2024

JMeter Training and Support for Web Application Performance Testing

PREPARED BY
JOHN REY GOH

Contents

Introduction to Performance:

Introduction To Performance Testing

Performance testing is a type of software testing that evaluates how a system performs under various conditions. It determines the responsiveness, stability, and scalability of a software application under different workloads.

Performance testing is a type of non-functional testing aimed at determining how a system performs in terms of responsiveness and stability under a particular workload. It helps identify and eliminate performance bottlenecks.

Need For Performance Testing

The need arises from the critical importance of software performance in today's fast-paced digital world. Poor performance can lead to user dissatisfaction, lost revenue, and damaged reputation. Performance testing helps identify and eliminate bottlenecks before they impact end-users.

Objectives of Performance Testing

- ✓ Assess Performance: Evaluate the speed, responsiveness, and stability of the application.
- ✓ Identify Bottlenecks: Pinpoint the elements that are causing slow performance.
- ✓ Ensure Stability: Verify that the application can handle high loads without crashing.
- ✓ Optimize Scalability: Ensure the application can scale up or down to meet demand.

Performance Testing Concepts

Performance Testing Concepts encompass a range of metrics, methodologies, and considerations that help us quantify and analyze system performance. Let's break these down:

1. Response Time:
   This is the time taken for the system to react to a given input. It's often measured from the moment a user action is initiated to when the system provides a complete response. We typically look at:
   - Average response time
   - Peak response time
   - Response time under different load conditions

2. Throughput:
   This measures the number of transactions the system can process in a given time unit. It's usually expressed as transactions per second (TPS) or requests per second (RPS).

3. Concurrency:
   This refers to the number of simultaneous users or transactions the system can handle without significant performance degradation.

4. Resource Utilization:
   This involves monitoring system resources such as CPU usage, memory consumption, disk I/O, and network bandwidth during testing.

5. Scalability:
   This concept examines how well the system's performance scales as the load increases. We look at both vertical scalability (adding more resources to a single node) and horizontal scalability (adding more nodes).

6. Bottlenecks:
   These are constraints in the system that limit overall performance. Identifying and addressing bottlenecks is a key goal of performance testing.

7. Load Balancing:
   This concept involves distributing workload across multiple computing resources to optimize resource use and minimize response time.

8. Caching:
   Caching strategies can significantly impact performance. We often test with and without caching to understand its effects.

9. Database Performance:
   This includes concepts like query optimization, indexing, and connection pooling, which can greatly affect overall system performance.

10. Network Latency:
    This measures the delay in data transmission over a network. It's particularly important for distributed systems and cloud applications.

11. Percentiles:
    Often, we look at percentile measurements (e.g., 90th percentile response time) to get a more nuanced view of performance beyond averages.

12. Baseline Performance:
    This establishes a reference point for system performance under normal conditions, against which we can compare performance under various test scenarios.

13. Performance Tuning:
    This involves adjusting system parameters to optimize performance based on test results.

14. Workload Modeling:
    This concept involves creating test scenarios that accurately reflect real-world usage patterns.

Functional V/s Performance Testing

Functional Testing

Purpose:

- Verifies that the software operates according to the specified requirements.
- Ensures each function of the software application behaves as expected.

Focus:

- Correctness of the application's features and functionalities.
- User interactions and data input/output validation.

Key Activities:

- Unit Testing: Testing individual components.
- Integration Testing: Testing combined parts of an application to ensure they work together.
- System Testing: Testing the complete system as a whole.
- User Acceptance Testing (UAT): Testing to validate the end-to-end business flow.

Tools:

- Selenium
- QTP (Quick Test Professional)
- TestComplete
- Postman (for API testing)

Example Scenarios:

- Checking if a login function works with valid and invalid credentials.
- Verifying if a user can add items to a shopping cart.
- Ensuring a form submission results in the correct database entry.

Performance Testing

Purpose:

- Evaluates the speed, responsiveness, and stability of the application under various conditions.
- Identifies performance bottlenecks and ensures the application can handle expected load.

Focus:

- Non-functional aspects like response time, throughput, and resource utilization.
- Behavior of the application under different levels of stress and load.

Key Activities:

- Load Testing: Simulating multiple users to test the system’s behavior under expected load.
- Stress Testing: Pushing the system beyond its limits to see how it handles extreme conditions.
- Endurance Testing: Running the system for an extended period to check for memory leaks or performance degradation.
- Spike Testing: Testing the system’s response to sudden spikes in user activity.

Tools:

- Apache JMeter
- LoadRunner
- Gatling
- NeoLoad

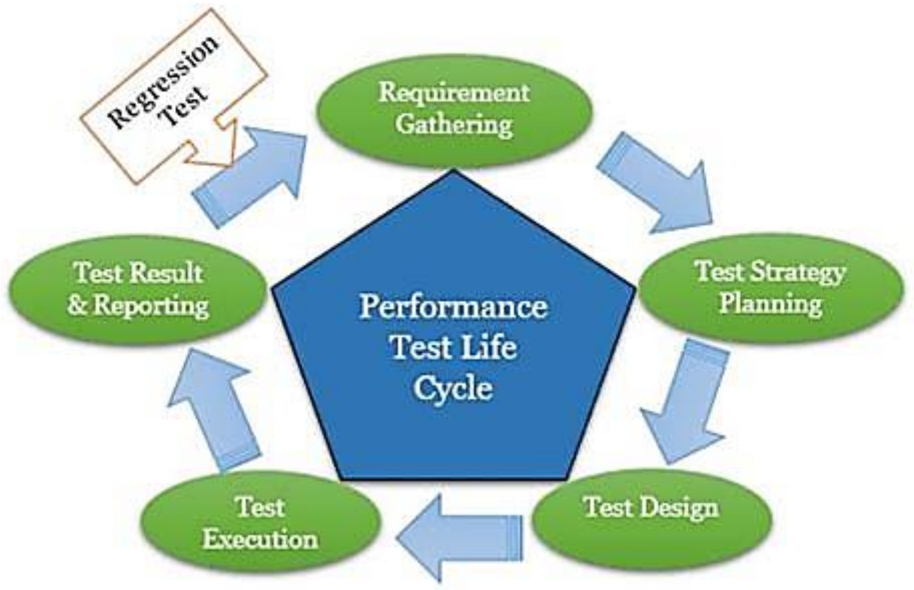
Example Scenarios:

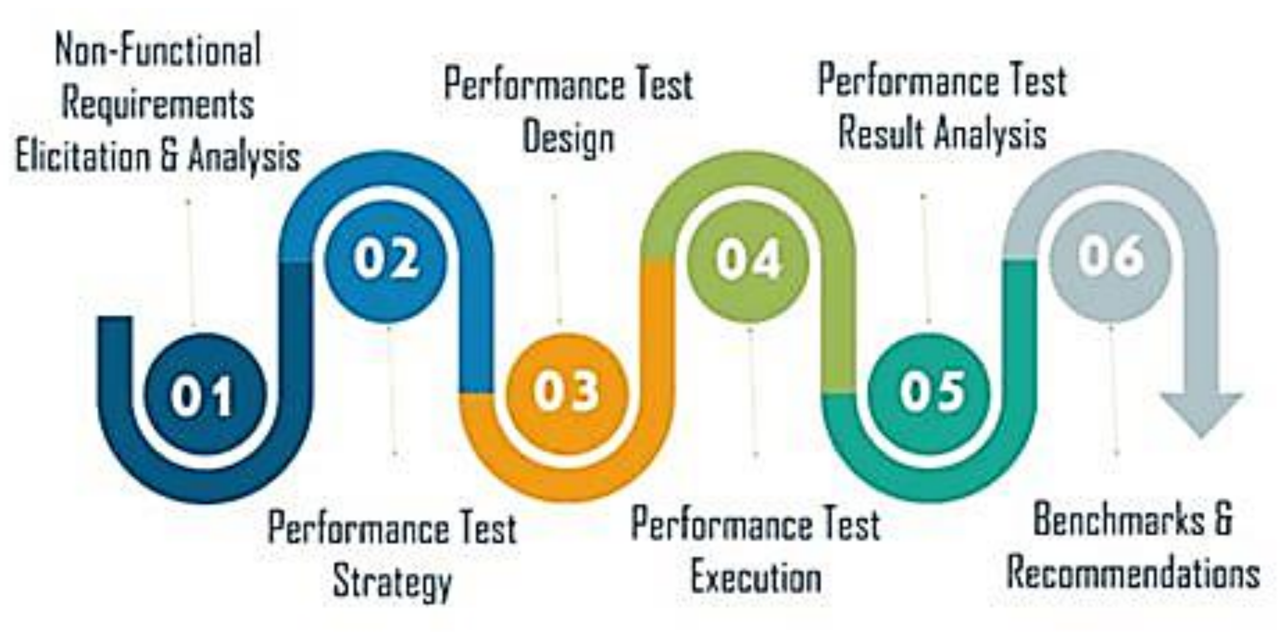
- Measuring the response time of a web page under heavy traffic.
- Determining the maximum number of concurrent users an application can support.
- Checking the system's stability during prolonged usage.

Performance Test Life Cycle

The Performance Test Life Cycle (PTLC) is a structured approach to ensure optimal application performance. It involves several key phases:

1. Requirement Gathering and Analysis: Identify performance goals, KPIs, and constraints.
2. Performance Test Planning: Define test objectives, scope, environment, resources, and schedule.
3. Test Design and Scripting: Create test scripts, scenarios, and data sets using tools like JMeter.
4. Workload Modeling: Define user behavior, load patterns, and data volume.
5. Test Environment Setup: Configure hardware, software, and network infrastructure.
6. Test Execution: Run the test scripts and monitor performance metrics.
7. Analysis and Reporting: Analyze test results, identify performance bottlenecks, and generate reports.
8. Tuning and Optimization: Implement changes to improve application performance.




Non-Functional Requirements Elicitation & Analysis
01
Performance Test Strategy
02
Performance Test Design
03
Performance Test Execution
04
Performance Test Result Analysis
05
Benchmarks & Recommendations
06

Why to use performance Testing tool?

Performance testing tools like JMeter are essential for:

- ✓ Identifying performance bottlenecks: Pinpoint areas of the application that are causing slowdowns.
- ✓ Simulating real-world load: Accurately replicate user behavior and traffic patterns.
- ✓ Measuring response times: Evaluate how quickly the application responds to user requests.
- ✓ Monitoring resource utilization: Analyze CPU, memory, and network usage.
- ✓ Preventing performance issues: Identify and resolve problems before deployment.

Performance Testing Types

Load Testing:

- Simulates expected user load to assess system performance.
- JMeter Example: Create a Thread Group with a specified number of users and ramp-up time. Add HTTP Request samplers to simulate user actions.

Stress Testing:

- Exceeds normal system load to determine its breaking point.
- JMeter Example: Increase the number of users in the Thread Group dramatically to simulate extreme load conditions.

Spike Testing:

- Simulates sudden, sharp increases in user load.
- JMeter Example: Use the Constant Throughput Timer to maintain a constant request rate, then suddenly increase the throughput.

Endurance Testing:

- Verifies system performance under sustained load over a long period.
- JMeter Example: Run a load test for an extended duration (e.g., several hours) and monitor resource utilization.

Demonstrating with JMeter
Note: To follow these examples, you'll need to have JMeter installed and a target application to test.

Creating a Simple Load Test

1. Create a Test Plan: Right-click on the Test Plan and add a Thread Group.
2. Add a Thread Group: Configure the number of threads (users), ramp-up period, and loop count.
3. Add HTTP Request Sampler: Define the target server, path, and request method.
4. Add a Listener: Choose a listener like View Results in Table or Graph Results to visualize results.

Example JMeter Script (Basic Structure)

```
<hashTree>
 <TestPlan guiclass="TestPlanGui" testname="Test Plan" enabled="true">
  <stringProp name="TestPlan.properties">
   <value/>
  </stringProp>
  <elementProp name="ThreadGroup">
   <collectionProp name="ThreadGroup.main_controller">
    <elementProp name="HTTP Request">
     <stringProp name="Server Name or IP">
      <value>your_server</value>
     </stringProp>
     <stringProp name="Path">
      <value>/your_path</value>
     </stringProp>
    </elementProp>
   </collectionProp>
  </elementProp>
  <elementProp name="Listener">
   <collectionProp name="Listener.listener">
    <elementProp name="View Results in Table">
     </elementProp>
   </collectionProp>
  </elementProp>
 </TestPlan>
 <hashTree/>
</hashTree>
```

Running the Test and Analyzing Results

1. Click the green play button to start the test.
2. Observe the results in the chosen listener.
3. Analyze response times, error rates, and other metrics to assess performance.

Core Concepts:

Architecture Of Client Server

A client-server architecture involves two main components:

- Client: A software application that requests services from a server.
- Server: A software application that provides services to clients.

Example: A web browser is a client requesting web pages from a web server.

Web Application and Concepts of Different tiers

A web application is a client-server application that uses web technology. It typically consists of three tiers:

- Presentation Tier: Handles user interface and interaction.
- Logic Tier: Processes business logic and data access.
- Data Tier: Stores and manages data.

