Universidad Autónoma de Madrid

Departamento de Informática

Sistemas Informáticos II
Práctica - 2

Roberto Marabini

Registro de Cambios

Versión[1]	Fecha	Autor	Descripción
1.0	1.3.2025	RM	Primera versión.

[1] La asignación de versiones se realizan mediante 2 números $X.Y$. Cambios en Y indican aclaraciones, descripciones más detalladas de algún punto o traducciones. Cambios en X indican modificaciones más profundas que o bien varían el material suministrado o el contenido de la práctica.

Índice

1. Objetivos

En el desarrollo de aplicaciones web, la medición del rendimiento es un aspecto fundamental para garantizar una experiencia óptima a los usuarios. *JMeter* es una herramienta de código abierto diseñada para realizar pruebas de rendimiento y carga en aplicaciones web. Permite simular múltiples usuarios concurrentes y evaluar cómo responde el servidor ante distintas condiciones de tráfico. A través de métricas como el tiempo de respuesta, la tasa de errores y el rendimiento bajo carga, *JMeter* proporciona información para la optimización del sistema.

Esta práctica tiene como objetivo analizar el rendimiento de las diferentes aplicaciones web desarrolladas en la asignatura mediante la configuración y ejecución de pruebas con *JMeter*. Se explorarán los conceptos clave de pruebas de carga, la configuración del servidor **gunicorn** y el análisis de los resultados obtenidos.

2. Algunos parametros usados para optimizar el servidor web gunicorn

En un entorno *Django* que utiliza **gunicorn** como servidor WSGI, el rendimiento suele ajustarse modificando tanto el número de *workers* como el número de *threads*. Los *workers* representan procesos independientes que manejan solicitudes, mientras que los *threads* permiten manejar múltiples solicitudes concurrentes dentro de cada *worker*.

Estos parámetros pueden configurarse al iniciar **gunicorn** con las opciones `-workers` y `-threads`. Aumentar el número de *workers* mejora la capacidad del servidor para manejar múltiples solicitudes simultáneamente, pero también incrementa el uso de memoria, ya que cada *worker* tiene su propio espacio de memoria. Por otro lado, aumentar los *threads* dentro de cada *worker* puede mejorar la eficiencia en aplicaciones con tareas de I/O intensivo (p.e. acceso a bases de datos), ya que los *threads* dentro de un mismo *worker* comparten memoria y recursos, reduciendo el consumo global. Sin embargo, en aplicaciones ligadas a la CPU, un número elevado de *threads* puede generar bloqueos y reducir el rendimiento.

Nuestra aplicación usa una variable de sesión para almacenar el identificador del censo (`numeroDNI`) tras verificar la existencia del votante en el censo. Por defecto, nuestra aplicación usa la política `SESSION_ENGINE=django.contrib.sessions.backends.cache` como backend de sesiones (esta variable se define en settings.py). Este método permite acceder rápidamente a los datos almacenados de sesión puesto que se almacenan en RAM, mejorando los tiempos de respuesta. Sin embargo, este método no es persistente y las sesiones se pierden si el servidor se reinicia o no se utiliza el mismo "worker" en la llamada que consulta los datos del censo y en la llamada que realiza el voto. En cambio, almacenar las sesiones en la base de datos (`SESSION_ENGINE=django.contrib.sessions.backends.db`) garantiza persistencia, pero introduce latencia adicional debido a las consultas necesarias para recuperar la información de cada sesión. Cuál de estas dos opciones es más conveniente depende de la aplicación. En una aplicación como la nuestra, que usa variables de sesión, se debe seleccionar la segunda opción si se usa más de un "worker".

3. Medición de rendimiento usando *JMeter*

A continuación describiremos en detalle como usar *JMeter* para medir el rendimiento de nuestra aplicación `P1-base`. Se asume:

- el alumno cuenta con la versión "5.6.3" de *JMeter* y este está configurado en inglés.
- el servidor se ejecuta en el puerto 8000 de la maquina virtual vm2.
- la base de datos se encuentra en vm1.
- *JMeter* se ejecuta desde la máquina host.

3.1. Ejecución de una Prueba en *JMeter* para P1-base

Para evaluar la capacidad de la aplicación **P1-base**, se ejecutará una prueba de carga en *JMeter* donde el número de peticiones aumentará progresivamente hasta que la aplicación falle. La prueba simulará la interacción de los usuarios con los dos

formularios de la aplicación: **censo** y **voto**. Cada usuario seguirá el siguiente flujo de solicitudes:

1. Solicitud **GET** para obtener el formulario de **censo**.
2. Enviar los datos completados mediante una solicitud **POST**. (La aplicación almacena el identificador de censo en una variable de sesión). Nótese que esta llamada POST devuelve una cookie con el identificador de sesión que debe devolverse en las llamadas siguientes puesto que sirve para identificar la variable de sesión en la que se ha almacenado el identificador de censo.
3. Solicitud **GET** para obtener el formulario de **voto**.
4. Enviar los datos completados mediante una solicitud **POST**.

Al terminar el proceso de configuración el interfaz de *JMeter* debe ser similar al mostrado en la figura 1

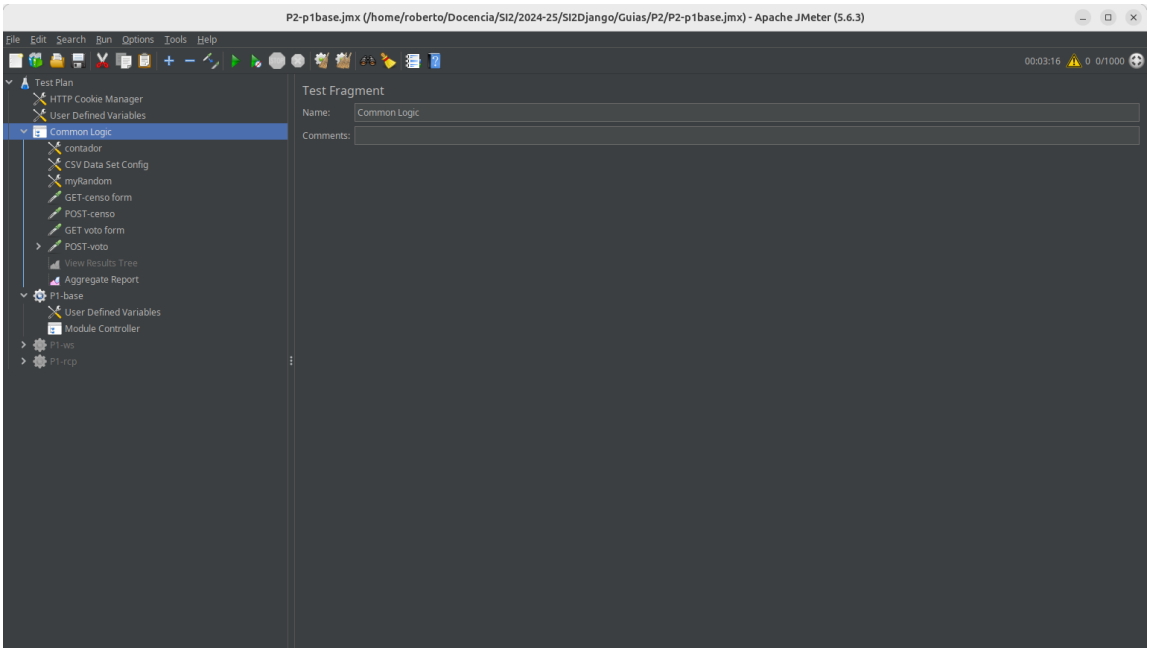

Figura 1: *JMeter* mostrando el "plan de prueba" completado.

A continuación, se describen los pasos para configurar y ejecutar el plan de prueba en *JMeter*.

Crear plan de pruebas

Al abrir *JMeter* este muestra un plan de pruebas (Test Plan) vacío tal y como se aprecia en la figura 2

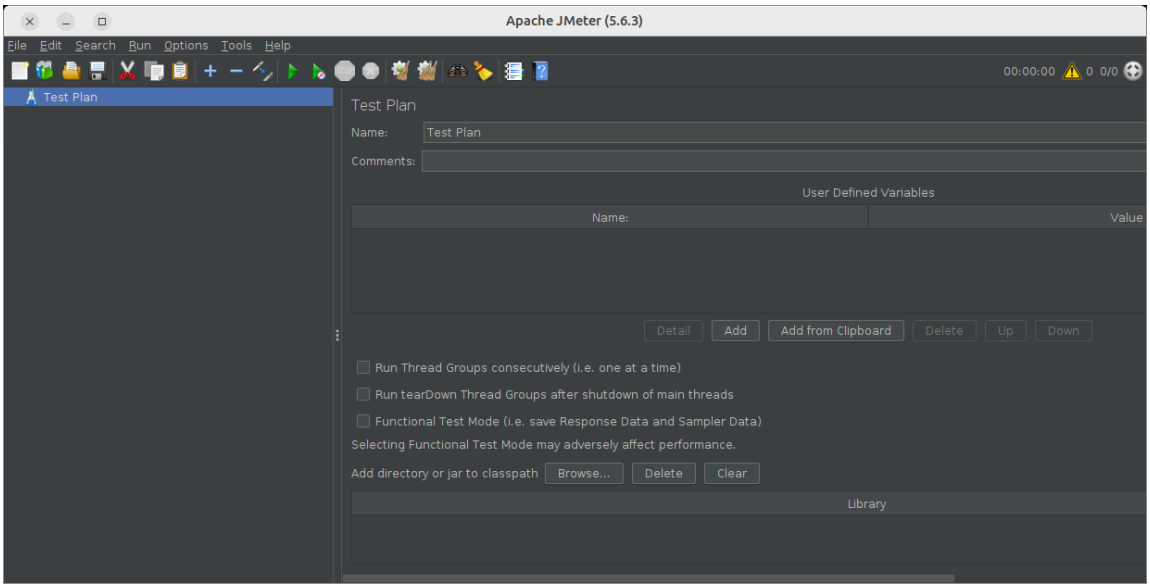

Figura 2: *JMeter* con un "plan de prueba" recien creado.

