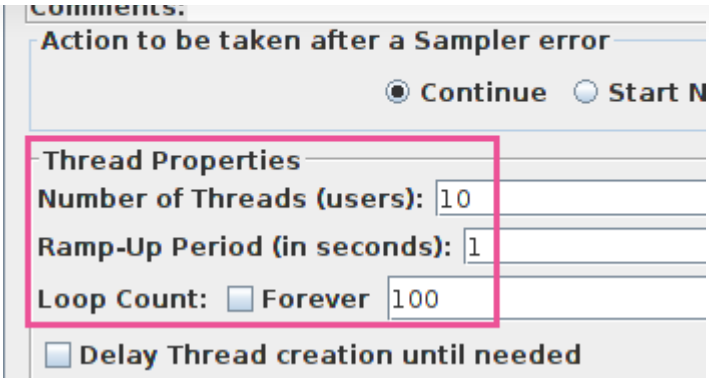
receptores de datos más. Nosotros, finalmente trabajamos con este esquema:



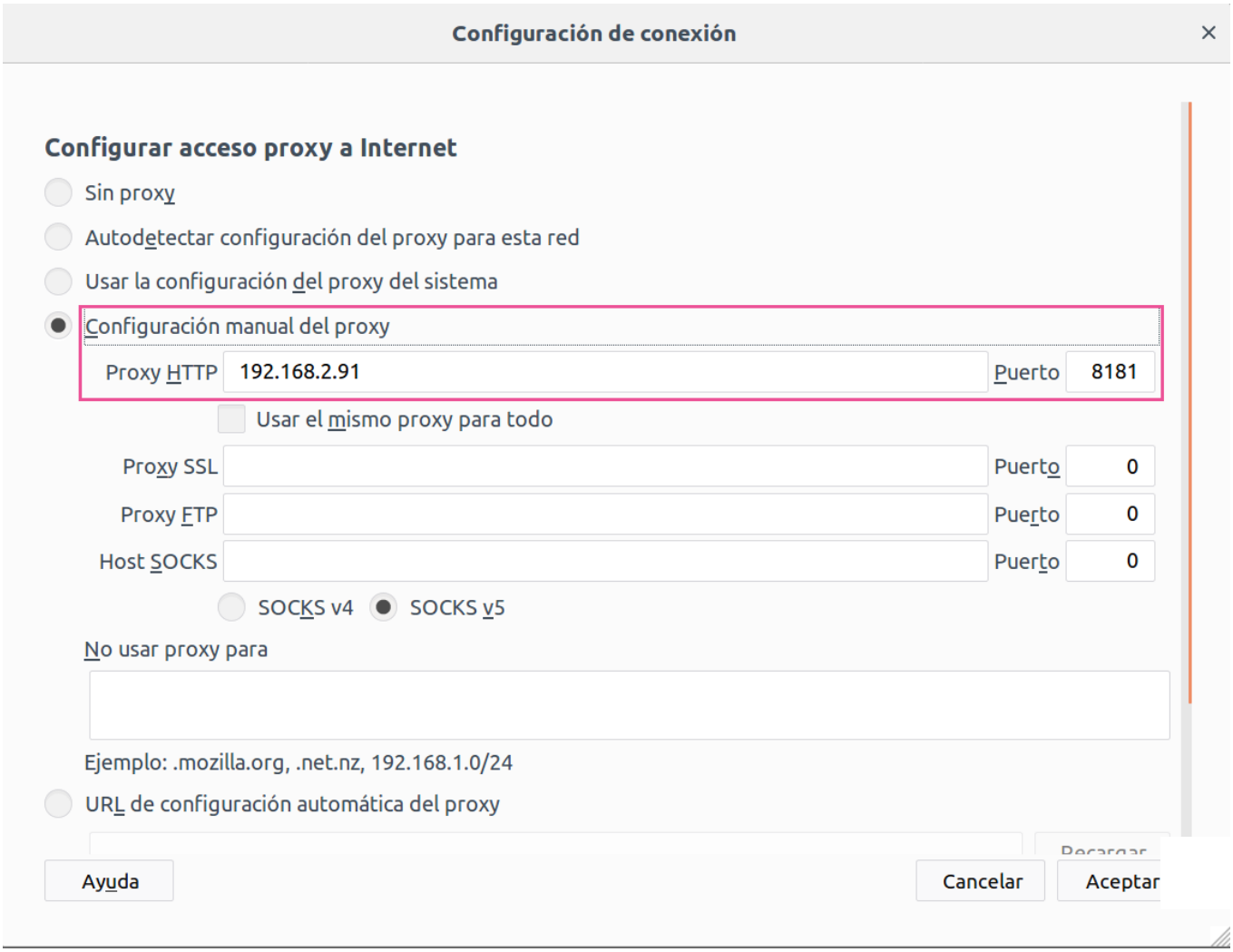
Para la **configuración**, vamos a '*HTTP(S) Test Script Recorder*' y en el campo del puerto escribimos el que queramos usar. En nuestro caso, 8181.