Example: An online shopping website. The presentation tier displays products, the logic tier processes orders, and the data tier stores product information and customer data.

Concept of Web Servers and Different Servers

A web server is a software application that handles HTTP requests and serves web pages.

- Apache HTTP Server: Open-source web server.
- Microsoft IIS: Web server included in Windows Server.
- Nginx: High-performance web server.

Other Servers:

- Application Server: Handles application logic (e.g., Tomcat, JBoss).
- Database Server: Stores and manages data (e.g., MySQL, Oracle).

Response Codes and their interpretation

1xx: Informational Responses

- 100 Continue: The server has received the request headers and the client should proceed to send the request body.
- 101 Switching Protocols: The requester has asked the server to switch protocols, and the server has agreed to do so.

2xx: Success

- 200 OK: The request was successful, and the server returned the requested resource.
- 201 Created: The request was successful, and a new resource was created.
- 202 Accepted: The request has been accepted for processing, but the processing is not complete.
- 204 No Content: The server successfully processed the request, but there is no content to return.

3xx: Redirection

- 301 Moved Permanently: The requested resource has been moved to a new permanent URI.
- 302 Found: The requested resource has been temporarily moved to a different URI.
- 304 Not Modified: The resource has not been modified since the version specified by the request headers.

4xx: Client Errors

- 400 Bad Request: The server could not understand the request due to invalid syntax.
- 401 Unauthorized: The request requires user authentication.
- 403 Forbidden: The server understood the request but refuses to authorize it.
- 404 Not Found: The server could not find the requested resource.
- 405 Method Not Allowed: The request method is not supported for the requested resource.
- 409 Conflict: The request could not be processed due to a conflict with the current state of the resource.
- 410 Gone: The requested resource is no longer available and will not be available again.

5xx: Server Errors

- 500 Internal Server Error: The server encountered an unexpected condition that prevented it from fulfilling the request.
- 502 Bad Gateway: The server received an invalid response from the upstream server.
- 503 Service Unavailable: The server is not ready to handle the request, often due to temporary overloading or maintenance.

JMeter Introduction:

JMeter Introduction

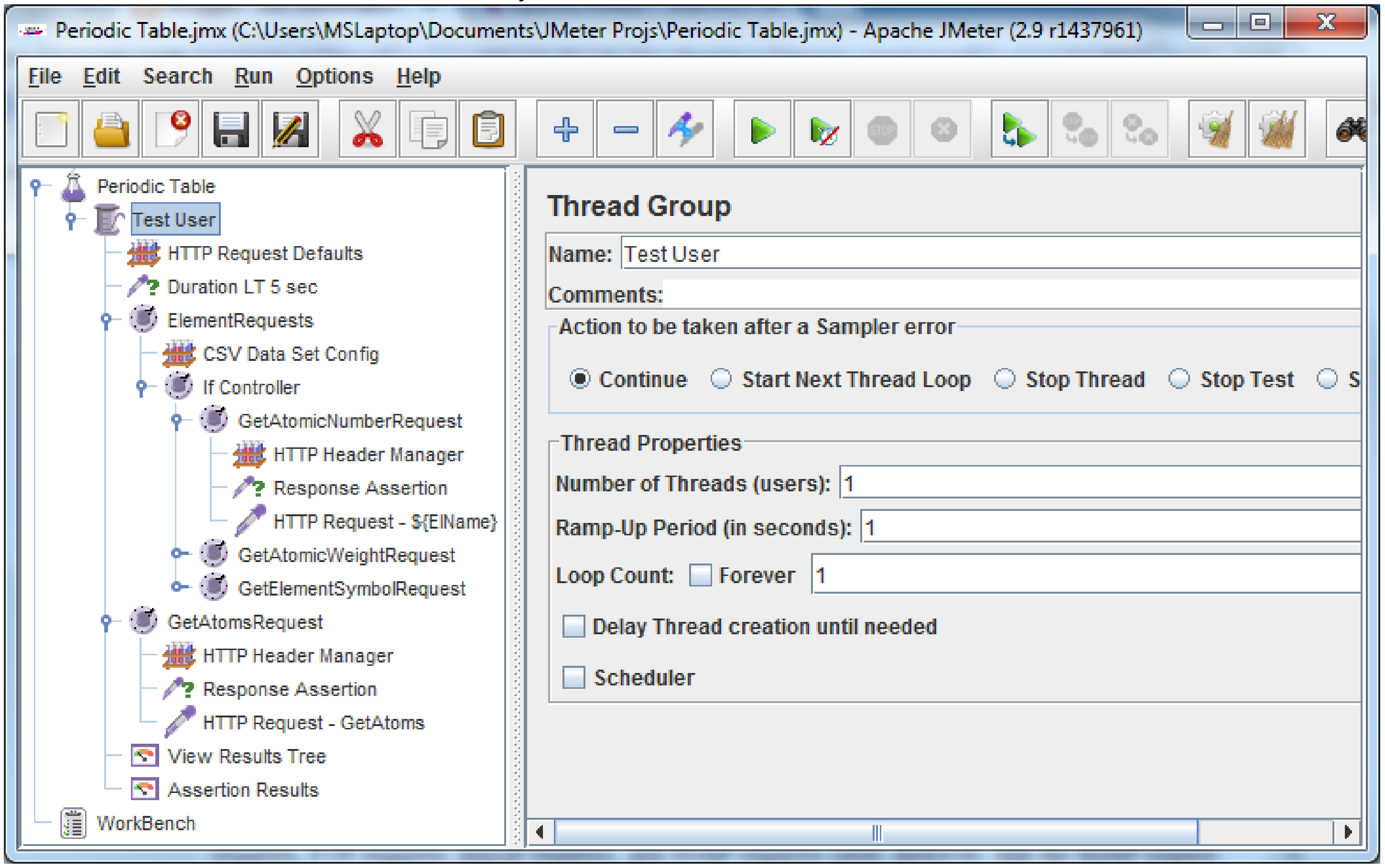
JMeter is an open-source Apache software foundation project. It is designed to load test functional behavior and measure performance. It can be used to simulate a heavy load on a server, network, or object to test its strength or analyze overall performance under different load types.

JMeter Tool overview

JMeter is a Java desktop application used for performance and load testing. It can be used to test static and dynamic resources, recording tests, replaying tests, and building a complex test plan.

JMeter Installation and Configuration[Windows]

1. Download: Download the latest JMeter binary from the Apache JMeter website. https://jmeter.apache.org/
2. Extract: Extract the downloaded zip file to a desired location.
3. Set Environment Variable: Add the bin directory of JMeter to the PATH environment variable.
4. Run JMeter: Double-click on the jmeter.bat file.



Thread Group

Name: Test User

Comments:

Action to be taken after a Sampler error

[✓ Continue] Start Next Thread Loop Stop Thread Stop Test S

Thread Properties

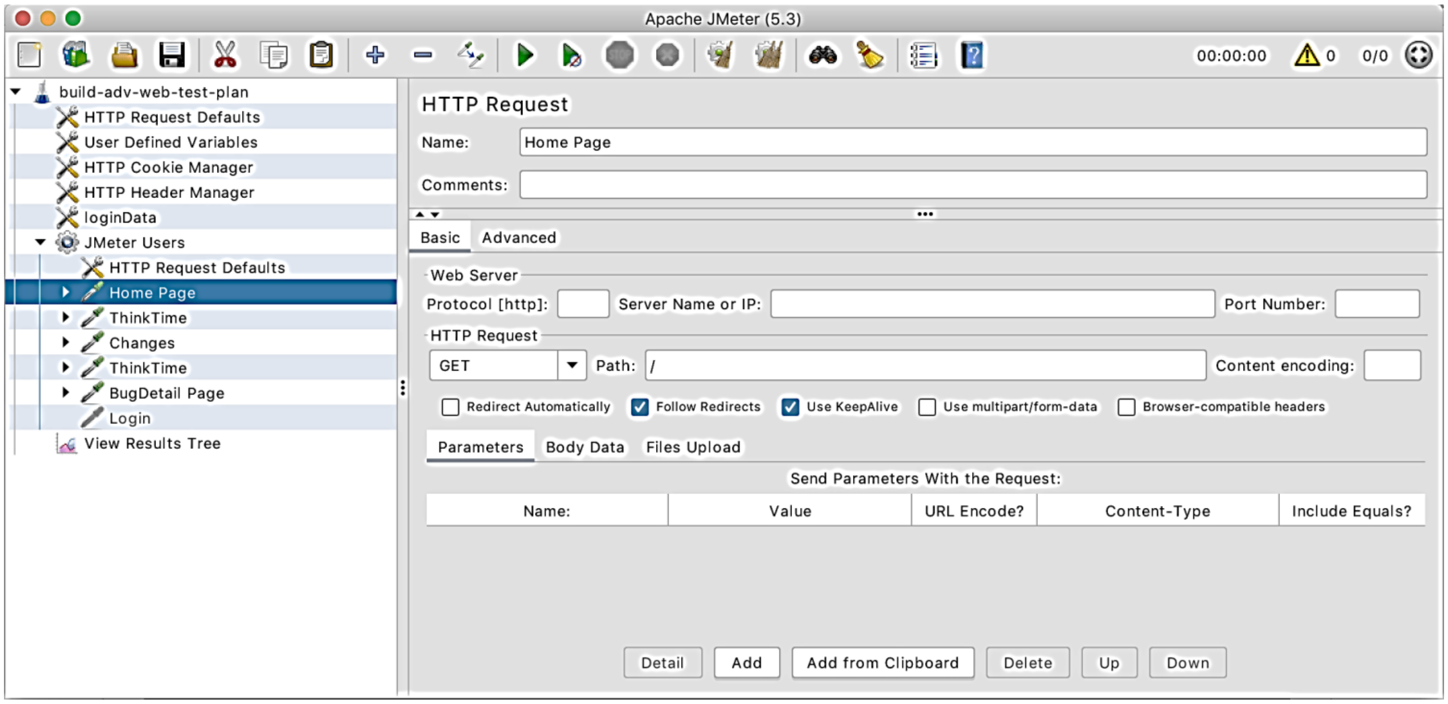
Number of Threads (users): 1

Ramp-Up Period (in seconds): 1

Loop Count: ☐ Forever 1

☐ Delay Thread creation until needed

☐ Scheduler



HTTP Request

Name: Home Page

Comments:

Web Server

Protocol [http]: Server Name or IP: Port Number:

HTTP Request

GET Path: / Content encoding:

☐ Redirect Automatically ✓ Follow Redirects ✓ Use KeepAlive ☐ Use multipart/form-data ☐ Browser-compatible headers

Send Parameters With the Request:

Name:	Value	URL Encode?	Content-Type	Include Equals?

Components of JMeter

- Test Plan: Root node of a test plan.
- Thread Group: Contains samplers, logic controllers, listeners, timers, configuration elements, and assertions.
- Sampler: Sends requests to a server.
- Listener: Processes and displays test results.
- Timer: Controls the delay between requests.
- Configuration Element: Provides configuration information.
- Assertion: Checks the response data.
- Controller: Controls the order of samplers.

Invoking JMeter from batch file

Create a batch file with the following content:

Code snippet

```
@echo off
set JMETER_HOME=C:\path\to\jmeter
set PATH=%JMETER_HOME%\bin;%PATH%
%JMETER_HOME%\bin\jmeter -n -t test.jmx -l results.jtl
```

Replace C:\path\to\jmeter with your JMeter installation directory.

Invoking JMeter from jar file

Use the following command in a command prompt:

Bash

```
java -jar "C:\path\to\apache-jmeter-x.x\bin\apache-jmeter.jar" -n -t test.jmx -l results.jtl
```

Replace C:\path\to\apache-jmeter-x.x with your JMeter installation directory.

Thread Group:

What is Thread Group

A Thread Group is a fundamental element in performance testing that represents a group of users performing a specific set of actions in your application. Each thread in a thread group simulates one user's actions.

Thread Group Properties

Key properties of a Thread Group include:

- Number of Threads (users)
- Ramp-up period
- Loop count
- Scheduler

JMeter GUI Basic Thread Group:

1. Open JMeter and create a new Test Plan
2. Right-click on the Test Plan > Add > Threads (Users) > Thread Group
3. In the Thread Group control panel, set:

```
Number of Threads (users): e.g., 100
Ramp-up period (seconds): e.g., 60
Loop Count: e.g., 10
```

setUp and tearDown Thread Group

These are special thread groups used for setup and cleanup operations.

- setUp Thread Group: Runs before the main test
- tearDown Thread Group: Runs after the main test

JMeter GUI setUp and tearDown Thread Groups:

1. Right-click on the Test Plan > Add > Threads (Users) > setUp Thread Group
2. Configure as needed (often with just 1 thread and 1 loop)
3. Add any setup actions (like database setup) to this group

4. Similarly, add a tearDown Thread Group for cleanup actions

Ultimate Thread Group

This is a more advanced thread group (available as a JMeter plugin) that allows for complex load scenarios. It provides fine-grained control over the number of threads, start time, hold-load time, and shutdown time.