Crear "test fragment"

En las prácticas P1A y P1B se han desarrollado tres proyectos que implementan la misma funcionalidad. Podríamos crear 3 test independientes, uno por proyecto, pero repetiríamos mucho código. *JMeter* ofrece la posibilidad de crear "test fragments" donde se puede almacenar el código que sea común a los diferentes test y llamarlo como si fuera una función. Para crear un "test fragment" presiona el botón derecho del ratón sobre "Test Plan" $\rightarrow$ "Add" $\rightarrow$ "Test Fragment". En la figura 1 el "test fragment" se ha etiquetado "Common Logic".

Crear grupos de hilos

Para cada caso a "testear" crearemos un Grupo de Hilos que representan conjuntos de usuarios que se conectan concurrentemente a la applicación.

- Hacer clic derecho sobre "Test Plan".
- Navegar a `Add → Threads (Users) → Thread Group`.
- Asignar un valor descriptivo al campo "Name" por ejemplo si vamos a probar el proyecto P1-base podemós llamar a este "'conjunto de hilos" P1-base
- Configurar el número de threads (Usuarios). En la primera prueba podéis seleccionar 1 para posteriormente ir incrementando este valor.
- Establecer un **Ramp-Up Period** para incrementar gradualmente la carga (por ejemplo, 2 segundos).
- Finalmente en "Loop count" pondremos el valor `${samples}`. Por ello, repetiremos el experimento "sample" veces, de forma que el numero total de muestras (votos emitidos) será el almacenado en la variable llamada "samples" que todavía no hemos definido.

En la figura 3 se muestra el formulario usado para crear un grupo de hilos.

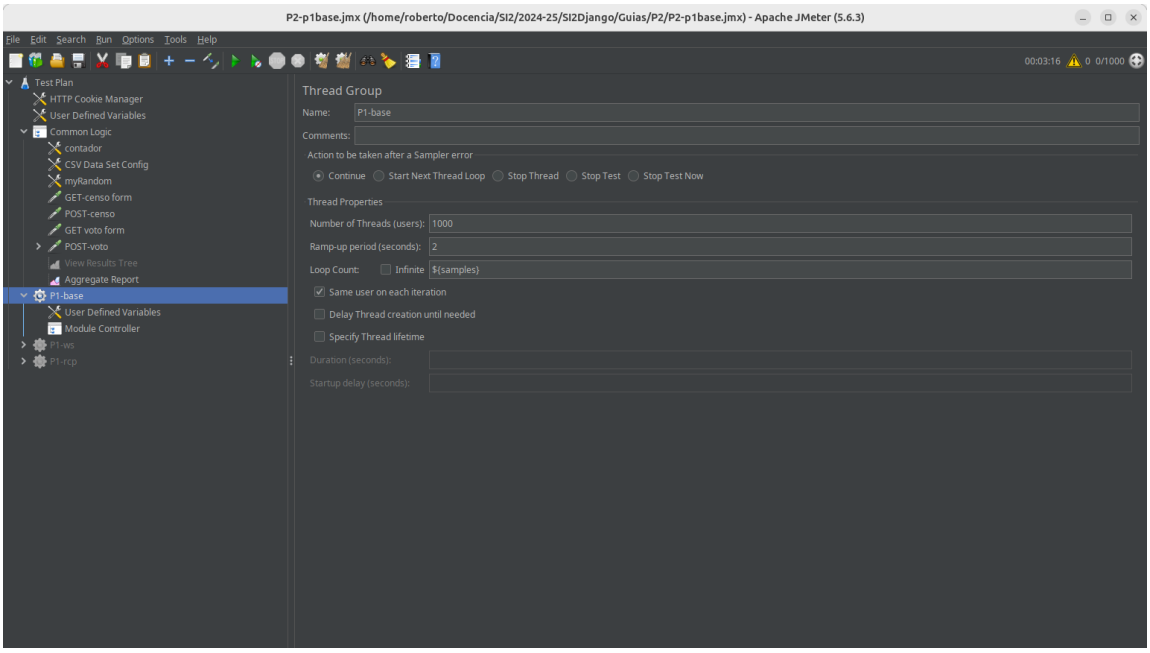

Figura 3: Creación del "Grupos de Hilos" "P1-base" en *JMeter*.

Manejo de cookies

Procederemos a añadir un gestor de cookies que se encargará de gestionar entre otras cookies la que contiene el identificador de sesión. En los navegadores todas las cookies enviadas por el servidor son devueltas al mismo en consultas posteriores de forma transparente al usuario. Por el contrario *JMeter* ignora las cookies a menos que instalemos el susodicho gestor de cookies.

El **HTTP Cookie Manager** en *JMeter* se utiliza para manejar las cookies de las peticiones HTTP automáticamente. Sigue estos pasos para agregarlo:

- Hacer clic derecho sobre el elemento "Test Plan".
- Navegar a `Add → Config Element → HTTP Cookie Manager`.
- Los valores por defecto deberían funcionar correctamente.

En la figura 4 se muestra el formulario usado para crear el "cookie manager".

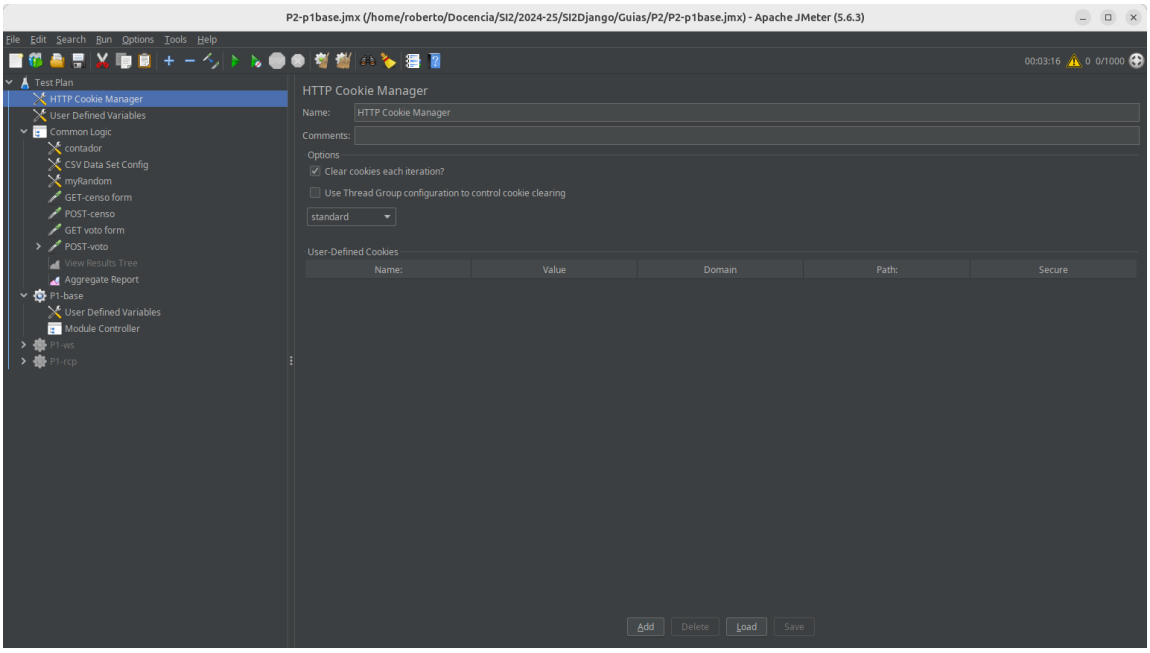

Figura 4: Creación de un "Cookie Manager" en *JMeter*.

Creación de variables de usuario

El elemento **User Defined Variables** en JMeter permite definir variables personalizadas que pueden ser reutilizadas en diferentes elementos del plan de prueba. Si la variable se llama `sample` la forma de introducirla en un formulario de creación de otro elemento sería `${sample}`. Esto es, el nombre de la variable se rodea con unas llaves y se precede con el caracter "$". Los beneficios de usar variables de usuario son:

- Permite reutilizar valores en diferentes partes del test.
- Facilita cambios globales en la configuración sin modificar múltiples elementos.
- Mejora la legibilidad y mantenimiento del plan de pruebas.

Algunas variables serán globales (válidas en todos los test) y otras locales (válidas en un conjunto de hilos). Para definirlas basta con

1. Hacer clic derecho sobre el "Test Plan"(variables globales) o el "Thread Group"(variables locales) donde deseas agregar las variables.
2. Navegar a `Add → Config Element → User Defined Variables`.