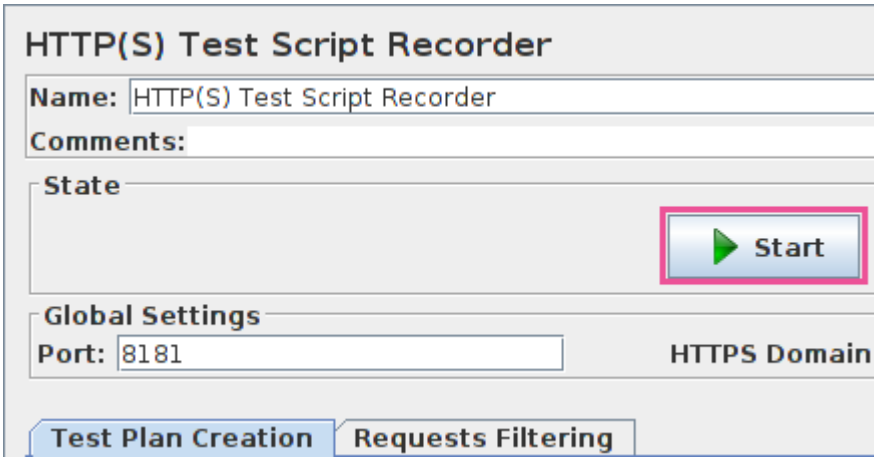
Pulsamos, ahora, en '*Thread Group*' y en '*Number of Threads*' escribimos el **número de usuarios** concurrentes que queremos que hagan las peticiones. En nuestro caso, 10. En '*Loop Count*' (repeticiones) pondremos 100. Ya tendríamos configurado correctamente JMeter.



Antes de empezar a grabar, debemos configurar el proxy de nuestro navegador con el puerto que escribimos anteriormente y la dirección IP de nuestro equipo:

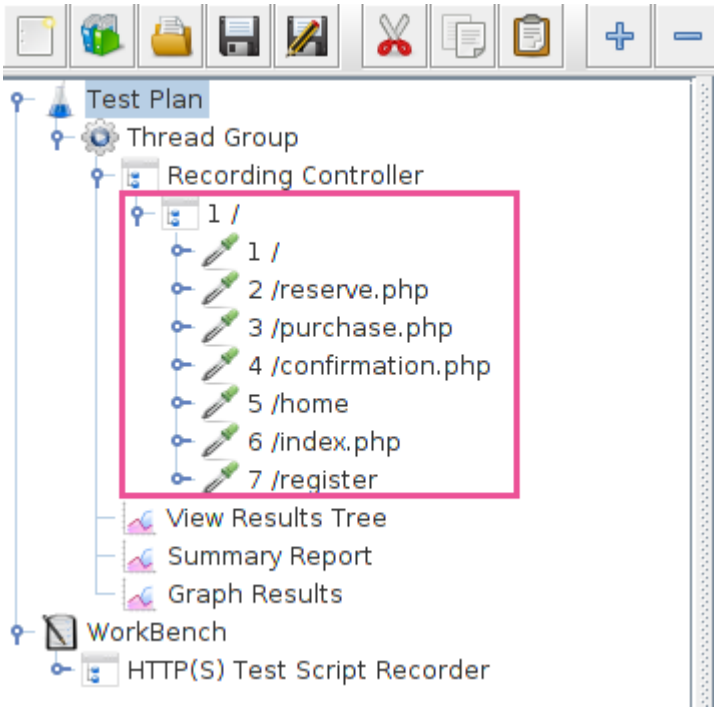


Test Script Recorder' y hacemos click en '*Start*'. Ya estamos 'escuchando'.