JMeter GUI Ultimate Thread Group:

1. Ensure you have the JMeter Plugins Manager installed
2. Right-click on the Test Plan > Add > Threads (Users) > Ultimate Thread Group
3. In the Ultimate Thread Group control panel:
    a. Click "Add Row"
    b. Set "Start Threads Count", "Initial Delay", "Startup Time", "Hold Load For", "Shutdown Time"
    c. Add more rows for complex scenarios

Concurrency Thread Group

Another advanced thread group (also a JMeter plugin) that maintains a constant concurrency level, automatically adjusting the number of active threads based on the application's response times.

JMeter GUI Concurrency Thread Group:

1. Ensure you have the JMeter Plugins Manager installed
2. Right-click on the Test Plan > Add > Threads (Users) > Concurrency Thread Group
3. In the Concurrency Thread Group control panel, set:

```
Target Concurrency: e.g., 100
Ramp Up Time: e.g., 300
Ramp-Up Steps Count: e.g., 5
Hold Target Rate Time: e.g., 600
```

Samplers

Introduction to Samplers

A sampler in JMeter is a component that sends requests to a server and waits for a response. The request can be of different types like HTTP, FTP, JDBC, etc. Depending on the response, the sampler will determine if the test was successful or failed.

HTTP Request Sampler

- Description: Sends an HTTP/HTTPS request to a web server.
- Usage: Often used for testing web applications.

Examples:

GET Request:

```
URL: http://example.com
Method: GET
Parameters: None
```

POST Request:

```
URL: http://example.com/login
Method: POST
Parameters: username=user&password=pass
```

PUT Request:

```
URL: http://example.com/api/resource/1
Method: PUT
Body Data: {"name": "new name", "value": "new value"}
```

Debug Sampler

- Description: Used to generate a sample containing the values of JMeter variables.
- Usage: Helpful for debugging tests by displaying variable values.

Examples:

Display All Variables:

```
Variables: All defined variables in the test plan.
```

Display Specific Variable:

```
Variables: username
```

Display System Properties:

```
System Properties: All system properties.
```

Flow Control Action Sampler

- Description: Allows you to control the flow of the test.
- Usage: Useful for pausing the test or stopping it based on conditions.

Examples:

Pause for 5000ms:

```
Action: Pause
Duration: 5000ms
```

Stop Test Now:

```
Action: Stop Test Now
```

Go to Next Iteration:

```
Action: Go to Next Iteration
```

JSR223 Sampler

- Description: Executes scripts written in Groovy, JavaScript, or other languages supported by JSR223.
- Usage: Provides flexibility for custom scripting.

Examples:

Groovy Script for Custom Logic:
groovy

```
log.info("Executing custom logic");
```

JavaScript for Simple Calculation:
javascript

```
var result = 5 + 10;
log.info("Result: " + result);
```

Groovy Script to Set Variable:
groovy

```
vars.put("myVar", "myValue");
```

AJP/1.3 Sampler

- Description: Sends requests to a web server using the AJP protocol.
- Usage: Used for testing web servers that support AJP/1.3.

Examples:

Simple GET Request:

```
Server: ajp://localhost:8009
Path: /
```

POST Request with Parameters:

```
Server: ajp://localhost:8009
Path: /submit
Parameters: name=John&age=30
```

GET Request with Headers:

```
Server: ajp://localhost:8009
Path: /info
Headers: User-Agent: JMeter
```

Access Log Sampler

- Description: Parses access logs and generates HTTP requests.
- Usage: Replays recorded HTTP traffic.

Examples:

Apache Access Log:

```
Log File: /var/log/apache2/access.log
Format: Common Log Format
```

Custom Log Format:

```
Log File: /var/log/custom/access.log
Format: %h %l %u %t \"%r\" %>s %b
```

Nginx Access Log:

```
Log File: /var/log/nginx/access.log
Format: Combined Log Format
```

BeanShell Sampler

- Description: Executes scripts written in BeanShell, a scripting language for Java.
- Usage: Allows for complex scripting and logic.

Examples:

Print a Message:
java

```
log.info("Hello, BeanShell!");
```

Set a Variable:
java

```
vars.put("myVar", "myValue");
```

Perform a Calculation:
java

```
int sum = 5 + 10;
log.info("Sum: " + sum);
```

Bolt Request

- Description: Sends requests to a Neo4j graph database using the Bolt protocol.
- Usage: Used for testing Neo4j database performance.

Examples:

Simple Query:

```
Query: 'MATCH (n) RETURN n LIMIT 10'
```

Create a Node:

```
Query: 'CREATE (n:Person {name: 'John'}) RETURN n'
```

Update a Node:

```
Query: 'MATCH (n:Person {name: 'John'}) SET n.age = 30 RETURN n'
```

FTP Request

- Description: Sends FTP commands to an FTP server.
- Usage: Tests file transfer operations.

Examples:

Upload a File:

```
Server: ftp.example.com
Command: PUT
Local File: /path/to/local/file.txt
Remote File: /path/to/remote/file.txt
```

Download a File:

```
Server: ftp.example.com
Command: GET
Remote File: /path/to/remote/file.txt
Local File: /path/to/local/file.txt
```

List Files:

```
Server: ftp.example.com
Command: LS
```

GraphQL HTTP Request

- Description: Sends GraphQL queries over HTTP.
- Usage: Tests GraphQL APIs.

Examples:

Simple Query:

```
URL: http://example.com/graphql
Query: { user(id: 1) { name } }
```

Mutation:

```
URL: http://example.com/graphql
Query: mutation { createUser(name: "John") { id name } }
```

Query with Variables:

```
URL: http://example.com/graphql
Query: query($id: ID!) { user(id: $id) { name } }
Variables: { "id": 1 }
```

JDBC Request

- Description: Sends SQL queries to a database.
- Usage: Tests database performance and functionality.

Examples:

Select Query:

```
Database: MySQL
Query: SELECT * FROM users
```

Insert Query:

```
Database: MySQL
Query: INSERT INTO users (name, age) VALUES ('John', 30)
```

Update Query:

```
Database: MySQL
Query: UPDATE users SET age = 31 WHERE name = 'John'
```

JMS Point-to-Point

- Description: Sends and receives messages using the JMS (Java Messaging Service) API.
- Usage: Tests point-to-point messaging systems.

Examples:

Send Message:

```
Queue: testQueue
Message: Hello, JMS!
```

Receive Message:

```
Queue: testQueue
```

Send and Receive:

```
Queue: testQueue
Message: Hello, JMS!
```

JMS Subscriber

- Description: Subscribes to a JMS topic and receives messages.
- Usage: Tests publish/subscribe messaging systems.

Examples:

Subscribe to Topic:

```
Topic: testTopic
```

Receive Message:

```
Topic: testTopic
```

Filter Messages:

```
Topic: testTopic
Selector: JMSCorrelationID='12345'
```

JUnit Request

- Description: Executes JUnit test cases.
- Usage: Integrates JUnit tests into JMeter.

Examples:

Run Single Test Case:

```
Class Name: com.example.TestClass
Method Name: testMethod
```

Run All Tests in Class:

```
Class Name: com.example.TestClass
```

Run Multiple Test Cases:

```
Class Names: com.example.TestClass1, com.example.TestClass2
```

Java Request

- Description: Executes a Java class that implements the JavaSamplerClient interface.
- Usage: Allows for custom Java code execution.

Examples:

Custom Class Execution:

```
Class Name: com.example.MySampler
Parameters: param1=value1
```

Complex Logic:

```
Class Name: com.example.MyComplexSampler
Parameters: param1=value1&param2=value2
```

Database Operations:

```
Class Name: com.example.DBOperationSampler
Parameters: dbUrl=jdbc:mysql://localhost/test&query=SELECT * FROM users
```

LDAP Extended Request

- Description: Sends LDAP (Lightweight Directory Access Protocol) requests.
- Usage: Tests LDAP directory services.

Examples:

Add Entry:

```
Server: ldap://localhost
DN: cn=John Doe,dc=example,dc=com
Attributes: cn=John Doe, sn=Doe, objectClass=inetOrgPerson
```

Modify Entry:

```
Server: ldap://localhost
DN: cn=John Doe,dc=example,dc=com
Modifications: replace: sn Doe
```

Delete Entry:

```
Server: ldap://localhost
DN: cn=John Doe,dc=example,dc=com
```

LDPA Request

- Description: Sends standard LDAP requests.
- Usage: Tests LDAP directory services.

Examples:

Search Entries:

```
Server: ldap://localhost
Base DN: dc=example,dc=com
Filter: (objectClass=inetOrgPerson)
```

Bind Request:

```
Server: ldap://localhost
DN: cn=admin,dc=example,dc=com
Password: adminpassword
```

Compare Request:

```
Server: ldap://localhost
DN: cn=John Doe,dc=example,dc=com
Attribute: cn=John Doe
```

Mail Reader Sampler

- Description: Reads emails from an email server using POP3, IMAP, or SMTP protocols.
- Usage: Tests email server functionality.

Examples:

Read from IMAP Server:

```
Server: imap.example.com
Protocol: IMAP
Username: user@example.com
Password: password
```

Read from POP3 Server:

```
Server: pop3.example.com
Protocol: POP3
Username: user@example.com
Password: password
```

Read from SMTP Server:

```
Server: smtp.example.com
Protocol: SMTP
Username: user@example.com
Password: password
```

OS Process Sampler

- Description: Executes a command on the local system.
- Usage: Tests the performance of operating system commands.

Examples:

Run Shell Script:

```
Command: /path/to/script.sh
```

Execute Command:

```
Command: ls -l /var/log
```

Run Batch File:

```
Command: C:\\path\\to\\script.bat
```

SMTP Sampler

- Description: Sends email messages using the SMTP protocol.
- Usage: Tests email sending functionality.

Examples:

Send Plain Text Email:

```
Server: smtp.example.com
To: recipient@example.com
From: sender@example.com
Subject: Test Email
Body: This is a test email.
```

Send HTML Email:

```
Server: smtp.example.com
To: recipient@example.com
From: sender@example.com
Subject: HTML Test Email
Body: <html><body><h1>Hello</h1><p>This is a test email.</p></body></html>
```

Send Email with Attachment:

```
Server: smtp.example.com
To: recipient@example.com
From: sender@example.com
Subject: Email with Attachment
Body: Please see the attached file.
Attachment: /path/to/attachment.txt
```

TCP Sampler

- Description: Sends TCP requests to a server.
- Usage: Tests TCP-based services.

Examples:

Send Simple Message:

```
Server: tcp.example.com
Port: 12345
Message: Hello, TCP!
```

Send JSON Data:

```
Server: tcp.example.com
Port: 12345
Message: {"type": "greeting", "message": "Hello, TCP!"}
```

Receive Response:

```
Server: tcp.example.com
Port: 12345
Message: ping
Expected Response: pong
```

Controllers

What are Logical Controllers

Logical Controllers in JMeter determine the order in which samplers are executed. They can be used to create complex test plans by controlling the flow of requests.

If controller

Purpose: Executes child samplers based on a condition.

Example 1: Check if a user is logged in before proceeding to other actions.

1. Add an If Controller to your Thread Group.
2. Add a JSR223 PreProcessor to set a variable (e.g., loggedIn) to true or false based on a condition.
3. Add child samplers under the If Controller, with the condition set to "${loggedIn} == 'true'".

Example 2: Check Response Text

1. Add an If Controller to your Thread Group.
2. Add an HTTP Request sampler as a child of the If Controller.
3. Add a Response Assertion to check if the response contains a specific text.
4. In the If Controller's condition field, use the ${JMeterThread.last_sample_ok} variable and a custom condition based on the assertion result.
5. Add samplers that depend on the response text under the If Controller.

Transaction controller

Purpose: Groups samplers into a transaction for performance measurement.

Example 1: Measuring Login Time

1. Add a Transaction Controller to your Thread Group.
2. Give it a name, e.g., "Login Transaction".
3. Add samplers related to the login process (e.g., navigate to login page, submit login form) as children of the Transaction Controller.
4. JMeter will calculate the total time taken for all samplers within the transaction.

Example 2: Grouping API Calls

1. Add a Transaction Controller to your Thread Group.
2. Give it a name, e.g., "API Calls".
3. Add multiple HTTP Request samplers for different API endpoints as children of the Transaction Controller.
4. JMeter will measure the overall performance of the API calls.

Example 3: Checkout Process Time

1. Add a Transaction Controller to your Thread Group.
2. Give it a name, e.g., "Checkout Process".
3. Add samplers representing different steps of the checkout process (e.g., add items to cart, proceed to checkout, payment, confirmation) as children of the Transaction Controller.
4. JMeter will calculate the total time taken for the entire checkout process.

Loop controller

Purpose: Repeats child samplers a specified number of times.

Example 1: Iterative Login Attempts

1. Add a Loop Controller to your Thread Group.
2. Set the Loop Count to the desired number of attempts.
3. Add an HTTP Request sampler for login as a child of the Loop Controller.