En las figuras 5 y 6 se muestran las valores a introducir. Su significado es el siguiente:

- **sample** numero de muestras (este valor debe variar en las pruebas)
- **candidado_X** lista con el nombre de tres posibles candidatos
- **debug** variable auxiliar que puede resultar útil para modificar temporalmente la entrada de un elemento del test.
- **port** puerto en el que se ejecuta *Django*
- **entrypoint** parte del URL que varía entre los diferentes proyectos. Este valor será distinto en cada "conjunto de hilos".
- **host** ordenador en el que se ejecuta *Django*

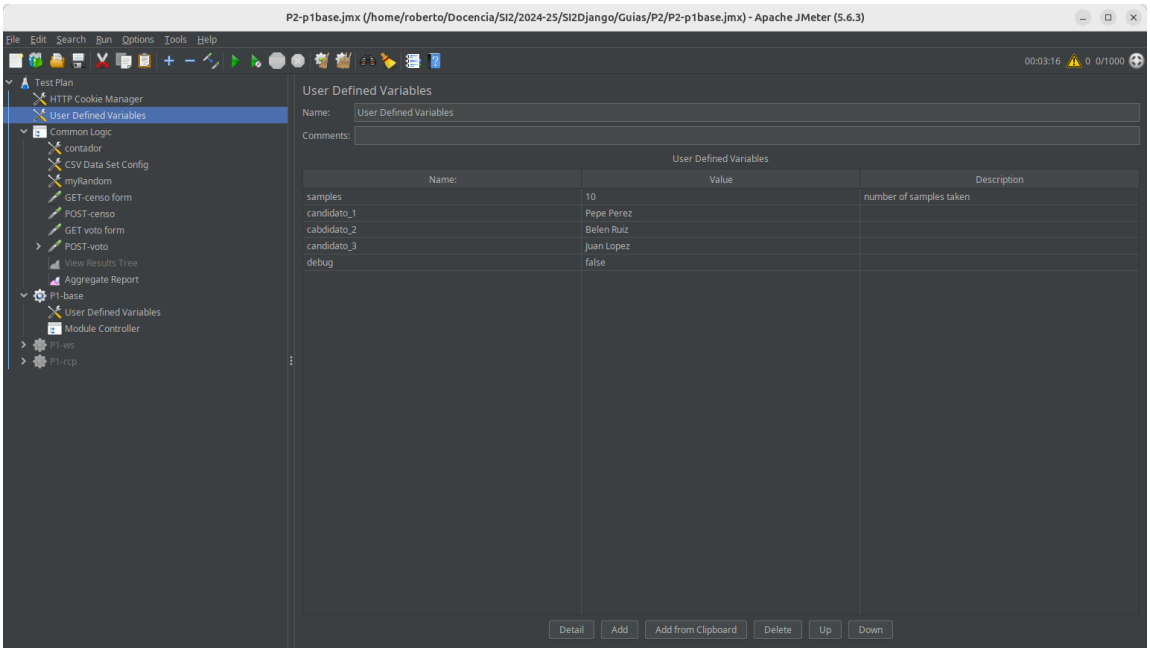

Figura 5: Creación de variables de usuario globales en *JMeter*.

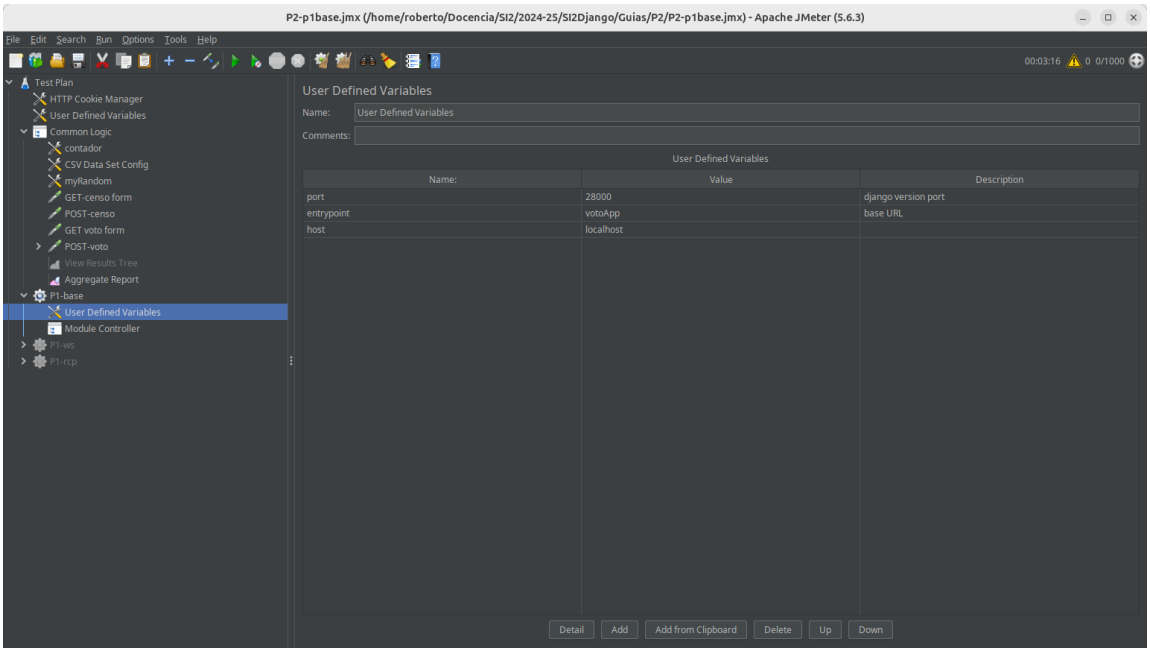

Figura 6: Creación de variables de usuario locales en *JMeter*.

3.2. Configuración de las Solicitudes HTTP

Vamos a proceder a escribir el código común a todas las pruebas el cual irá asociado al "Test Fragment". Se deberán agregar cuatro *Samplers* de tipo **HTTP Request** para simular la interacción con los formularios (dos llamadas GET y dos llamadas POST). Antes de añadirlos vamos a definir algunos elementos que nos permitirán variar los datos introducidos en los formularios de creación de los elementos de forma automática. En concreto, introduciremos un contador que usaremos para modificar los valores de `idProcesoElectoral`, `idMesaElectoral` e `idCircunscripción` así como un lector de ficheros "CSV" que devolverá valores válidos de "censo" y finalmente un generador de números aleatorios que nos permitirá elegir un candidato entre los tres definidos (variables `candidato_X`)

Creación del Contador

Los contadores permiten generar valores secuenciales que pueden ser utilizados en múltiples solicitudes dentro de un test. Para agregar el contador llamado `contador`, sigue estos pasos:

1. **Hacer clic derecho sobre el "Test Fragment"** donde deseas agregar el contador.
2. Navegar a `Add → Config Element → Counter`.
3. Se agregará un **Counter** al plan de pruebas.
4. En la configuración usad los valores:
   - **Name**: `contador`
   - **Start**: 1
   - **Increment**: 1
   - **Reference Name**: `contador`

En la figura 7 se muestra el formulario usado para crear un “Counter”.

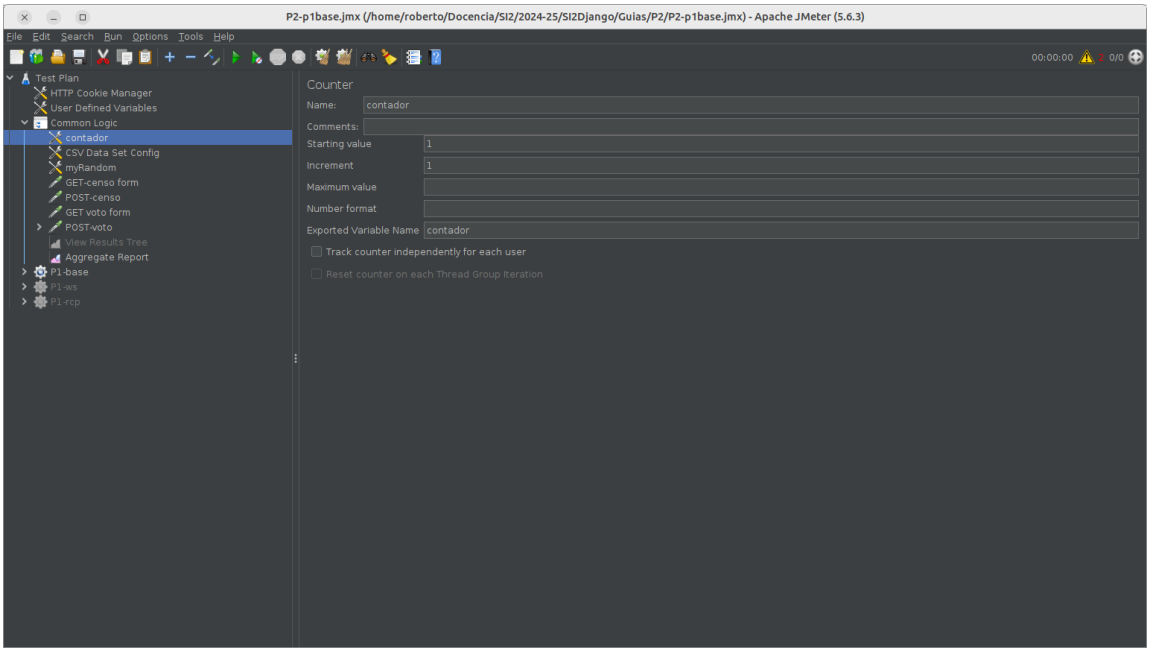

Figura 7: Creación de el "Counter" llamado "contador en *JMeter*.