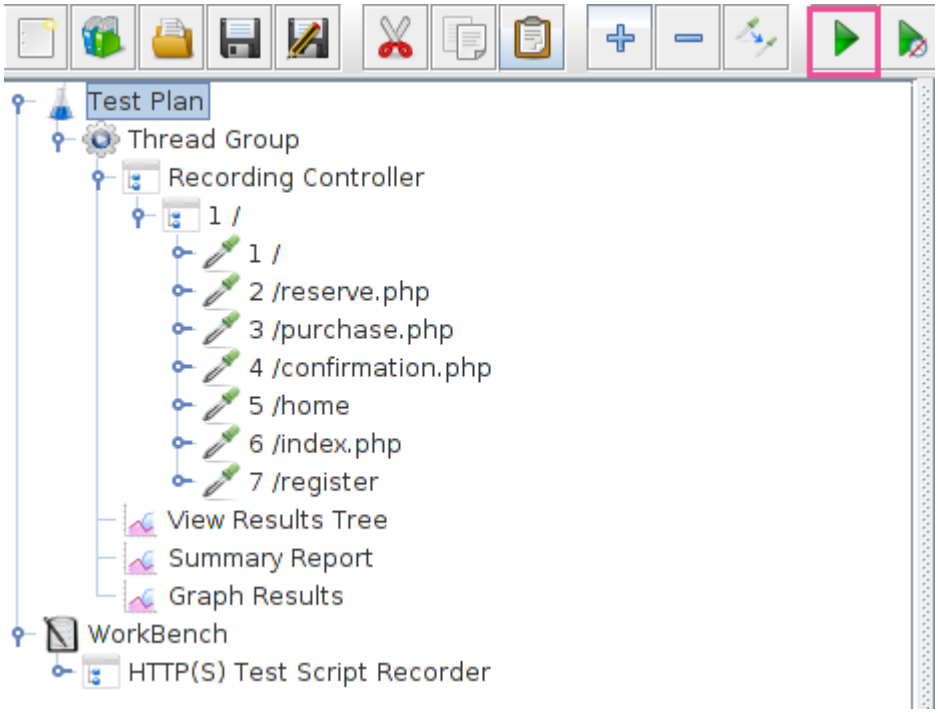


Ya solo nos queda navegar a través de la web que queramos probar. Para nuestro ejemplo, hemos navegado en una página de prueba: Blazedemo.

Las peticiones se van guardando en nuestro **Test Plan**:



hay que pulsar el botón de '*Run*'.



Los **resultados** pueden observarse en los '*Listeners*' que añadimos anteriormente:

Summary Report

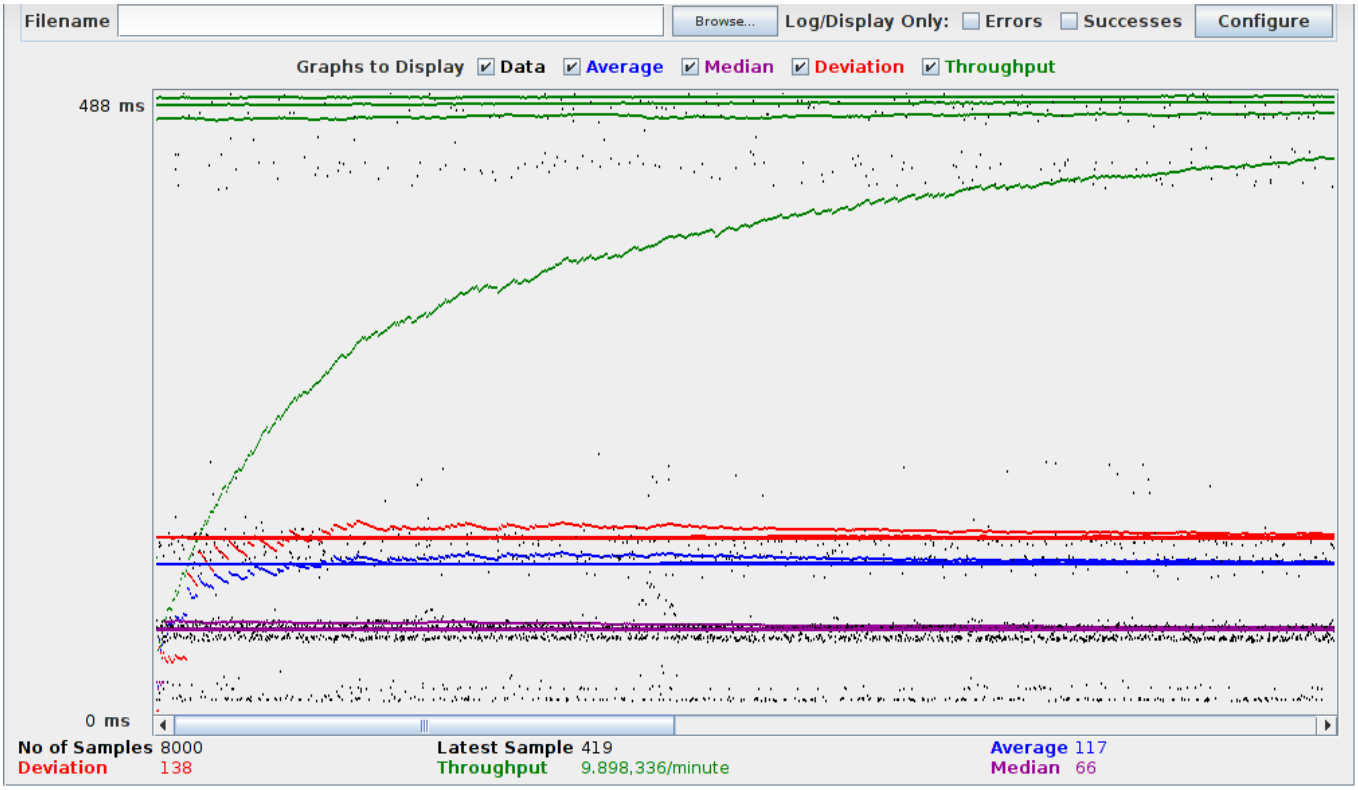
Name: Summary Report

Comments:

Write results to file / Read from file

Filename Browse... **Log/Display Only:** Errors Successes **Configure**

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB...	Sent KB/sec	Avg. Bytes
1 /	2000	242	9	650	230,94	0,00%	41,2/sec	1343,39	79,81	33354,2
2 /reserve.p...	1000	65	56	141	13,50	0,00%	20,8/sec	216,40	9,64	10658,9
3 /purchase...	1000	65	56	132	10,49	0,00%	20,8/sec	202,78	10,84	9973,0
4 /confirmat...	1000	64	56	133	9,48	0,00%	20,8/sec	182,97	12,16	8995,9
5 /home	1000	133	114	232	17,71	0,00%	20,8/sec	335,88	15,18	16533,3
6 /index.php	1000	63	56	184	10,20	0,00%	20,8/sec	161,82	7,37	7949,0
7 /register	1000	66	57	129	10,89	0,00%	20,8/sec	224,42	7,45	11022,1
TOTAL	8000	117	9	650	138,14	0,00%	165,0/sec	2655,04	141,85	16480,1



Ejemplo básico con JMeter + Maven

Para usar JMeter dentro de un proyecto Maven, vamos a utilizar el **plugin jmeter-maven-plugin**. La configuración de éste en el archivo POM es la siguiente:

```
<plugins>
        <plugin>
                <groupId>com.lazerycode.jmeter</groupId>
                <artifactId>jmeter-maven-plugin</artifactId>
                <version>2.7.0</version>
                <executions>
```

```
                                </goals>
                                <phase>verify</phase>
                        </execution>
                </executions>
                <configuration>

                        <generateReports>false</generateReports>
                        <resultsFileFormat>xml</resultsFileFormat>
                        <testResultsTimestamp>false</testResultsTimestamp>

                </configuration>
        </plugin>
</plugins>
```

Por defecto, JMeter crea un reporte a través de un archivo .csv. Para que esto no ocurra tenemos que poner a false el *generateReports*. El siguiente plugin que vamos a utilizar necesita leer los datos de un archivo .xml, de ahí la configuración de *resultFileFormat*. El otro plugin que usamos es **jmeter-graph-maven-plugin** que nos mostrará los distintos resultados en forma de gráficas. En el archivo POM indicamos cuál será el archivo de entrada y los de salida que corresponderá a cada una de las gráficas que añadamos:

```
<plugin>
        <groupId>de.codecentric</groupId>
        <artifactId>jmeter-graph-maven-plugin</artifactId>
        <version>0.1.0</version>
        <executions>
                <execution>
                        <id>create-graphs-collection</id>
                        <goals>
                                <goal>create-graph</goal>
                        </goals>
```

```
<graph>
        <pluginType>ThroughputVsThreads</plu
        <width>1920</width>
        <height>1080</height>
        <outputFile>${project.build.directory
</graph>
<graph>
        <pluginType>ResponseCodesPerSecond</
        <width>1920</width>
        <height>1080</height>
        <outputFile>${project.build.directory
</graph>
```

Las distintas gráficas y su significado pueden verse en jmeter-plugins.org. Para ejecutar nuestro plan de pruebas de JMeter, debemos guardarlo dentro de **/src/test/jmeter** del proyecto Maven. Desde la terminal, accedemos a la raíz del proyecto Maven y ejecutamos: **$mvn verify**. Una vez ejecutado, aparecerá algo como:

Y las gráficas aparecerán donde indicamos en la configuración del plugin. Estos son algunos ejemplos:

- **Bytes throughput over time**. Cantidad de bytes enviados y recibidos por JMeter durante la prueba de carga.