Example 2: Data-Driven Testing

1. Add a Loop Controller to your Thread Group.
2. Set the Loop Count to the number of data sets.
3. Add a CSV Data Set Config as a child of the Loop Controller to provide data.
4. Use variables from the CSV Data Set in your child samplers.

Example 3: Error Handling Retry

1. Add a Loop Controller with a counter.
2. Add an HTTP Request sampler as a child of the Loop Controller.
3. Add a Response Assertion to check for errors.
4. Use a JSR223 PostProcessor to increment the counter if an error occurs.
5. Set the Loop Controller's condition to check if the counter is less than a maximum retry count.

While Controller

Purpose: Continuously executes child samplers until a condition is met.

Example 1: Retry Failed Login

1. Add a While Controller to your Thread Group.
2. Set the condition to "${__javaScript("${counter}" < 3)}".
3. Create a user-defined variable named "counter" and initialize it to 0.
4. Add a JSR223 PostProcessor to increment the counter if the login fails.
5. Add an HTTP Request sampler for login as a child of the While Controller.

Example 2: Data-Driven Testing with Loop

6. Add a While Controller to your Thread Group.
7. Set the condition to "${__javaScript("${counter}" < ${dataCount})}".
8. Create user-defined variables "counter" and "dataCount".
9. Add a CSV Data Set Config to provide data.
10. Add samplers that use data from the CSV as children of the While Controller.
11. Increment the "counter" variable in a JSR223 PostProcessor.

Example 3: Continuous Load Generation

12. Add a While Controller with no condition (infinite loop).
13. Add samplers to simulate user actions as children of the While Controller.
14. Use other controllers (e.g., Runtime Controller) to control the overall test duration.

Critical Section Controller

Purpose: Ensures that only one thread executes child samplers at a time.

Example 1: Shared Resource Access

1. Add a Critical Section Controller to your Thread Group.

2. Add samplers that access a shared resource (e.g., database update) as children of the Critical Section Controller.
3. Ensure data consistency by preventing multiple threads from accessing the resource simultaneously.

Example 2: Synchronized Test Steps
1. Add a Critical Section Controller to your Thread Group.
2. Add samplers that must be executed sequentially (e.g., login, create account, logout) as children of the Critical Section Controller.
3. Guarantee the correct order of execution for these steps.

Example 3: Limited Resource Simulation
1. Add a Critical Section Controller to your Thread Group.
2. Add samplers that simulate access to a limited resource (e.g., server capacity).
3. Control the number of concurrent users accessing the resource.

ForEach Controller

Purpose: Iterates over a list of values and executes child samplers for each value.
Example 1: Send requests to multiple URLs.
1. Add a ForEach Controller to your Thread Group.
2. Define a variable name (e.g., url) and a list of values (e.g., ["url1", "url2", "url3"]).
3. Add an HTTP Request sampler under the ForEach Controller, using the variable ${url} as the path.

Example 2: Multiple User Logins
1. Add a ForEach Controller to your Thread Group.
2. Define a variable name (e.g., username) and a list of user names.
3. Add an HTTP Request sampler for login as a child of the ForEach Controller, using the ${username} variable.

Example 3: API Testing with Multiple Parameters
4. Add a ForEach Controller to your Thread Group.
5. Define a variable name (e.g., parameter) and a list of parameter values.
6. Add an HTTP Request sampler as a child of the ForEach Controller, using the ${parameter} variable in the request body or URL.

Example 4: Database Queries with Multiple Values
7. Add a ForEach Controller to your Thread Group.
8. Define a variable name (e.g., value) and a list of values to query.
9. Add a JDBC Request sampler as a child of the ForEach Controller, using the ${value} variable in the SQL query.

Include Controller

Purpose: Includes the contents of another test plan or test fragment.

Example 1: Reusable Test Logic
1. Create a separate test plan with common test steps.
2. Add an Include Controller to your main test plan.
3. Set the path to the external test plan in the Include Controller.
4. Reuse the included test logic in multiple test plans.

Example 2: Test Plan Organization
1. Divide a large test plan into smaller, more manageable test fragments.
2. Use Include Controllers to incorporate these fragments into the main test plan.
3. Improve test plan readability and maintainability.

Example 3: Sharing Test Data
1. Create a test fragment with configuration elements (e.g., CSV Data Set Config) to provide test data.
2. Include this test fragment in multiple test plans to share the same data.

Interleave Controller

Purpose: Executes child samplers in a round-robin fashion.

Example 1: Load Balancing Simulation
1. Add an Interleave Controller to your Thread Group.
2. Add multiple HTTP Request samplers for different servers as children of the Interleave Controller.
3. Simulate load distribution across multiple servers.

Example 2: User Behavior Variation
1. Add an Interleave Controller to your Thread Group.
2. Add samplers representing different user actions (e.g., search, browse, purchase) as children of the Interleave Controller.
3. Simulate various user behaviors by randomly selecting samplers.

Example 3: Test Data Iteration
1. Add an Interleave Controller to your Thread Group.
2. Add multiple HTTP Request samplers with different data sets as children of the Interleave Controller.
3. Cycle through different data sets for each iteration.

Once Only Controller

Purpose: Executes child samplers only once, regardless of the number of iterations.

Example 1: Initialization Steps

1. Add a Once Only Controller to your Thread Group.
2. Add samplers for initialization steps (e.g., login, configuration) as children of the Once Only Controller.
3. Ensure these steps are executed only once per thread.

Example 2: Test Data Setup

1. Add a Once Only Controller to your Thread Group.
2. Add samplers to set up test data (e.g., create test users) as children of the Once Only Controller.
3. Prepare the test environment before running actual test cases.

Example 3: Global Variables

1. Add a Once Only Controller to your Thread Group.
2. Add a JSR223 PreProcessor to define global variables.
3. Make these variables accessible to all other samplers in the test plan.

Random Controller

Purpose: Executes child samplers in random order.

Example 1: User Behavior Variation

1. Add a Random Controller to your Thread Group.
2. Add samplers representing different user actions (e.g., search, browse, purchase) as children of the Random Controller.
3. Simulate unpredictable user behavior by randomly executing samplers.

Example 2: Test Data Randomization

1. Add a Random Controller to your Thread Group.
2. Add multiple HTTP Request samplers with different data sets as children of the Random Controller.
3. Randomly select data sets for each iteration.

Example 3: Load Distribution

1. Add a Random Controller to your Thread Group.
2. Add multiple HTTP Request samplers for different servers as children of the Random Controller.
3. Distribute load randomly across servers.

Random Order Controller

Purpose: Executes child samplers in a random order, but each sampler is executed only once.

Example 1: Randomizing Test Steps

1. Add a Random Order Controller to your Thread Group.
2. Add multiple HTTP Request samplers representing different test steps as children of the Random Order Controller.

3. Ensure that each test step is executed exactly once, but in a random order.

Example 2: Data-Driven Testing with Randomization

1. Add a Random Order Controller to your Thread Group.
2. Add multiple HTTP Request samplers with different data sets as children of the Random Order Controller.
3. Randomize the order in which data sets are processed.

Example 3: Load Distribution with Randomization

1. Add a Random Order Controller to your Thread Group.
2. Add multiple HTTP Request samplers for different servers as children of the Random Order Controller.
3. Distribute load across servers in a random order.

Recording Controller

Purpose: Used for recording test scripts.

Example 1: Basic Recording

1. Add a Recording Controller to your Thread Group.
2. Configure the HTTP(S) Test Script Recorder to capture traffic.
3. Start the recorder, perform actions in your browser, and stop the recorder.
4. Analyze the recorded samplers and modify them as needed.

Example 2: Parameterization

1. Record a test script with dynamic values (e.g., user names, passwords).
2. Use parameterization techniques (e.g., CSV Data Set Config, User Defined Variables) to replace dynamic values with variables.

Example 3: Correlation

1. Record a test script with dynamic values that change with each request.
2. Use regular expressions or other correlation techniques to extract dynamic values from previous responses and store them in variables.

Runtime Controller

Purpose: Executes child samplers for a specified duration.

Example 1: Load Testing for a Specific Duration

1. Add a Runtime Controller to your Thread Group.
2. Set the desired duration (e.g., 300 seconds).
3. Add samplers to simulate user load as children of the Runtime Controller.
4. Generate load for the specified duration.

Example 2: Stress Testing with Time Limit

1. Add a Runtime Controller to your Thread Group.
2. Set a short duration (e.g., 60 seconds) for a high load test.
3. Add samplers to simulate a large number of users as children of the Runtime Controller.
4. Test the system's behavior under extreme load for a limited time.

Example 3: User Behavior Simulation

1. Add a Runtime Controller to your Thread Group.
2. Set a duration that represents an average user session.
3. Add samplers to simulate user actions within that time frame.
4. Model realistic user behavior patterns.

Module controller

Purpose: Includes the contents of another test plan or test fragment.
Note: The Module Controller is essentially the same as the Include Controller.

Simple controller

Purpose: A container for samplers without any specific logic.

Example 1: Grouping Related Samplers

1. Add a Simple Controller to your Thread Group.
2. Give it a descriptive name (e.g., "Login Process").
3. Add samplers related to the login process as children of the Simple Controller.
4. Improve test plan organization and readability.

Example 2: Conditional Execution (using If Controller)

1. Add a Simple Controller to your Thread Group.
2. Add an If Controller as a child of the Simple Controller.
3. Use the If Controller to conditionally execute samplers based on certain conditions.

Example 3: Grouping Samplers for Listeners

1. Add a Simple Controller to your Thread Group.
2. Add samplers that you want to analyze together as children of the Simple Controller.
3. Apply listeners to the Simple Controller to view aggregated results for the group of samplers.

Throughput Controller

Purpose: Executes enough child samplers to reach a specified throughput.

Example 1: Maintaining Constant Throughput

1. Add a Throughput Controller to your Thread Group.
2. Set the desired throughput (e.g., 100 requests per minute).

3. Add samplers as children of the Throughput Controller.
4. JMeter will adjust the number of threads to achieve the target throughput.

Example 2: Simulating User Load

1. Add a Throughput Controller to your Thread Group.
2. Set the desired throughput based on expected user load.
3. Add samplers representing user actions as children of the Throughput Controller.
4. Simulate the expected user load on the system.

Example 3: Performance Testing with Throughput Goal

1. Add a Throughput Controller to your Thread Group.
2. Set a high throughput value to stress the system.
3. Add samplers as children of the Throughput Controller.
4. Monitor system performance under heavy load.

Switch Controller

Purpose: Selects a child sampler based on a value.

Example 1: Dynamic Test Cases

1. Add a Switch Controller to your Thread Group.
2. Define a variable to hold the switch value.
3. Add child samplers, each with a corresponding switch value.
4. The sampler with the matching switch value will be executed.

Example 2: Test Environment Selection

1. Add a Switch Controller to your Thread Group.
2. Define a variable to indicate the test environment (e.g., "dev", "test", "prod").
3. Add child samplers for each environment with different server configurations.
4. Select the appropriate environment based on the switch value.

Example 3: User Segmentation

1. Add a Switch Controller to your Thread Group.
2. Define a variable to represent user segments (e.g., "new_user", "returning_user").
3. Add child samplers with different user flows for each segment.
4. Simulate different user behaviors based on user segmentation.

Assertions

What is assertion in JMeter

Assertions in JMeter are used to validate that the responses received from the server match expected results. They help ensure that the application is functioning correctly under load.

Response Assertion

This is used to check for specific content in the response.

Example setup:

1. Right-click on HTTP Request > Add > Assertions > Response Assertion
2. Select "Text Response" in "Apply to"
3. Add a pattern to test, e.g., "Welcome to our website"
4. Choose "Contains" or "Matches" as appropriate

JSON assertion

Used to validate JSON responses.

Example setup:

1. Right-click on HTTP Request > Add > Assertions > JSON Assertion
2. In "Assert JSON Path exists", enter: $.name
3. In "Expected Value", enter: "John Doe"

Size Assertion

Checks if the response size is within a specified range.

Example setup:

1. Right-click on HTTP Request > Add > Assertions > Size Assertion
2. Set "Size Field to Test" to "Response Body"
3. Set size constraints, e.g., ">= 1000" and "<= 10000"

JSR223 Assertion

Allows you to write custom assertions using scripting languages.

Example setup (using Groovy):

1. Right-click on HTTP Request > Add > Assertions > JSR223 Assertion
2. Choose "groovy" as language
3. Enter script:

[groovy]

```
groovyCopyassert prev.getResponseDataAsString().contains("Expected Text")
```

XPath2 Assertion

Used for XML responses, allows XPath 2.0 expressions.

Example setup:

- Right-click on HTTP Request > Add > Assertions > XPath2 Assertion
- Enter XPath expression, e.g.: //book[author='Neal Stephenson']/title

Compare Assertion

Compares multiple responses.

Example setup:

1. Right-click on Thread Group > Add > Assertions > Compare Assertion
2. Select "Compare Content" and add the samplers to compare

Duration Assertion

Checks if the sample duration is within limits.

Example setup:

1. Right-click on HTTP Request > Add > Assertions > Duration Assertion
2. Set "Duration to assert" to, e.g., 1000 (milliseconds)

HTML Assertion

Checks for HTML syntax errors.