Lectura del fichero CSV con información de censo

El elemento **CSV Data Set Config** en *JMeter* permite leer datos desde un archivo CSV y usarlos como variables en el test. Para agregarlo, sigue estos pasos:

- Hacer clic derecho sobre el "Test Fragment".
- Navegar a `Add → Config Element → CSV Data Set Config`.

Después de agregar el **CSV Data Set Config**, debes configurar los siguientes parámetros:

- **Filename**: Ruta del archivo CSV (Ejemplo: `/P1-base/votoApp/management/commands/data2.csv`).
- **Variable Names**: Nombres de las variables. Si dejamos este campo en blanco los nombres serán aquellos que figuran en la primera línea del fichero cvs, esto es, numeroDNI, nombre, fechaNacimiento, anioCenso y codigoAutorizacion.

- **Delimiter**: Separador de valores usa una coma ’,’.

En la figura 8 se muestra el formulario usado para crear un elemento “CSV Data Set Config”.

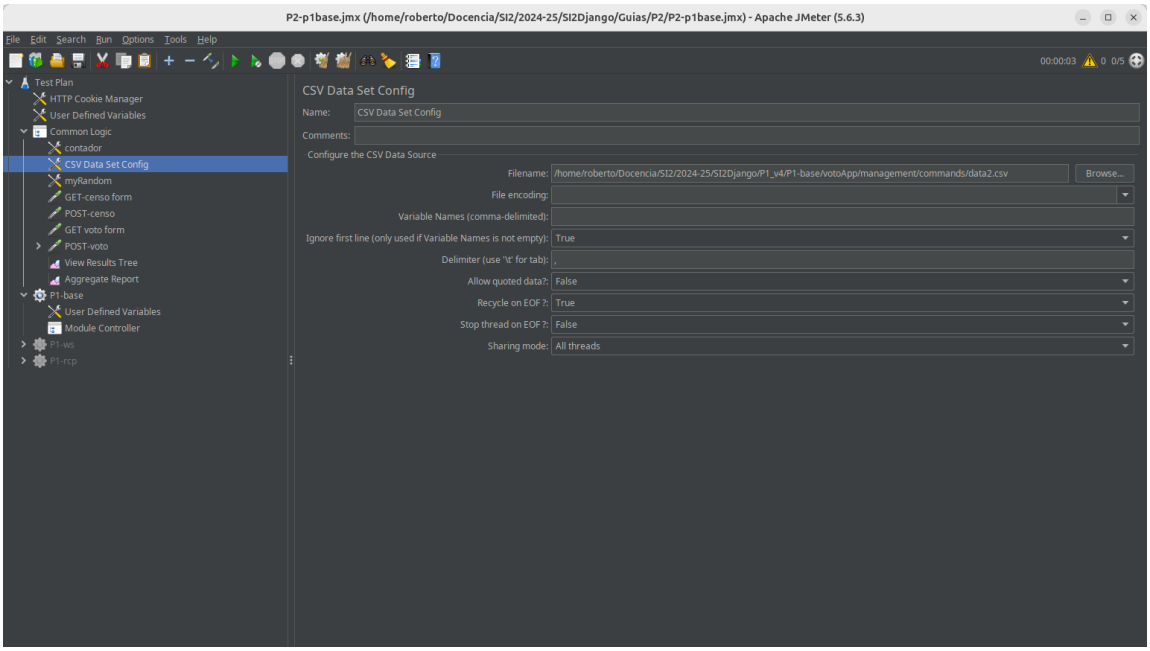

Figura 8: Creación de un elemento “CSV Data Set Config” en *JMeter*.

Creación de variables aleatorias

El elemento **Random Variable** en *JMeter* permite generar valores aleatorios dentro de un rango definido y almacenarlos en una variable para su uso en el test. Lo usaremos para seleccionar un candidato de los tres definidos.

Para agregar uno de estos elementos sigue estos pasos:

1. Hacer clic derecho sobre el elemento "Test Fragment".
2. Navegar a `Add → Config Element → Random Variable`.

Después de agregar el **Random Variable**, debes configurar los siguientes parámetros:

- **Variable Name**: Nombre de la variable (Ejemplo: `myRandom`).
- **Minimum Value**: Valor mínimo del rango (Ejemplo: 1).
- **Maximum Value**: Valor máximo del rango (Ejemplo: 3).

En la figura 9 se muestra el formulario usado para crear un elemento "Random Variable".

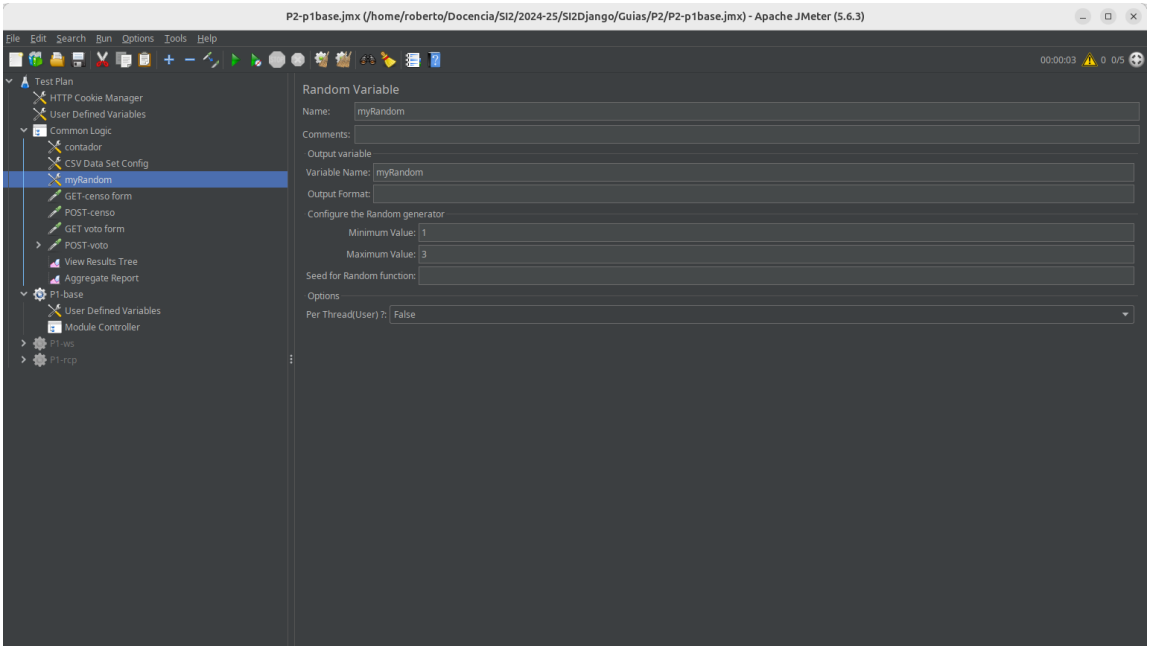

Figura 9: Creación de un elemento "Random Variable" en *JMeter*.

Creación de elementos HTTP request

Como se comentó con anterioridad se deberán agregar cuatro *Samplers* de tipo **HTTP Request** para simular la interacción con los formularios. Se añadirán al elemento "Test Fragments" los siguientes elementos "HTTP Request" (dichos elementos se agregan usando la secuencia, Add → Sampler → HTTP Request):

1. **Obtener el formulario de Censo:**
   - Agregar un **HTTP Request**.

- Configurar el método como **GET**.
- En el campo *Server Name or IP*, ingresar la dirección del servidor almacenada en la variable **host**.
- En el campo *Port*, ingresar el puerto en el que escucha el servidor almacenado en la variable **port**.
- En el campo *Path*, especificar la ruta del formulario de censo usando la variable `entrypoint` (por ejemplo, `${entrypoint}/censo/`).

En la figura 12 se muestra el formulario usado para crear este elemento.