- **Server hits per second**. Hits enviados por el plan de prueba al servidor por segundo.
  Rendimiento JMeter - Server hits per second - SDOS
- **Response codes per second**. Código de respuesta por segundo devuelto durante la prueba.

El código de este proyecto lo puedes encontrar en nuestro repositorio.

En resumen

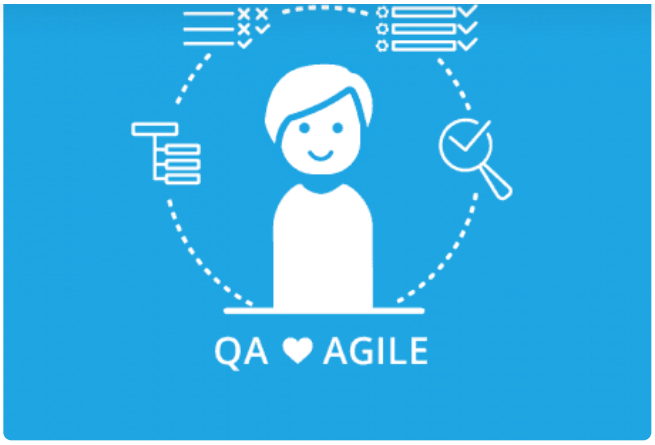
Está claro que las pruebas funcionales son las más importantes, ya que son las que garantizan que la aplicación realizará las funciones marcadas por el cliente. Pero asegurar la calidad de una aplicación es mucho más que eso.

Por esto, QA y desarrolladores tienen que colaborar estrechamente no sólo para que la aplicación haga esta cosa o la otra, sino también para que lo haga en el mínimo de tiempo posible. Ahí puede estar la diferencia entre una buena aplicación y otra mejor, entre una buena y mejor satisfacción del cliente.

En definitiva, no sólo de pruebas funcionales vive el QA.

POSTS RELACIONADOS

Cómo crear desplegar un artefacto con Maven



¿Qué papel tiene QA en las metodologías ágiles?

COMENTARIOS

Victor 30 Jul 2018

Muy didáctico. Normalmente, la gente confunde pruebas de carga con pruebas de estrés, y no terminan de ver claro la diferencia de concepto con respecto a unas pruebas funcionales, por la dificultad añadida de la concurrencia, entre otras cosas. Quien vaya a hacer unas pruebas de rendimiento, tiene que tener claro que no son pruebas funcionales, pero

una secuencia de ejecuciones representativa sin necesidad de validar el rendimiento de todas las funcionalidades. Disponer de este conocimiento funcional, tambien ayuda a identificar errores funcionales en la generación de los scripts de carga y en la ejecución de las pruebas.
Por mi experiencia realizando pruebas de rendimiento, también diría que es conveniente planificarse desde la ejecución del caso más sencillo probando directamente el componente en el que se intuya que pueda haber un cuello de botella, y extender las pruebas al resto del sistema según se certifica el rendimiento de componentes más sencillos.
También es muy importante poder probar en un entorno estable o aislado en el que no interfieran ciclos de pruebas y desarrollos correctivos o evolutivos, o cualquier otra actualización de datos o configuraciones parametricas, ya que esto desconcierta a los técnicos ante la aparición de errores y puede suponer pérdidas de días enteros hasta que se detecta el origen del error.
Finalmete, comentar que, si bien el caso ideal las pruebas se deberían hacer en entornos productivos, en la práctica real esto resulta prácticamente imposible. La siguiente aproximación, que sería probar en un entorno preproductivo y escalar a un entorno de producción, también será raramente practicable si no es que se ha planteado un sistema escalable desde el principio. Aquí, lo que nos encontraremos en la mayoría de ocasiones son sistemas preexistentes en los que no podremos probar en entornos productivos, y en los que no será posible obtener un ratio de mejora sobre las pruebas que hagamos en entornos o reproductivos. Lo que podemos hacer en estas situaciones es aprovechar las pruebas en entornos reproductivos para identificar cuellos de botella que seguramente se reproducirán en entornos productivos, y tratar de implicar al propietario de la aplicación para

habitual, hasta una ejecución de pruebas con una carga medida en horarios de poca afluencia para evitar posibles daños colaterales.

DEJA TU COMENTARIO

Hablemos

Si tienes una idea o un proyecto, nos encantará conocerte.

CONTACTO

SDOS Tecnología que se siente

Aviso legal

Calidad y Medio Ambiente