Example setup:

1. Right-click on HTTP Request > Add > Assertions > HTML Assertion
2. Select error reporting options as needed

JSON JMESPath Assertion

Uses JMESPath expressions to query JSON responses.

Example setup:

1. Right-click on HTTP Request > Add > Assertions > JSON JMESPath Assertion
2. Enter JMESPath expression, e.g.: people[?age > 30].name

MD5Hex Assertion

Checks if the MD5 hash of the response matches an expected value.

Example setup:

1. Right-click on HTTP Request > Add > Assertions > MD5Hex Assertion
2. Enter expected MD5 hash

SMIME Assertion

Verifies digital signatures in S/MIME encrypted responses.

Example setup:

1. Right-click on HTTP Request > Add > Assertions > SMIME Assertion
2. Configure with appropriate certificate details

XML Assertion

Checks if the response is well-formed XML.

Example setup:

1. Right-click on HTTP Request > Add > Assertions > XML Assertion
2. No additional configuration needed

XML Schema Assertion

Validates XML against a schema.

Example setup:

1. Right-click on HTTP Request > Add > Assertions > XML Schema Assertion
2. Provide path to XML Schema file

XPath Assertion

Uses XPath to validate XML responses.

Example setup:

1. Right-click on HTTP Request > Add > Assertions > XPath Assertion
2. Enter XPath expression, e.g.: //book[author='Jane Austen']/title

Beanshell Assertion

Allows custom assertions using Beanshell scripting.

Example setup:

1. Right-click on HTTP Request > Add > Assertions > Beanshell Assertion
2. Enter Beanshell script, e.g.:

```
javaCopyString response = new String(ResponseData);
if (!response.contains("Expected Text")) {
  Failure = true;
  FailureMessage = "Response doesn't contain expected text";
}
```

Note(s):

- ✓ You can add multiple assertions to a single sampler.
- ✓ Use the "View Results Tree" listener to debug assertion failures.
- ✓ Assertions can significantly impact test performance, so use them judiciously in large load tests.

Timers

What are Timers

Timers in JMeter are used to introduce delays between requests, simulating real user behavior and controlling the pace of the test. They're added to Thread Groups or individual samplers and affect all samplers that follow them in the scope.

Constant timer

This timer adds a constant delay between requests.

Example setup:

1. Right-click on Thread Group > Add > Timer > Constant Timer
2. Set "Thread Delay" to, e.g., 1000 (milliseconds)

Uniform Random timer

This timer adds a random delay within a specified range.

Example setup:

1. Right-click on Thread Group > Add > Timer > Uniform Random Timer
2. Set "Constant Delay Offset" to, e.g., 1000
3. Set "Random Delay Maximum" to, e.g., 2000

Precise Throughput timer

This timer tries to achieve a specified throughput by dynamically adjusting delays.

Example setup:

1. Right-click on Thread Group > Add > Timer > Precise Throughput Timer
2. Set "Target Throughput" to, e.g., 60
3. Set "Test Duration" to, e.g., 3600 (1 hour in seconds)

Constant throughput timer

Similar to Precise Throughput Timer, but uses a different algorithm.

Example setup:

1. Right-click on Thread Group > Add > Timer > Constant Throughput Timer
2. Set "Target throughput" to, e.g., 60.0
3. Select "calculate Throughput based on" as "all active threads"

JSR223 timer

Allows you to create custom timers using scripting languages.

Example setup (using Groovy):

1. Right-click on Thread Group > Add > Timer > JSR223 Timer
2. Choose "groovy" as language
3. Enter script:

```
groovyCopydef delay = 1000 + Math.random() * 2000
log.info("Waiting for ${delay} ms")
return delay.intValue()
```

Synchronizing timer

This timer blocks threads until a specified number of threads have been blocked, then releases them all at once.

Example setup:

1. Right-click on Thread Group > Add > Timer > Synchronizing Timer
2. Set "Number of Simulated Users to Group by" to, e.g., 10

Poisson Random timer

This timer generates random delays with a Poisson distribution.

Example setup:

1. Right-click on Thread Group > Add > Timer > Poisson Random Timer
2. Set "Lambda" to, e.g., 1000
3. Set "Constant Delay Offset" to, e.g., 500

BeanShell Timer

Allows you to create custom timers using BeanShell scripting.

Example setup:

1. Right-click on Thread Group > Add > Timer > BeanShell Timer
2. Enter BeanShell script, e.g.:

```
javaCopylong delay = 1000 + (long)(Math.random() * 2000);
log.info("Waiting for " + delay + " ms");
return delay;
```

Note(s):
Key points to remember when using timers:

- ✓ Timers are processed before each sampler in their scope.
- ✓ If there are multiple timers, all of their delays will be added together.
- ✓ To apply a timer to a specific sampler, add it as a child of that sampler.
- ✓ Use the "View Results Tree" listener to see the effect of timers on your requests.

Here's an example of how you might use timers in a test plan:

1. Create a Thread Group
2. Add an HTTP Request sampler
3. Add a Constant Timer as a child of the HTTP Request
    - Set Thread Delay to 1000 ms
4. Add another HTTP Request sampler
5. Add a Uniform Random Timer as a child of the second HTTP Request
    - Set Constant Delay Offset to 500 ms
    - Set Random Delay Maximum to 1500 ms

Correlation

What is Correlation

Correlation in performance testing refers to the process of capturing dynamic values from a server response and using them in subsequent requests. These dynamic values, such as session IDs, tokens, or unique identifiers, are often required to simulate real user behavior accurately.

How it is done in JMeter

In JMeter, correlation is achieved using post-processors. Post-processors are elements that execute actions after a sampler request has been made. They are used to extract data from responses and save it into variables, which can then be used in subsequent requests.

What are post processors in JMeter

Post processors in JMeter are elements that allow you to process the response data received from a server. They are added as children of samplers and are executed after the sampler has finished its request.

Introduction to Regular Expression Extractor

The Regular Expression Extractor is a post-processor in JMeter used to extract values from the response data using regular expressions. It allows you to define a pattern to match in the response, and the matched values are stored in JMeter variables.

Usage of Regular Expression Extractor

Example 1: Extracting a Session ID from an HTTP Response

Add an HTTP Request Sampler:

```
URL: http://example.com/login
Method: POST
Parameters: username=user&password=pass
```

Add a Regular Expression Extractor:

```
Apply to: Main sample only
Field to check: Response Body
Reference Name: sessionID
Regular Expression: session_id=(\w+)
Template: $1$
Default Value: NOT_FOUND
```

Use the Extracted Variable in Subsequent Request:

```
Add another HTTP Request Sampler.
URL: http://example.com/dashboard
Parameters: session_id=${sessionID}
```

Example 2: Extracting a Token from a JSON Response

Add an HTTP Request Sampler:

```
URL: http://example.com/api/authenticate
Method: POST
Parameters: username=user&password=pass
```

Add a Regular Expression Extractor:

```
Apply to: Main sample only
Field to check: Response Body
Reference Name: authToken
Regular Expression: "token":"(\w+)"
Template: $1$
Default Value: NOT_FOUND
```

Use the Extracted Token in Subsequent Request:

```
Add another HTTP Request Sampler.
URL: http://example.com/api/data
Header: Authorization: Bearer ${authToken}
```

Example 3: Extracting a Dynamic Value from HTML Response

Add an HTTP Request Sampler:

```
URL: http://example.com/form
Method: GET
```

Add a Regular Expression Extractor:

```
Apply to: Main sample only
Field to check: Response Body
Reference Name: formToken
Regular Expression: <input type="hidden" name="token" value="(\w+)"/>
Template: $1$
Default Value: NOT_FOUND
```

Use the Extracted Value in Subsequent Request:

```
Add another HTTP Request Sampler.
URL: http://example.com/submit
Method: POST
Parameters: token=${formToken}&field1=value1
```

Usage of Boundary value Extractor

The Boundary Extractor is another post-processor in JMeter used to extract values from the response data using boundary conditions. It captures the data located between specified left and right boundaries.

Example 1: Extracting a Session ID Using Boundary Extractor

Add an HTTP Request Sampler:

```
URL: http://example.com/login
Method: POST
Parameters: username=user&password=pass
```

Add a Boundary Extractor:

```
Apply to: Main sample only
Field to check: Response Body
Reference Name: sessionID
Left Boundary: session_id=
Right Boundary: ;
Default Value: NOT_FOUND
```

Use the Extracted Variable in Subsequent Request:

```
Add another HTTP Request Sampler.
URL: http://example.com/dashboard
Parameters: session_id=${sessionID}
```

Example 2: Extracting a Token from an XML Response

Add an HTTP Request Sampler:

```
URL: http://example.com/api/authenticate
Method: POST
Parameters: username=user&password=pass
```

Add a Boundary Extractor:

```
Apply to: Main sample only
Field to check: Response Body
Reference Name: authToken
Left Boundary: <token>
Right Boundary: </token>
Default Value: NOT_FOUND
```

Use the Extracted Token in Subsequent Request:

```
Add another HTTP Request Sampler.
URL: http://example.com/api/data
Header: Authorization: Bearer ${authToken}
```

Example 3: Extracting a Dynamic Value from a Text Response

Add an HTTP Request Sampler:

```
URL: http://example.com/form
Method: GET
```

Add a Boundary Extractor:

```
Apply to: Main sample only
Field to check: Response Body
Reference Name: formToken
Left Boundary: token:
Right Boundary: ;
Default Value: NOT_FOUND
```

Use the Extracted Value in Subsequent Request:

```
Add another HTTP Request Sampler.
URL: http://example.com/submit
Method: POST
Parameters: token=${formToken}&field1=value1
```

Config Element

What are Config Elements

Config Elements are components in JMeter that set up variables and default values for use in your test plan. They help in making your tests more flexible and reusable.

What is parameterization

Parameterization is the practice of replacing hard-coded values with variables. This allows for easier maintenance, data-driven testing, and more dynamic test scenarios.

Demo: Parameterize a simple HTTP request using a CSV Data Set Config element in JMeter

1. Prepare Your CSV File

Create a CSV file (e.g., user_data.csv) with the following content:

```
username,password
user1,pass1
user2,pass2
user3,pass3
```

2. Add a CSV Data Set Config Element
    a. Open JMeter and create a new Test Plan.
    b. Right-click on the Test Plan and add a Thread Group:
    c. Add > Threads (Users) > Thread Group
    d. Right-click on the Thread Group and add a CSV Data Set Config:
    e. Add > Config Element > CSV Data Set Config
    f. Configure the CSV Data Set Config:

```
Filename: Path to your CSV file (e.g., /path/to/user_data.csv)
Variable Names: username,password
Delimiter: ,
Recycle on EOF: True
Stop thread on EOF: False
Sharing mode: All threads
```

3. Add an HTTP Request Sampler
    a. Right-click on the Thread Group and add an HTTP Request sampler:
    b. Add > Sampler > HTTP Request
    c. Configure the HTTP Request:

```
Name: Login Request
Server Name or IP: example.com
Method: POST
Path: /login
Parameters:
Name: username
Value: ${username} (This references the username variable from the CSV file)
Add another parameter:
Name: password
Value: ${password} (This references the password variable from the CSV file)
```

4. Add a Listener to View Results
    a. Right-click on the Thread Group and add a View Results Tree listener:
    b. Add > Listener > View Results Tree

5. Run the Test
    a. Save your Test Plan.
    b. Click the green Start button (or press Ctrl+R) to run the test.
    c. View the results in the View Results Tree listener to see the responses for each HTTP request made using different username and password pairs from the CSV file.

CSV Data Set Config

Example:

1. Create a CSV file named "test_data.csv" with content:

```
username,password,email
john_doe,pass123,john@example.com
jane_smith,secret456,jane@example.com
```

2. In JMeter GUI:
   a. Right-click on Thread Group > Add > Config Element > CSV Data Set Config
   b. Configure as follows:

   ```
   Filename: path/to/test_data.csv
   Variable Names: username,password,email
   Delimiter (use '\t' for Tab): ,
   Recycle on EOF: True
   Stop thread on EOF: False
   ```

   c. Add an HTTP Request sampler

   ```
   In the HTTP Request, use ${username}, ${password}, and ${email} as parameters
   ```

This setup allows you to run tests with multiple sets of user data easily.

HTTP Cookie Manager

Example:

1. Right-click on Test Plan > Add > Config Element > HTTP Cookie Manager
2. Leave default settings
3. Add an HTTP Request for login:

```
Server Name: example.com
Path: /login
Method: POST

Parameters:
username: ${username}
password: ${password}
```

4. Add another HTTP Request for a secured page:

```
Server Name: example.com
Path: /dashboard
```

The Cookie Manager will automatically handle cookies, maintaining the session between requests.

HTTP Cache Manager

Example:

1. Right-click on Thread Group > Add > Config Element > HTTP Cache Manager
2. Leave default settings
3. Add multiple HTTP Requests for static resources:

```
/styles/main.css
/scripts/app.js
/images/logo.png
```

The Cache Manager will simulate browser caching, potentially reducing server load during testing.