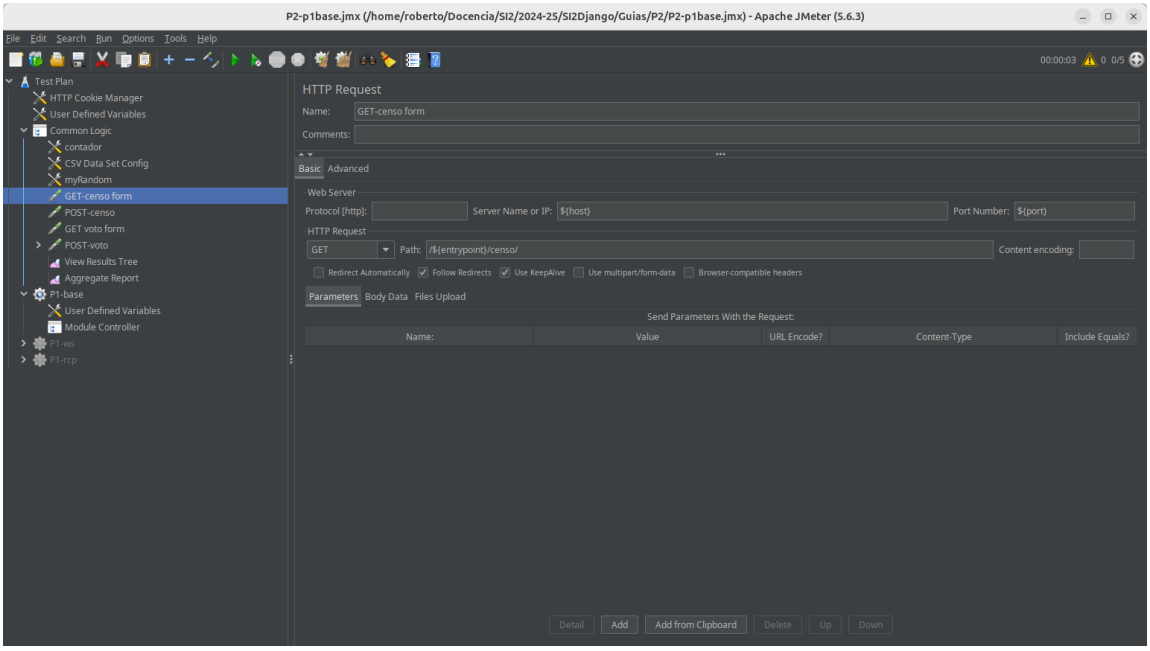

Figura 10: Creación de un elemento "HTTP Request" realizando una llamada GET al endpoint `censo`.

2. **Enviar los datos del Censo:**
   - Agregar otro **HTTP Request**.
   - Configurar el método como **POST**.

- Los campos *Path*, *Port* y *Path* son los mismos que en caso anterior
- En la pestaña *Parameters*, agregar los datos del formulario (nombre, edad, dirección, etc.) usando variables tal y como se muestra en la figura 11.

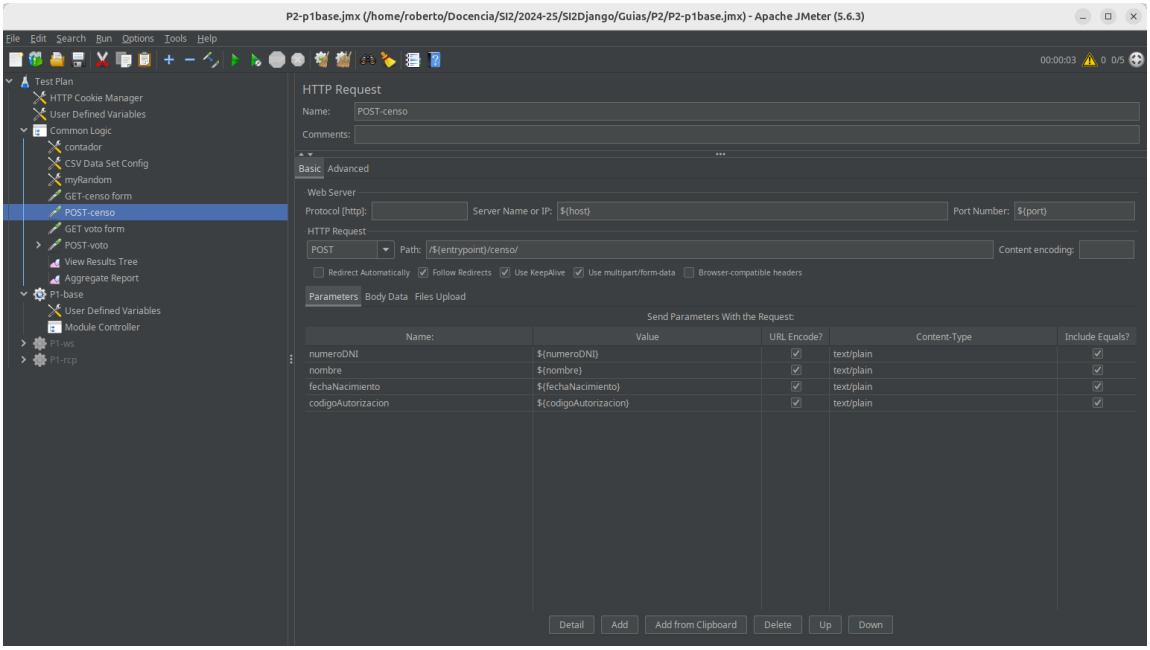

Figura 11: Creación de un elemento "HTTP Request" realizando una llamada POST al endpoint `censo`.

3. **Obtener el formulario de Voto:** Proceder como en el caso del formulario de censo. La única diferencia radica en que la variable `PATH` apunta a "voto" en lugar de a "censo"

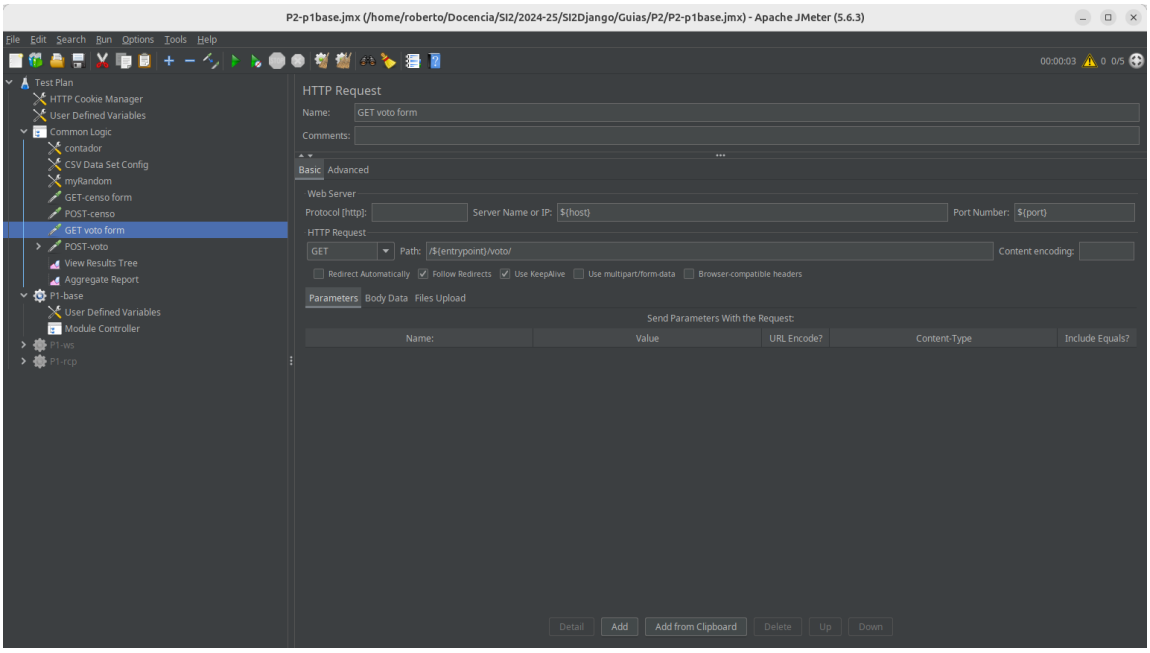

Figura 12: Creación de un elemento "HTTP Request" realizando una llamada GET al endpoint `voto`.

4. **Registrar el voto:** Proceder como en el caso del formulario de voto. Como en el caso anterior la variable `PATH` apunta a "voto" en lugar de a "censo" y los parámetros a agregar serán idProcesoElectoral, idCircunscripcion, etc. tal y como se muestra en la figura 13. Nótese que los casi cualquier valor es acceptable para los parámetros `idProcesoElectoral`, `idCircunscripción` e `idMesaElectoral`. La única limitación es que la misma persona no puede votar dos veces en el mismo proceso electoral.