HTTP Request Default

Example:

1. Right-click on Thread Group > Add > Config Element > HTTP Request Defaults
2. Configure as follows:

```
Protocol: https
Server Name: api.example.com
Path: /v1
```

3. Add HTTP Request samplers:

```
Path: /users (full path will be https://api.example.com/v1/users)
Path: /products (full path will be https://api.example.com/v1/products)
```

This reduces repetition in your test plan and makes it easier to change common settings.

User defined Variable

Example:

1. Right-click on Test Plan > Add > Config Element > User Defined Variables
2. Add the following variables:

```
Name: base_url, Value: https://example.com
Name: api_key, Value: abc123xyz789
Name: timeout, Value: 5000
```

3. In HTTP Request samplers, use:

```
Server Name: ${base_url}
Add to URL: api_key=${api_key}
```

4. In Timers, use:

```
Thread Delay: ${timeout}
```

This makes it easy to change these values in one place for the entire test.

Listeners

What are Listeners

Listeners in JMeter are components that provide different ways to view the results of your test execution. They allow you to analyze the data collected during a test run, helping you understand the performance of your application under test.

View Result Tree

The View Result Tree is one of the most commonly used listeners. It shows detailed results for each sampler request in your test.

Example:

1. Create a Thread Group
2. Add an HTTP Request sampler (e.g., to http://example.com)
3. Right-click on Thread Group > Add > Listener > View Results Tree
4. Run the test
5. In the View Results Tree, you'll see:
    - Request data
    - Response data
    - Response headers
    - Request headers
    - Response code
    - Response message

    This listener is particularly useful for debugging your test plan and checking individual responses.

Aggregate Report

The Aggregate Report provides statistical information about the test results.

Example:

1. Create a Thread Group with multiple HTTP Request samplers
2. Right-click on Thread Group > Add > Listener > Aggregate Report
3. Run the test
4. The Aggregate Report will show:
    - Sample count
    - Average response time
    - Median response time
    - 90% line (90th percentile of response time)
    - Min and Max response times
    - Error rate
    - Throughput (requests per second)

    This listener is excellent for getting an overview of your test performance.

Summary Report

The Summary Report is similar to the Aggregate Report but with less detail, making it lighter on resources.

Example:

1. Create a Thread Group with multiple HTTP Request samplers
2. Right-click on Thread Group > Add > Listener > Summary Report
3. Run the test
4. The Summary Report will show:
    - Sample count
    - Average response time
    - Min and Max response times
    - Error rate
    - Throughput (requests per second)

    This listener is useful for quick performance overviews, especially in longer running tests.

Script Recording and Replay in Jmeter

Script Recording in JMeter allows you to capture HTTP/HTTPS traffic between a browser and a web server, creating a test plan automatically.

Adding necessary elements in JMeter

Example: Basic Recording Setup

1. Start JMeter
2. Right-click on Test Plan > Add > Threads (Users) > Thread Group
3. Right-click on Thread Group > Add > Logic Controller > Recording Controller
4. Right-click on Test Plan > Add > Non-Test Elements > HTTP(S) Test Script Recorder

Recording Settings in JMeter

Configuring HTTP(S) Test Script Recorder

1. Double-click on HTTP(S) Test Script Recorder
2. Set Port to 8888 (or any free port)
3. In "URL Patterns to Include", add ..html. to record only HTML requests
4. Click "Add" in the "Certificate" section to create a JMeter CA certificate

Browser Configuration for Recording

For this example, we'll use Firefox:

1. Open Firefox
2. Go to Options > Network Settings
3. Configure manual proxy:

```
HTTP Proxy: localhost
Port: 8888 (the port you set in JMeter)
```

4. Check "Use this proxy server for all protocols"

Recording the test steps using JMeter on sample site

Recording a login sequence on a sample site

1. In JMeter, click "Start" on the HTTP(S) Test Script Recorder
2. In Firefox, navigate to your sample site (e.g., http://example.com)
3. Perform a login sequence:
    a. Click on "Login"
    b. Enter username and password
    c. Click "Submit"
4. In JMeter, click "Stop" on the HTTP(S) Test Script Recorder

You should now see recorded requests under the Recording Controller

Recording think times

To record think times:

1. In HTTP(S) Test Script Recorder, check "Think Time" under "Record Time"
2. Set a threshold (e.g., 1000 ms)
3. During recording, pause between actions to simulate user think time
4. JMeter will add Constant Timer elements to your test plan

Filtering option in JMeter

Using filters to exclude specific requests

1. In HTTP(S) Test Script Recorder
2. In "URL Patterns to Exclude", add:

```
..png.
..jpg.
..js.
..css.
```

3. This will exclude requests for images, JavaScript, and CSS files

How to replay a script in JMeter

To replay your recorded script:

1. Ensure your Thread Group is properly configured (e.g., Number of Threads, Ramp-up Period)
2. Add listeners (e.g., View Results Tree, Summary Report)

3. Click the "Start" button (green play button) in JMeter's toolbar
4. View results in your added listeners

Recording via Blazemeter

BlazeMeter provides a browser extension for easy recording:

1. Install the BlazeMeter Extension for Chrome or Firefox
2. Click the BlazeMeter icon and start recording
3. Perform your test scenario in the browser
4. Stop the recording
5. Export the recorded script as a JMX file
6. Import the JMX file into JMeter (File > Open)

API Testing with JMeter

What is API testing?

API testing involves testing the functionality of application programming interfaces (APIs) to ensure they meet expectations in terms of performance, reliability, and security. It focuses on validating the API's behavior, response formats, error handling, and overall performance.

How it can be done in JMeter?

JMeter provides several components to effectively perform API testing:

- HTTP Request sampler: The core component for making API calls. You can configure the HTTP method (GET, POST, PUT, DELETE), request headers, parameters, and body.
- Assertions: Validate the API response against expected values. JMeter offers various assertion types like Response Assertion, JSON Assertion, XPath Assertion, etc.
- Listeners: View test results in different formats. Common listeners include View Results Tree, Summary Report, Graph Results, etc.
- Config Elements: Provide additional configuration information. HTTP Header Manager, HTTP Cookie Manager, and User Defined Variables are commonly used.
- Preprocessors and Postprocessors: Execute actions before or after a sampler. They can be used for dynamic data handling, correlation, and other tasks.

Example 1: GET Request - Fetching User Data
This example will demonstrate how to test a GET request to retrieve user data from a public API.

Steps:

1. Start JMeter and create a new Test Plan
2. Right-click on Test Plan > Add > Threads (Users) > Thread Group
3. Configure Thread Group:

```
Number of Threads (users): 1
Ramp-up period: 1
```

```
Loop Count: 1
```

4. Right-click on Thread Group > Add > Sampler > HTTP Request
5. Configure HTTP Request:

```
Name: Get User Data
Method: GET
Protocol: https
Server Name: jsonplaceholder.typicode.com
Path: /users/1
```

6. Right-click on Thread Group > Add > Listener > View Results Tree
7. Right-click on HTTP Request > Add > Assertions > JSON Assertion
8. Configure JSON Assertion:

```
Assert JSON Path exists: $.name
Expected Value: Leanne Graham
```

9. Run the test by clicking the green "Start" button
10. Check the results in the View Results Tree

Example 2: POST Request - Creating a New Resource
This example will demonstrate how to test a POST request to create a new resource.

Steps:

1. In your existing Test Plan, right-click on Thread Group > Add > Sampler > HTTP Request
2. Configure HTTP Request:

```
Name: Create Post
Method: POST
Protocol: https
Server Name: jsonplaceholder.typicode.com
Path: /posts
```

3. In the HTTP Request, go to the "Body Data" tab and add:

   json

```
{
  "title": "foo",
  "body": "bar",
  "userId": 1
}
```

4. Add an HTTP Header Manager:
   Right-click on the HTTP Request > Add > Config Element > HTTP Header Manager

5. Add header:

```
Name: Content-type
Value: application/json; charset=UTF-8
```

6. Right-click on HTTP Request > Add > Assertions > JSON Assertion
7. Configure JSON Assertion:

```
Assert JSON Path exists: $.id
```

8. Run the test and check the results in the View Results Tree

Example 3: PUT Request - Updating a Resource
This example demonstrates updating an existing resource using a PUT request.

Steps:

1. Create a new Thread Group
2. Add an HTTP Request:

```
Name: Update User
Method: PUT
Protocol: https
Server Name: reqres.in
Path: /api/users/2
```

   Body Data:

   json

```
{
  "name": "morpheus",
  "job": "zion resident"
}
```

3. Add HTTP Header Manager:

```
Name: Content-type
Value: application/json
```

4. Add JSON Assertion:

```
Assert JSON Path exists: $.updatedAt
```

5. Add View Results Tree listener
6. Run the test and check results

Example 4: DELETE Request - Removing a Resource
This example shows how to test a DELETE request.

Steps:

1. Add a new HTTP Request to your Thread Group:

```
Name: Delete User
Method: DELETE
Protocol: https
Server Name: reqres.in
Path: /api/users/2
```

2. Add Response Assertion:

```
Apply to: Main sample and sub-samples
Pattern Matching Rules: Response Code
Patterns to Test: 204
```

3. Add View Results Tree listener
4. Run the test and verify the response code

Example 5: Testing API Rate Limiting
This example demonstrates how to test API rate limiting.

Steps:

1. Create a new Thread Group:

```
Number of Threads (users): 10
Ramp-up period: 1
Loop Count: 5
```

2. Add an HTTP Request:

```
Name: Rate Limited API
Method: GET
Protocol: https
Server Name: api.github.com
Path: /users/octocat
```

3. Add Response Headers to JSON Extractor:

```
Names of created variables: rate_limit_remaining
JSON Path expressions: X-RateLimit-Remaining
```

4. Add If Controller:

```
Condition: ${__jexl3(${rate_limit_remaining} < 5)}
```

5. Under If Controller, add a Test Action:

```
Action: Pause
Duration: 5000 (5 seconds)
```

6. Add View Results Tree listener
7. Run the test and observe rate limiting behavior

Example 6: API Performance Testing with Ramp-up
This example demonstrates a simple performance test with gradual user ramp-up.

Steps:

1. Create a new Thread Group:

```
Number of Threads (users): 100
Ramp-up period: 30
Loop Count: 10
```

2. Add an HTTP Request:

```
Name: List Users
Method: GET
Protocol: https
Server Name: reqres.in
Path: /api/users?page=2
```

3. Add Response Assertion:

```
Apply to: Main sample and sub-samples
Pattern Matching Rules: Response Code
Patterns to Test: 200
```

4. Add JSON Assertion:

```
Assert JSON Path exists: $.data[0].id
```

5. Add Summary Report listener
6. Add View Results Tree listener
7. Add Graph Results listener
8. Run the test and analyze the performance metrics

JMeter Functions

_time

The _time function returns the current time in various formats.

Examples:

${__time(yyyy-MM-dd HH:mm:ss)}	Result: 2024-08-07 14:30:45
${__time(EEEE, MMMM d, yyyy)}	Result: Wednesday, August 7, 2024
${__time(HH:mm:ss.SSS)}	Result: 14:30:45.123
${__time(,)} (using default format)	Result: 1691418645123 (milliseconds since epoch)
${__time(dd/MM/yy)}	Result: 07/08/24

To use these in JMeter GUI:
1. Add a Sampler (e.g., HTTP Request)
2. In the sampler, add a parameter with the function as the value
3. Run the test and view results

_Random

The _Random function generates random numbers.

Examples:

${__Random(1,100)}	Result: Random number between 1 and 100
${__Random(1000,9999,ID)}	Result: Random 4-digit number, stored in variable 'ID'
${__Random()}	Result: Random number between 0 and Integer.MAX_VALUE
${__Random(50,150,weight)}	Result: Random number between 50 and 150, stored in 'weight'
${__Random(-10,10)}	Result: Random number between -10 and 10

To use in JMeter GUI:
1. Add a Sampler
2. Use the function in a parameter or in the sampler configuration
3. Run the test to see random values generated

_V

The _V function retrieves variables.

Examples:

${__V(username)}	Result: Value of variable 'username'
${__V(count,0)}	Result: Value of 'count', or 0 if not defined
${__V(${dynamicVar})}	Result: Value of variable whose name is stored in 'dynamicVar'
${_V(user${ID})}	Result: Value of variable like 'user_1', 'user_2', etc.
${__V(THREAD_NUMBER)}	Result: Current thread number

To use in JMeter GUI:

1. Define variables using User Defined Variables or CSV Data Set Config
2. Use _V function in samplers or other elements
3. Run the test to see variable values

_P

The _P function retrieves JMeter properties.

Examples:

${__P(jmeter.version)}	Result: JMeter version
${__P(user.dir)}	Result: JMeter working directory
${__P(os.name)}	Result: Operating system name
${__P(file.encoding)}	Result: System file encoding
${__P(custom.property,default)}	Result: Value of 'custom.property' or 'default' if not set

To use in JMeter GUI:

1. Set properties in jmeter.properties file or via command line
2. Use _P function in test elements
3. Run the test to see property values

Workload Modelling

What is Workload Modelling

Workload Modelling is the process of simulating the behavior of users interacting with an application. It involves creating test scenarios that represent different types of user activities and their frequency, helping to predict how the system performs under various conditions. This process is crucial for performance testing to ensure applications can handle real-world usage.