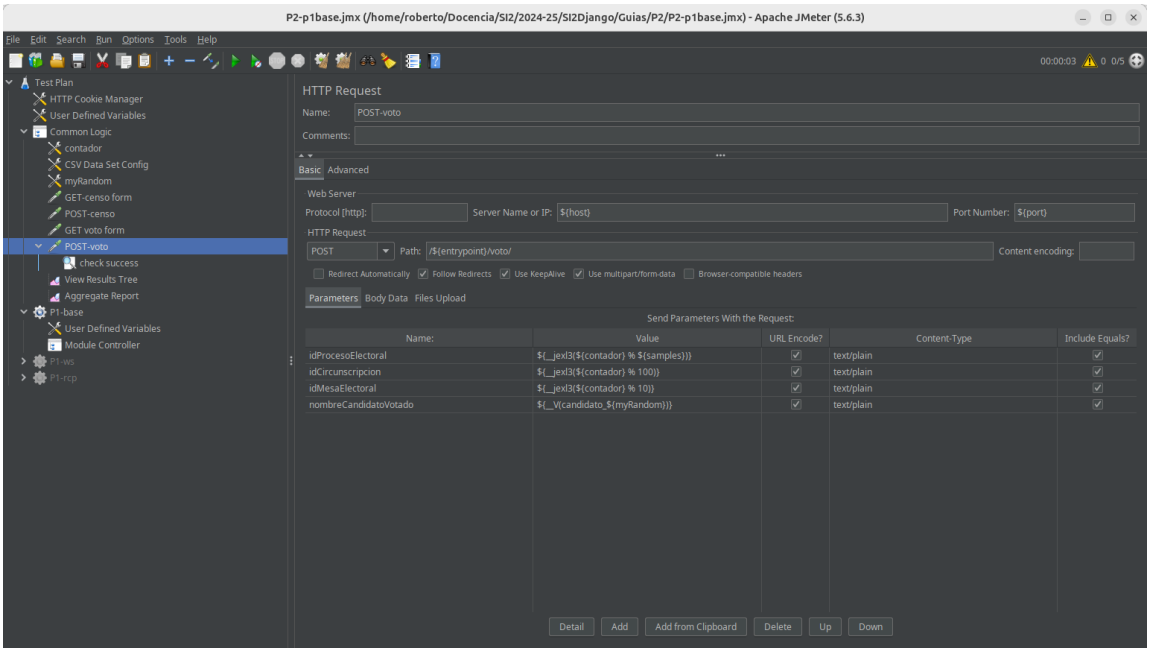

Figura 13: Creación de un elemento "HTTP Request" realizando una llamada POST al endpoint `voto`. "jexl3" significa que se va a realizar una operación aritmética, " %" establece que la operación a realizar es la operación módulo.

Comprobar el resultado de un "HTTP Request"

El elemento **Response Assertion** en JMeter permite verificar si la respuesta de una solicitud HTTP u otro sampler contiene los valores esperados. Para agregarlo, sigue estos pasos:

1. Hacer clic derecho sobre el último "HTTP Request" (o cualquier otro sampler donde quieras validar la respuesta).

2. Navegar a Add → Assertions → Response Assertion.

3. Se agregará un nuevo elemento **Response Assertion** en la estructura del test.

Después de agregar el **Response Assertion** configura los parámetros tal y como se muestra en la figura 14. Este elemento comprueba si la respuesta al formulario

que registra un voto contiene la cadena de caracteres “Voto Registrado” y si este es el caso asume que el proceso de registro se ha completado con éxito.

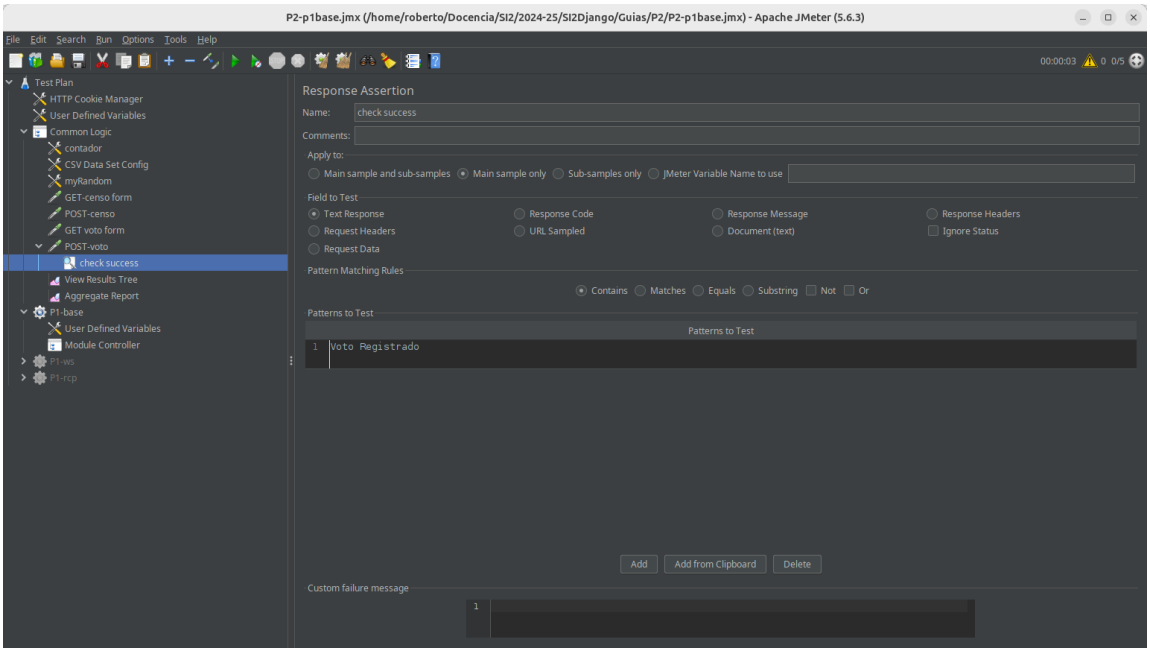

Figura 14: Creación de un elemento “Response Assertion” que comprueba el resultado de la llamada al endpoint `voto`.

3.3. Configuración de Reportes y Análisis de Resultados

En *JMeter*, los elementos de escucha (**Listeners**) permiten visualizar y analizar los resultados de las pruebas. Dos de los más utilizados son:

- **Aggregate Report**: Muestra métricas agregadas como el tiempo de respuesta promedio, la desviación estándar y la tasa de errores.
- **View Results Tree**: Permite ver los detalles individuales de cada solicitud y su respuesta.

Para crear estos elementos en *JMeter*, sigue estos pasos:

1. Hacer clic derecho sobre el “HTTP Request” donde deseas crear los elementos.

2. Navegar a `Add → Listener → Aggregate Report` para crear el reporte agregado.
3. Navegar a `Add → Listener → View Results Tree` para crear el visor de resultados.

Descripción del resultado del "Aggregate Report" El **Aggregate Report** muestra métricas sobre el rendimiento de las peticiones. Sus columnas incluyen:

- **Label**: Nombre del "HTTP request" analizado.
- **# Samples**: Número total de solicitudes enviadas.
- **Average**: Tiempo de respuesta promedio.
- **Min**: Tiempo mínimo de respuesta.
- **Max**: Tiempo máximo de respuesta.
- **90 % Line, 95 % Line, 99 % Line**: Percentiles de tiempo de respuesta.
- **Error %**: Porcentaje de solicitudes fallidas.
- **Throughput**: Solicitudes procesadas por segundo.

En la figura 15 se muestra el resultado de ejecutar este elemento.

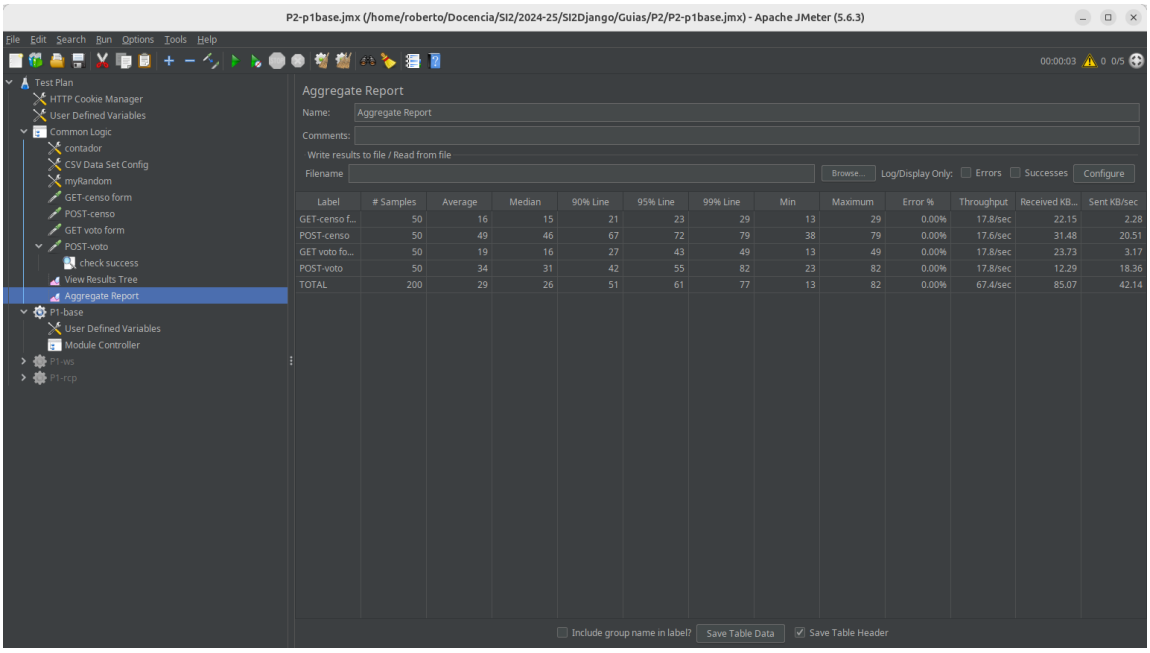

Figura 15: Resultado de ejecutar el elemento “Aggregate Report”.

Configuración del View Results Tree El **View Results Tree** permite ver los detalles individuales de cada solicitud y la respuesta obtenida. Sus pestañas incluyen:

- **Sampler result**: Muestra información detallada de la solicitud.
- **Request**: Contenido de la petición enviada.
- **Response data**: Contenido de la respuesta recibida, útil para verificar errores o datos esperados.

En la figura 16 se muestra el resultado de ejecutar este elemento.

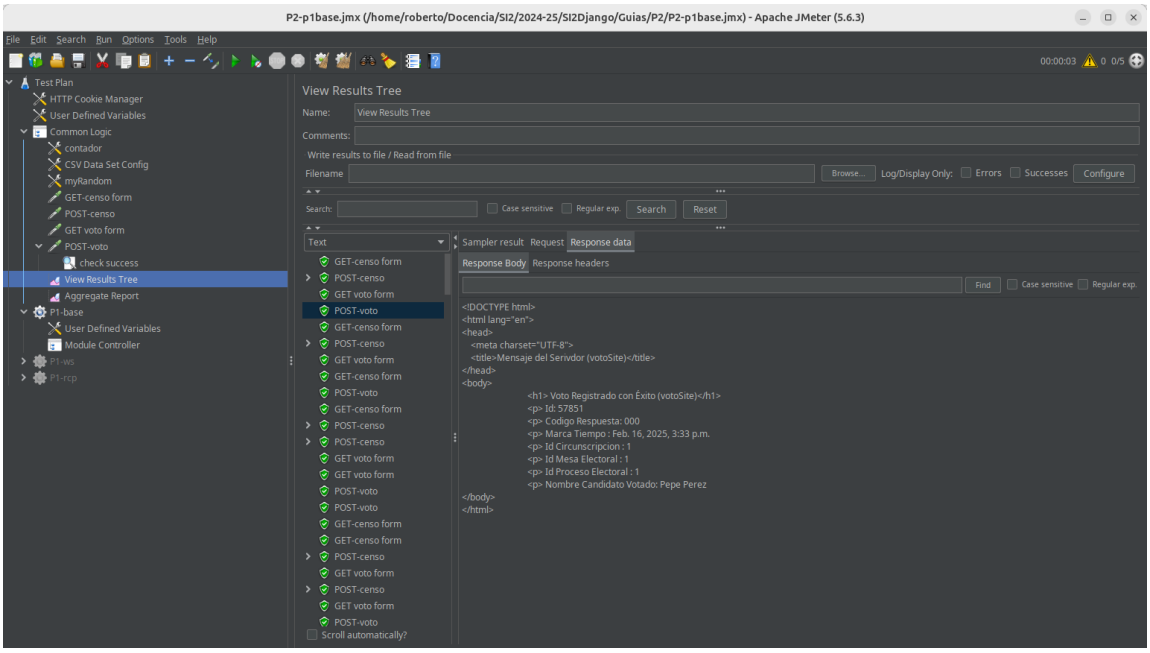

Figura 16: Resultado ejecutar el elemento "View Results Tree".

Consideraciones de rendimiento

- El elemento **View Results Tree** consume bastante memoria, por lo que se recomienda usarlo solo para depuración y no en pruebas de carga masivas.

Llamar al "Test Fragment" desde los "Thread Groups"

El **Module Controller** en *JMeter* permite reutilizar un **Test Fragment** dentro del mismo **Test Plan**, facilitando la modularidad y evitando la duplicación de elementos.

Pasos para crear el "Module Controller"

- Hacer clic derecho en `Thread Group`.
- Navegar a `Add → Logic Controller → Module Controller`.
- Hacer clic en el **Module Controller**.

- En la opción `Module To Run`, seleccionar el **Test Fragment** creado previamente.

En la figura 17 se muestra el resultado de ejecutar este elemento.

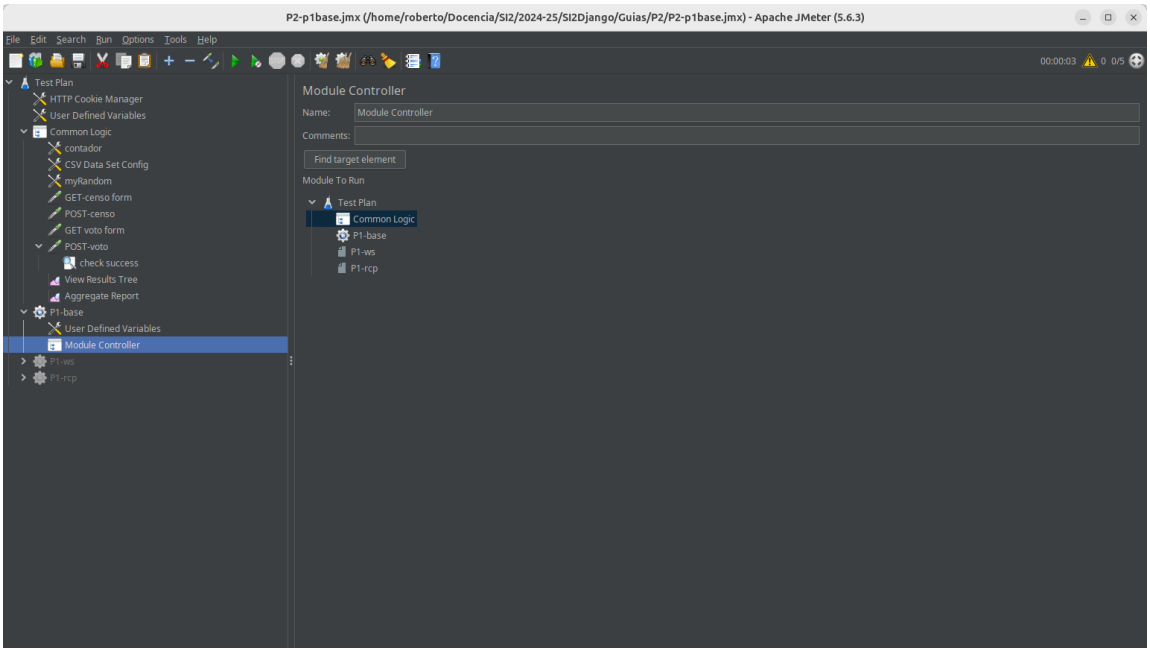

Figura 17: Formulario de creación del elemento 'Module Controler".

Ya tenéis configurado el plan de pruebas. Es el momento de probarlo con `samples=10` y Number of threads=10.

NOTA: en todas las pruebas que involucren el proyecto P1-base se asumirá la existencia de un único PC. La base de datos se desplegará en la máquina virtual `vm1` y el servidor en la máquina virtual `vm2`. En las pruebas que involucren los proyectos P1-ws o P1-rcp se utilizarán dos PCs, en el primero se ejecutará el proyecto cliente (en `vm3`) y en el segundo PC se ejecutarán tanto el proyecto servidor (en `vm2`) como la base de datos (en `vm1`). No hagáis las mediciones ejecutando las 3 máquinas virtuales en el mismo PC puesto que no habrá acceso a la tarjeta de red y los resultados obtenidos serán muy diferentes a los del caso solicitado.

Ejercicio 1: Siguiendo todos los pasos anteriores, defina el plan completo de pruebas para el proyecto P1-Base. Guarda este plan en un fichero llamado P2_P1-base.jmx y entrégalo con la práctica. Adjunta en la memoria una captura del resultado de ejecutar el "Aggregate Report" y comenta el significado de cada columna.

Ejercicio 2: Generaliza el plan de pruebas de forma que las pruebas se realicen sobre los tres proyectos creados en la práctica 1: P1-base, P1-rcp y P1-ws. Guarda este plan en un fichero llamado P2-projects.jmx y adjúntalo a la práctica.
El plan de pruebas para P1-base ya lo tenéis creado y para los dos otros casos tendréis que duplicar el "Thread Group" P1-base, renombrar la copia adecuadamente y modificar los parámetros que almacenan el "entrypoint", "host" y "port" para que apunten a la aplicación web adecuada.

Cuestión 1: Enumera las diferencias entre el "Thread Group" que prueba el proyecto P1-base y el que prueba el proyecto P1-ws

Ejercicio 3: Usando el "Thread Group" llamado "P1-base" si "Ramp-Up" = 1 sec y "Number of Threads" = 1000 se intentan crear 1000 solicitudes en 1 segundo. ¿En esta situación que ocurre en los ordenadores de los laboratorios? (a) *JMeter* no es capaz de obtener los recursos necesarios para crear 1000 peticiones en 1 segundo, (b) Aunque *JMeter* puede crear 1000 peticiones por segundo el "throughput" (esto, las peticiones atendidas por el servidor por segundo) será muy inferior aunque finalmente serán todas atendidas, (c) la capacidad del servidor se desborda y las peticiones se pierden o devuelven errores. A la vez que ejecutas las pruebas comprueba el uso de: memoria, disco, cpu y red en los distintos ordenadores (o maquinas virtuales) implicados en el cálculo. Para ello puedes usar la herramienta nmon (véase appendice A).
Argumenta tu respuesta e incluye una descripción de los experimentos realizados para comprobar cual de las hipótesis es la correcta. Aporta capturas de pantalla para dar soporte a tu argumento.

Ejercicio 4: En un "Thread Group", la variable "Loop Count" define cuantas veces cada "thread" (usuario) ejecuta una secuencia de peticiones. ¿Cuando comienza cada loop (bucle)? (a) El segundo bucle empieza tan pronto como todas las peticiones del primer bucle se han creado o (b) El segundo bucle comienza tan pronto todas las peticiones del primer bucle se han atendido.
Argumenta tu respuesta incluye una descripción de los experimentos realizados para comprobar cual de las hipótesis es la correcta. Aporta capturas de pantalla para dar soporte a tu argumento.