Types of Testing in Workload Modelling

- Load Testing: Determines how the system behaves under expected load conditions.
- Stress Testing: Identifies the system's breaking point by applying higher than normal loads.
- Spike Testing: Tests the system's performance when there is a sudden, drastic increase in load.
- Endurance Testing: Evaluates the system's performance under a sustained load over a prolonged period.

Load, Stress, Spike, Endurance Testing real world scenario design in JMeter

Load Testing

Scenario: Simulating 100 concurrent users accessing the home page of a website.

1. Add a Thread Group:
    a. Right-click on Test Plan > Add > Threads (Users) > Thread Group.
    b. Set Number of Threads (users): 100

        c. Set Ramp-Up Period (seconds): 10
        d. Set Loop Count: 1
2. Add an HTTP Request Sampler:
    a. Right-click on Thread Group > Add > Sampler > HTTP Request.
    b. Server Name or IP: example.com
    c. Path: /
3. Add a Listener:
    a. Right-click on Thread Group > Add > Listener > View Results Tree.
4. Run the Test:
5. Click the green Start button (or press Ctrl+R) to run the test.

Stress Testing

Scenario: Simulating increasing load until the system breaks, starting with 100 users and increasing by 50 every 10 seconds.

1. Add a Thread Group:
    a. Right-click on Test Plan > Add > Threads (Users) > Thread Group.
    b. Set Number of Threads (users): 100
    c. Set Ramp-Up Period (seconds): 10
    d. Set Loop Count: forever
2. Add an HTTP Request Sampler:
    a. Right-click on Thread Group > Add > Sampler > HTTP Request.
    b. Server Name or IP: example.com
    c. Path: /
3. Add a Constant Throughput Timer:
    a. Right-click on Thread Group > Add > Timer > Constant Throughput Timer.
    b. Target Throughput (in samples per minute): 100
4. Add a Listener:
    a. Right-click on Thread Group > Add > Listener > Aggregate Report.
5. Run the Test and monitor the system to determine when it starts failing or degrading in performance.

Spike Testing

Scenario: Simulating a sudden spike of 1000 users for a short duration.

1. Add a Thread Group:
    a. Right-click on Test Plan > Add > Threads (Users) > Thread Group.
    b. Set Number of Threads (users): 1000
    c. Set Ramp-Up Period (seconds): 1
    d. Set Loop Count: 1
2. Add an HTTP Request Sampler:
    a. Right-click on Thread Group > Add > Sampler > HTTP Request.
    b. Server Name or IP: example.com
    c. Path: /
3. Add a Listener:
    a. Right-click on Thread Group > Add > Listener > Summary Report.
4. Run the Test and observe the system's response to the sudden load.

Endurance Testing

Scenario: Simulating 100 users accessing the website continuously for 24 hours.

1. Add a Thread Group:
   a. Right-click on Test Plan > Add > Threads (Users) > Thread Group.
   b. Set Number of Threads (users): 100
   c. Set Ramp-Up Period (seconds): 10
   d. Set Loop Count: forever
2. Add an HTTP Request Sampler:
   a. Right-click on Thread Group > Add > Sampler > HTTP Request.
   b. Server Name or IP: example.com
   c. Path: /
3. Add a Listener:
   a. Right-click on Thread Group > Add > Listener > Graph Results.
4. Run the Test for a prolonged period (24 hours) and monitor the system's stability and performance over time.

Non-GUI Test Execution

What is Non-GUI Mode execution in Jmeter

Non-GUI mode in JMeter allows you to run tests without the graphical user interface, using only the command line. This mode is more efficient for running large-scale tests.

Why to use Non-GUI mode in execution

- ✓ Reduced resource consumption
- ✓ Faster test execution
- ✓ Suitable for running tests on servers without GUI
- ✓ Easier integration with CI/CD pipelines
- ✓ Better for running long-duration or high-load tests

How to setup and run Test in Non-GUI mode

1. Create your test plan in JMeter GUI and save it as a .jmx file.
2. Open a command prompt or terminal.
3. Navigate to the JMeter bin directory.
4. Run the test using the following command:

```
jmeter -n -t path/to/your/testplan.jmx -l path/to/results.jtl
```

```
Where:
-n: Run in non-gui mode
-t: Specify the test plan file
-l: Specify the log file to save results
```

5. Wait for the test to complete. Results will be saved in the specified .jtl file.

Report Dashboard Generation in Non-GUI mode

1. To generate a HTML dashboard report in non-GUI mode, follow these steps:
2. Run the test with additional parameters:

```
jmeter -n -t path/to/your/testplan.jmx -l path/to/results.jtl -e -o path/to/report/folder
```

```
Where:
-e: Generate report dashboard after load test
-o: Output folder for report dashboard
```

3. After the test completes, JMeter will generate an HTML report in the specified folder.
4. Open the index.html file in the report folder to view the dashboard.
5. The dashboard includes various statistics, graphs, and tables summarizing your test results, including response times, throughput, errors, and more.

Distributed Testing

What is Distributed Load Testing

Distributed load testing in JMeter involves leveraging multiple machines (slaves) to simulate a higher load on an application compared to what a single machine can handle. This architecture is particularly useful for large-scale performance testing.

Key Components:

- Master: Controls the test execution, coordinates slave machines, and collects results.
- Slaves: Execute test plans sent by the master, simulating user load.

Configuring servers

Prerequisites:

- ✓ Multiple machines (master and slaves) with identical JMeter and Java versions.
- ✓ Network connectivity between all machines.
- ✓ Firewall configuration to allow RMI traffic (usually ports 1099 and 1098).

Steps:

1. Configure Master:
    a. Open jmeter.properties file on the master machine.
    b. Add IP addresses of slave machines to the remote_hosts property:

    ```
    remote_hosts=192.168.1.100,192.168.1.101
    ```

2. Configure Slaves:
    - No specific configuration is required on slave machines.

Running from command line

1. Starting Slaves:
   a. Navigate to the bin directory of JMeter on each slave machine.
   b. Run the jmeter-server script:

```
./jmeter-server
```

2. Running Test from Master:
   a. Navigate to the bin directory of JMeter on the master machine.
   b. Run the following command to start the test:

```
./jmeter -n -t <test_plan.jmx> -r -l <results_file.jtl>
```

```
-n: Non-GUI mode
-t: Specifies the test plan file
-r: Remote execution (runs on all configured slaves)
-l: Specifies the results file
```

Gathering results

- Results are collected in the specified results_file.jtl on the master machine.
- JMeter GUI can be used to open and analyze the results file.
- Third-party reporting tools can also be used for in-depth analysis.

Example 1: Basic Setup

1. Master: 192.168.1.100
2. Slaves: 192.168.1.101, 192.168.1.102
3. Master Configuration:

```
remote_hosts=192.168.1.101,192.168.1.102
```

4. Running Test:

```
./jmeter -n -t my_test.jmx -r -l results.jtl
```

Example 2: Large-Scale Load Testing

1. Master: Central server
2. Slaves: Multiple distributed servers in different locations
3. Configuration:
   a. Configure remote_hosts on the master with IP addresses of all slaves.
   b. Ensure network connectivity and firewall rules.
4. Running Test:

```
./jmeter -n -t performance_test.jmx -r -l performance_results.jtl
```

Example 3: Distributed Testing with Load Balancing

1. Master: Load balancer
2. Slaves: Multiple application servers
3. Configuration:

4. Configure load balancer to distribute traffic to slaves.
5. Configure remote_hosts on the master with IP addresses of slaves.
6. Running Test:

```
./jmeter -n -t load_balanced_test.jmx -r -l load_balanced_results.jtl
```

JMeter Reporting

What is JTL

JTL (JMeter Test Logs) files are the primary format for storing test results in JMeter. They contain detailed information about each request, such as response times, success/failure status, and other metrics. JTL files can be saved in XML or CSV format and are used for further analysis and generating reports.

Understanding JTL File

A JTL file contains a record for each sample (request) executed during a test. Each record includes:

- Timestamp: The time when the request was made.
- Elapsed: The time taken to get a response (in milliseconds).
- Label: The name of the request.
- Response Code: The HTTP response code returned by the server.
- Response Message: The HTTP response message.
- Thread Name: The name of the thread that executed the request.
- Data Type: The type of data (text, binary, etc.).
- Success: Whether the request was successful or not (true/false).
- Bytes: The size of the response (in bytes).
- Sent Bytes: The size of the request (in bytes).
- Latency: The time to the first response byte.
- Idle Time: Time the request was idle.
- Connect: The time taken to establish a connection.

HTML Report generation in JMeter

JMeter can generate comprehensive HTML reports that provide a detailed overview of test results. Here's how to generate HTML reports:

Step-by-Step

1. Run the Test Plan and Save Results to a JTL File
    a. Create a simple test plan with an HTTP Request.
    b. Add a Listener > View Results Tree.
    c. In the View Results Tree, enable "Write results to file/Read from file" and specify the file name (e.g., results.jtl).
    d. Run the test plan.

2. Generate HTML Report from Command Line
    a. Open a terminal or command prompt.
    b. Navigate to the JMeter bin directory.
    c. Execute the following command:

    ```
    jmeter -g /path/to/results.jtl -o /path/to/output/report
    ```

    d. Replace /path/to/results.jtl with the path to your JTL file and /path/to/output/report with the desired output directory for the report.
    e. View the HTML Report

3. Open the index.html file in the output report directory in a web browser to view the generated HTML report.

Example 1: Simple Load Test

1. Thread Group: 10 users, 1-second ramp-up, 1 loop.
2. HTTP Request: Server example.com, path /.
3. Listener: View Results Tree, save results to simple_load_test.jtl.
4. Generate HTML Report using the command:

```
jmeter -g simple_load_test.jtl -o simple_load_test_report
```

Example 2: Stress Test

1. Thread Group: 100 users, 10-second ramp-up, 1 loop.
2. HTTP Request: Server example.com, path /stress.
3. Listener: View Results Tree, save results to stress_test.jtl.
4. Generate HTML Report using the command:

```
jmeter -g stress_test.jtl -o stress_test_report
```

Example 3: Endurance Test

1. Thread Group: 20 users, 5-second ramp-up, 10 loops.
2. HTTP Request: Server example.com, path /endurance.
3. Listener: View Results Tree, save results to endurance_test.jtl.
4. Generate HTML Report using the command:

```
jmeter -g endurance_test.jtl -o endurance_test_report
```

Real-time results monitoring

JMeter provides several ways to monitor test results in real-time during the execution of a test plan. Here are three examples:

Example 1: View Results Tree

1. Add a Listener:
2. Right-click on the Thread Group > Add > Listener > View Results Tree.
3. Run the Test and observe the results in real-time.

Example 2: Summary Report

1. Add a Listener:
2. Right-click on the Thread Group > Add > Listener > Summary Report.
3. Run the Test and monitor the aggregated results, including average, min, max, and throughput metrics.

Example 3: Aggregate Graph

1. Add a Listener:
2. Right-click on the Thread Group > Add > Listener > Aggregate Graph.
3. Run the Test and view the real-time graphical representation of results, showing metrics such as average response time and error percentage.

Analyzing and Interpreting Load Test Results

Interpreting JMeter reports

JMeter provides a variety of listeners to visualize test results. The most commonly used ones are:

- View Results in Table: Displays detailed information about each sample, including label, sample count, error count, average, median, 90th percentile, min, max, error, throughput, and kB/sec.
- Graph Results: Visualizes response times over time.
- Aggregate Report: Provides summary statistics for all samples.
- Summary Report: Similar to Aggregate Report but with additional columns.
- Assertion Results: Displays results of assertions.
- View Results Tree: Provides a hierarchical view of the test plan.
- HTTP(S) Test Script Recorder: Captures HTTP traffic.

Example:
Let's assume we have a JMeter test plan to test a web application. After running the test, we can analyze the results using the following listeners:

- ✓ View Results in Table: Identify specific requests with high response times or errors.
- ✓ Graph Results: Visualize response time trends over time to identify performance issues.
- ✓ Aggregate Report: Get overall performance metrics like average response time, throughput, and error rate.

Identifying performance bottlenecks

To identify performance bottlenecks, analyze the following metrics:

- Response Times: High response times indicate potential bottlenecks.
- Throughput: Low throughput can indicate server overload or network congestion.
- Error Rates: High error rates point to application issues or server errors.
- Server Metrics: Monitor server CPU, memory, and disk usage to identify resource constraints.
- Network Metrics: Check network latency, packet loss, and bandwidth to rule out network issues.

Example:
If we observe high response times for a specific request in the "View Results in Table" listener, we can:

- Analyze the request: Check if the request is making unnecessary database calls or complex calculations.
- Profile the application: Use profiling tools to identify performance-critical code sections.
- Monitor server metrics: Check if the server is under heavy load.
- Check network metrics: Ensure network connectivity is optimal.