4. Saturación de un servidor: Curva de productividad

La **curva de productividad** de un servidor web representa la relación entre la cantidad de solicitudes procesadas y el rendimiento del servidor en función de la carga. Se utiliza para analizar el comportamiento del servidor bajo diferentes niveles de tráfico y detectar el punto en el que su rendimiento comienza a degradarse.

La curva de productividad generalmente presenta tres fases:

- **Fase de Crecimiento Lineal:** Para cargas bajas, el rendimiento del servidor aumenta proporcionalmente al número de solicitudes por segundo.
- **Punto de Saturación:** Se alcanza la capacidad máxima del servidor, donde el *throughput* se mantiene constante, pero la latencia comienza a aumentar.
- **Fase de Degradación:** Si la carga continúa aumentando, el servidor empieza a rechazar solicitudes o a procesarlas con tiempos de respuesta muy altos.

Idealmente el servidor debe diseñarse para que opere dentro de la fase de crecimiento lineal, evitando llegar a la saturación y minimizando la degradación del rendimiento.

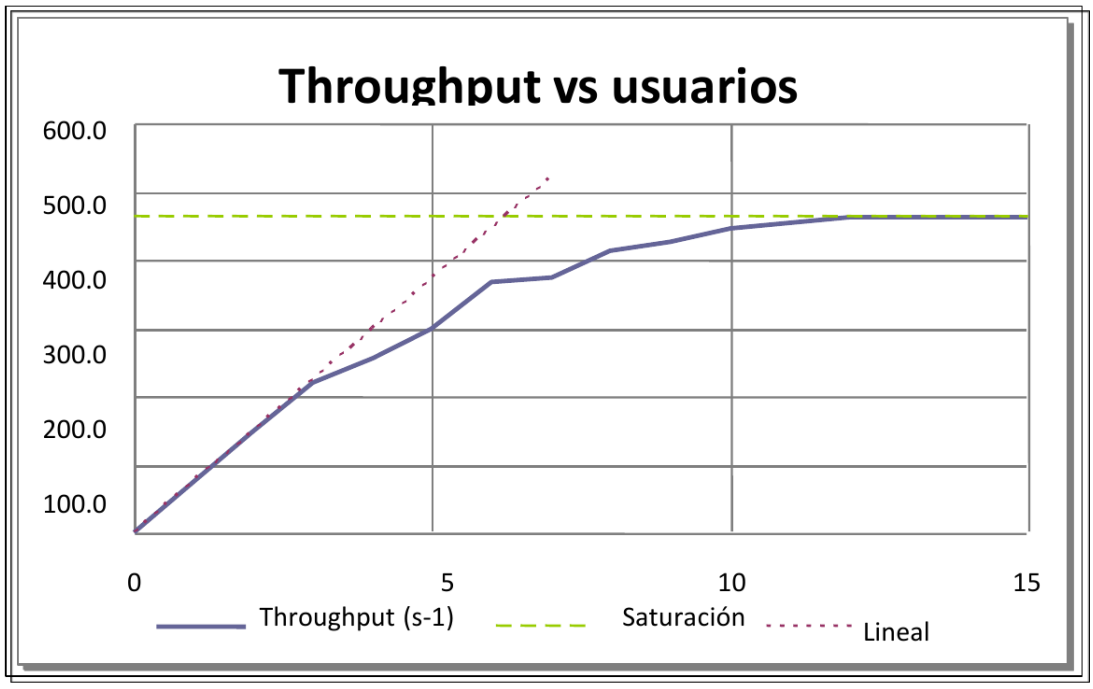

Figura 18: Curva de productividad de un servidor web.

A la hora de calcular la curva de productividad es útil automatizar las medidas. Para ello edita el "Thread Group" y modifica "Number of Threads (Users)" a `${__P(users,10)}`. Esto significa que el número de usuarios se define en la variable `users` y que por defecto vale 10. Si ahora ejecutamos *JMeter* en modo línea de comandos

```
jmeter.sh -n -t fichero_plan.jmx -Jusers=5 -l results_$i.jtl \
-Jsummariser.name=summary  -e -o output_users_5
```

La información relacionada con el valor $users = 5$ estará almacenada en el directorio `output_users_5`. La forma más sencilla de acceder a la misma sería abrir el fichero "index.html" que hay en la raíz de `output_users_5` y apuntar en una tabla el `thoughput Total` vs el número de usuarios (o hilos). Repite la misma operación para diferentes números de usuarios hasta obtener una representacion similar a la de la figura 18.

A la hora de realizar estos cálculos es importante no usar el interfaz gráfico de *JMeter* para no introducir demoras adicionales relacionadas con el repintado de la aplicación.

Ejercicio 5: Calcula la curva de productividad para cada uso de los tres proyectos llamados P1-base, P1-ws-client/server y P1-rpc-client/server e incluye el resultado en la memoria. Esto es, se espera que incluyáis una grafica como la mostrada en la figura 18 para cada caso. ¿Qué implementación es más eficiente? ¿Por qué? Cada experimento debe repetirse 10 veces (usad la variable "samples" para lograr esta repetición). El "ramp up" debe ser pequeño con respecto al tiempo que se tarda en ejecutar todas las peticiones. Finalmente, aseguraros de que las prueba (grupos de hilos) se ejecutan secuencialmente y no concurrentemente. Incluid en la memoria junto a la gráfica la tabla donde habéis apuntado los throughputs.

5. Pruebas de Carga

En esta sección vamos a trabajar únicamente con el "Thread Group" P1-base (deshabilita los otros).

Ejercicio 6: ¿Qué servidor es más eficiente **gunicorn** con una CPU o el servidor de desarrollo "python manage.py runserver 8000"? Calcula la curva de productividad para el caso "python manage.py runserver 8000" y comparala con la optenida para **gunicorn**. Pare realizar esta prueba tendrás que parar **gunicorn** y lanzar el servidor de desarrollo. Nota: Por defecto, "python manage.py runserver" solo es accesible desde el localhost (127.0.0.1). Para permitir conexiones desde otros dispositivos, usa "python manage.py runserver 0.0.0.0:8000". Da una repuesta razonada e incluye en la memoria el resultado de las medidas realizadas para contestar esta pregunta.

Como ya se ha comentado **gunicorn** permite usar varios "workers" (parametro –workers) y/o "threads" (parámetro –threads) para incrementar el rendimiento.

Ejercicio 7: Calcula la curva de productividad de la aplicación P1-base usando **gunicorn** con 2 workers. Puesto que la maquina virtual está configurada para usar una única CPU incrementar el número de workers a dos no otorga a **gunicorn** más recursos. Modifica la configuración de la máquina virtual para que tenga a su disposición 2CPUs en lugar de una. Incluye la curva calculada en la memoria y comenta el resultado. Nota: no olvides usar `SESSION_ENGINE=django.contrib.sessions.backends.db` en settings.py puesto que en caso contrario las variables de sesión pueden no ser recuperables.

Cuestión 2: Si repetimos el ejercicio 7 usando el proyecto P1-ws-client/master en lugar de P1-base, ¿en que fichero settings.py tenemos que modificar la variable SESSION_ENGINE?: (a) en el del servidor, (b) en el del cliente, (c) en ambos, (d) en ninguno. Justifica la respuesta.

Material a entregar al finalizar la práctica: se debe subir a *Moodle* un fichero zip que contenga la memoria (en formato pdf) respondiendo a todos los ejercicios y cuestiones planteadas en esta guía junto con los ficheros que definen los planes de pruebas solicitados. Igualmente incluid el código que implementa los proyectos P1-base, P1-ws-client/server y P1-rcp-client/server de forma que sea fácil ejecutar los planes de prueba. Se considerará que el código funciona si lo hace en los ordenadores del laboratorio. El código entregado en *Moodle* será el evaluado, bajo ninguna condición se evaluará el código existente en los diversos repositorios.

6. Criterios de evaluación

Para aprobar con un 5 es necesario haber implementado el plan de pruebas para el proyecto P1-base usando un worker de **gunicorn**. El plan debe funcionar al ser ejecutado. A esta nota inicial se sumarán las cantidades siguientes:

Ejercicio	Puntuación
2	0.5
3	0.5
4	0.5
5	1.0
6	0.5
7	1.0
Cuestión	
1	0.5
2	0.5

A. Utilidad nmon

La utilidad `nmon` es una herramienta para monitorear el rendimiento del sistema en Linux. Para visualizar métricas de **CPU**, **memoria**, **acceso a disco** y **uso de red** en tiempo real, se puede ejecutar el siguiente comando:

```
1 nmon
```

Una vez dentro de la interfaz de `nmon`, se pueden presionar las siguientes teclas para activar cada métrica:

- **c**: Uso de CPU.
- **m**: Memoria.
- **d**: Actividad de discos.
- **n**: Estadísticas de red.

Además, para registrar estos datos en un archivo y analizarlos posteriormente, se puede usar:

```
1 nmon -f -s 1 -c 60
```

Esto generará un archivo con extensión `.nmon`, el cual puede ser procesado con herramientas como `nmon_analyzer` o `nmonchart`.