Good Practices

Embedded resources

Embedded resources (images, CSS, JavaScript) can significantly impact performance. To optimize them:

- ✓ Compress resources: Reduce file size without affecting quality.
- ✓ Optimize images: Use appropriate image formats (JPEG, PNG) and compression levels.
- ✓ Combine files: Reduce HTTP requests by combining multiple files into one.
- ✓ Enable browser caching: Store resources locally to reduce server load.

Example 1: Embedding an Image

1. Record the Image Request:
    a. Open the JMeter GUI and add an HTTP(S) Test Script Recorder to your test plan.
    b. Start recording and browse to the webpage containing the image.
    c. Stop recording.

2. Find the Image Request:
    a. In the Recording Controller, locate the HTTP request for the image.

3. Base64 Encode the Image:
    a. Use an external tool or online service to convert the image file to a Base64 string.

4. Modify the HTTP Request:
    a. Right-click on the image request and select "Add -> Config Element -> HTTP Header Manager".
    b. Add a header named "Content-Type" with the value "image/jpeg" (or the appropriate image format).
    c. Right-click on the image request and select "Add -> Config Element -> HTTP Header Manager".
    d. Add a header named "Content-Type" with the value "application/octet-stream".
    e. Right-click on the image request and select "Add -> Config Element -> HTTP Header Manager".
    f. Add a header named "Content-Transfer-Encoding" with the value "base64".
    g. Change the HTTP method to "POST".
    h. In the Body Data tab, paste the Base64 encoded image data.

5. Remove the Image Request:
    a. Delete the image request from the Recording Controller.

Example 2: Embedding CSS

1. Record the CSS Request:
    a. Open the JMeter GUI and add an HTTP(S) Test Script Recorder to your test plan.
    b. Start recording and browse to the webpage containing the CSS file.
    c. Stop recording.

2. Find the CSS Request:
    a. In the Recording Controller, locate the HTTP request for the CSS file.

3. Extract CSS Content:
    a. Add a Regular Expression Extractor as a child of the CSS request.
    b. Configure the Regular Expression Extractor to extract the CSS content from the response body.

4. Modify the Main HTML Request:
    a. Locate the HTTP request for the main HTML page.
    b. Add a HTTP Header Manager to the main HTML request and set the header name to "Content-Type" with the value "text/html".
    c. Add a User Defined Variable to the main HTML request. Name it "cssContent" and set its value to the reference name of the Regular Expression Extractor.
    d. In the Body Data of the main HTML request, insert the following:

```
<style>${cssContent}</style>
```

5. Remove the CSS Request:
    a. Delete the CSS request from the Recording Controller.

Example 3: Embedding JavaScript

1. Record the JavaScript Request:
    a. Open the JMeter GUI and add an HTTP(S) Test Script Recorder to your test plan.
    b. Start recording and browse to the webpage containing the JavaScript file.
    c. Stop recording.

2. Find the JavaScript Request:
    a. In the Recording Controller, locate the HTTP request for the JavaScript file.

3. Extract JavaScript Content:
    a. Add a Regular Expression Extractor as a child of the JavaScript request.
    b. Configure the Regular Expression Extractor to extract the JavaScript content from the response body.

4. Modify the Main HTML Request:
    a. Locate the HTTP request for the main HTML page.

   b. Add a HTTP Header Manager to the main HTML request and set the header name to "Content-Type" with the value "text/html".
   c. Add a User Defined Variable to the main HTML request. Name it "jsContent" and set its value to the reference name of the Regular Expression Extractor.
   d. In the Body Data of the main HTML request, insert the following:

```
<script>${jsContent}</script>
```

5. Remove the JavaScript Request:
   a. Delete the JavaScript request from the Recording Controller.

Cache

Caching improves performance by storing frequently accessed data. In JMeter, you can use the HTTP Cache Manager to simulate browser caching behavior.

Example:

✓ Configure the HTTP Cache Manager with appropriate settings (cache size, policy, etc.) to simulate browser caching. Analyze the results to see the impact on performance.

Example 1: Basic Caching with HTTP Cache Manager

1. Add HTTP Cache Manager:
   a. Right-click on the Test Plan and select "Add -> Config Element -> HTTP Cache Manager".
   b. Set the cache size to a desired value (e.g., 50000).

2. Run the Test:
   a. Execute the test and observe the cache hit rate in the "View Results in Table" listener.

Example 2: Caching with Expiration Policy

1. Add HTTP Cache Manager:
   a. Right-click on the Test Plan and select "Add -> Config Element -> HTTP Cache Manager".
   b. Set the cache size to a desired value.
   c. Check the "Use cache for all requests" box.
   d. Set the "Cache policy" to "Standard".

2. Configure Cache Expiration:
   a. In the same HTTP Cache Manager, set the "Default cache expiration time" to a desired value (e.g., 600000 milliseconds).

3. Run the Test:
   a. Execute the test and verify that cached responses are used when applicable.

Example 3: Conditional Requests

1. Enable Conditional Requests:
   a. In the HTTP Request sampler, check the "Use Keep-Alive" and "Follow Redirects" options.
   b. Add If-Modified-Since Header:

2. Add an HTTP Header Manager to the HTTP Request sampler.
    a. Set the header name to "If-Modified-Since" and the value to the last modified date of the resource (you can use a timestamp or a variable).

3. Run the Test:
    a. Execute the test and observe if the server returns a 304 Not Modified status code.

Configuring and using JMeter Plugins

How to use Plugins Manager

The JMeter Plugins Manager is a tool that simplifies the installation, upgrade, and removal of plugins in JMeter.

1. Download and Install Plugins Manager
    a. Download the plugins-manager.jar from the JMeter Plugins website
    b. Place it in the lib/ext directory of your JMeter installation

2. Access Plugins Manager
    a. Start JMeter
    b. Go to Options > Plugins Manager

3. Install Plugins
    a. In the Available Plugins tab, browse or search for plugins
    b. Check the boxes next to plugins you want to install
    c. Click "Apply Changes and Restart JMeter"

    Examples:
    - Install "3 Basic Graphs"
    - Install "Custom Thread Groups"
    - Install "Dummy Sampler"
    - Install "HTTP/2 Sampler"
    - Install "JSON Formatter"

4. Upgrade Plugins
    a. Go to the Upgrades tab
    b. Select plugins to upgrade
    c. Click "Apply Changes and Restart JMeter"

5. Uninstall Plugins
    a. Go to the Installed Plugins tab
    b. Uncheck plugins you want to remove
    c. Click "Apply Changes and Restart JMeter"

PerfMon Metric Collector

PerfMon Metrics Collector allows you to monitor server health parameters during a test.

1. Install PerfMon Server Agent
    a. Download ServerAgent-x.x.x.zip
    b. Extract and run ServerAgent on the server to be monitored

2. Add PerfMon Metrics Collector to Test Plan
    a. Right-click on Test Plan > Add > Listener > jp@gc - PerfMon Metrics Collector

3. Configure PerfMon Metrics Collector
    a. Add row for each metric to collect
    b. Specify server IP, port, and metric type

    Examples:
    a. CPU Usage:
    ```
    Metric: CPU
    Server: 192.168.1.100:4444
    ```
    b. Memory Usage:
    ```
    Metric: Memory
    Server: 192.168.1.100:4444
    ```
    c. Disk I/O:
    ```
    Metric: Disk I/O
    Server: 192.168.1.100:4444
    ```
    d. Network I/O:
    ```
    Metric: Network I/O
    Server: 192.168.1.100:4444
    ```
    e. JMX (for Java applications):
    ```
    Metric: JMX
    Server: 192.168.1.100:4444
    ```

4. Run the test and observe real-time server metrics

SSHMon Samples Collector

SSHMon Samples Collector allows you to execute commands on remote servers via SSH and collect the results.

1. Add SSHMon Samples Collector
    a. Right-click on Test Plan > Add > Listener > jp@gc - SSHMon Samples Collector

2. Configure SSH Connection
    a. Specify hostname, port, username, and authentication method

3. Add Commands
    a. Add rows for each command to execute
    b. Specify command and parsing rules

    Examples:
    a. CPU Load:
    ```
    Command: uptime | awk '{print $10}'
    Parse as: Number
    ```
    b. Free Memory:
    ```
    Command: free -m | awk 'NR==2 {print $4}'
    Parse as: Number
    ```
    c. Disk Usage:
    ```
    Command: df -h / | awk 'NR==2 {print $5}' | sed 's/%//'
    Parse as: Number
    ```
    d. Active SSH Sessions:
    ```
    Command: who | wc -l
    Parse as: Number
    ```
    e. Apache Web Server Status:
    ```
    Command: systemctl is-active apache2 && echo 1 || echo 0
    Parse as: Number
    ```

4. Run the test and observe collected samples

Response times Over Time

This graph shows how response times vary over the duration of the test.

1. Add Response Times Over Time
    a. Right-click on Test Plan > Add > Listener > jp@gc - Response Times Over Time

2. Configure Graph
    a. Customize settings like interval, include/exclude sample labels

3. Run Test
    a. Execute your test plan

4. Analyze Graph
    a. Observe how response times change over time

    Examples of analysis:
    - Identify performance degradation over time
    - Spot periodic spikes in response time
    - Compare response times of different requests
    - Detect the impact of increasing load on response times
    - Identify the time when the system stabilizes under load

Analyzing Results

JMeter offers various ways to analyze test results.

1. Add Listeners
    a. View Results Tree
    b. Aggregate Report
    c. Summary Report
    d. Graph Results
    e. jp@gc - Transactions per Second

2. Run Test
3. Analyze Results

Examples:

1. View Results Tree:
    - Examine individual request/response details
    - Debug errors in requests

2. Aggregate Report:
    - Analyze average, median, 90th percentile response times
    - Check error rates for each request type

3. Summary Report:
    - Get an overview of test performance
    - Compare throughput of different requests

4. Graph Results:
    - Visualize response times, throughput, and errors over time
    - Identify performance trends

5. Transactions per Second:
    - Monitor system throughput in real-time
    - Identify peak transaction rates and when they occur

Real-world Examples and Case Studies

Real-world scenarios and challenges

Scenario 1: E-commerce Website Load Testing
Objective: Ensure the website can handle 500 concurrent users during a sale event.

1. Thread Group Configuration:
    a. Add Thread Group: Test Plan > Add > Threads (Users) > Thread Group
    b. Set Number of Threads (users): 500
    c. Set Ramp-Up Period (seconds): 60
    d. Set Loop Count: 1
2. HTTP Request Sampler:
    a. Add HTTP Request: Thread Group > Add > Sampler > HTTP Request
    b. Server Name: www.example.com
    c. Path: /home
3. Listeners:
    a. Add Summary Report: Thread Group > Add > Listener > Summary Report
    b. Add View Results Tree: Thread Group > Add > Listener > View Results Tree
4. Run the Test:
5. Analyze the results in the Summary Report and View Results Tree listeners.

Challenges and Lessons Learned:
- Challenge: Network bottleneck causing delays.
- Lesson Learned: Ensure network infrastructure is optimized and scalable.

Scenario 2: API Stress Testing
Objective: Test the API's stability under extreme load conditions.

1. Thread Group Configuration:
    a. Add Thread Group: Test Plan > Add > Threads (Users) > Thread Group
    b. Set Number of Threads (users): 1000
    c. Set Ramp-Up Period (seconds): 30
    d. Set Loop Count: 10
2. HTTP Request Sampler:

    a. Add HTTP Request: Thread Group > Add > Sampler > HTTP Request
    b. Server Name: api.example.com
    c. Path: /v1/resource
    d. Method: GET
3. Listeners:
    a. Add Aggregate Report: Thread Group > Add > Listener > Aggregate Report
    b. Add Response Time Graph: Thread Group > Add > Listener > Response Time Graph
4. Run the Test:
5. Analyze the results in the Aggregate Report and Response Time Graph listeners.

Challenges and Lessons Learned:

- Challenge: API timeouts and errors under load.
- Lesson Learned: Optimize API endpoints and implement proper error handling.

Scenario 3: Endurance Testing of a Web Application
Objective: Test the application's performance over a prolonged period.

1. Thread Group Configuration:
    a. Add Thread Group: Test Plan > Add > Threads (Users) > Thread Group
    b. Set Number of Threads (users): 50
    c. Set Ramp-Up Period (seconds): 10
    d. Set Loop Count: forever
2. HTTP Request Sampler:
    a. Add HTTP Request: Thread Group > Add > Sampler > HTTP Request
    b. Server Name: webapp.example.com
    c. Path: /dashboard
3. Listeners:
    a. Add Graph Results: Thread Group > Add > Listener > Graph Results
    b. Add Summary Report: Thread Group > Add > Listener > Summary Report
4. Run the Test:
5. Monitor the performance over 24 hours and analyze the results in the Graph Results and Summary Report listeners.

Challenges and Lessons Learned:

- Challenge: Memory leaks causing degradation over time.
- Lesson Learned: Regularly review and optimize application code for resource management.

JMeter integration with Jenkins

Run JMeter test from GIT using Jenkins

Setting up Jenkins and Git

1. Install Jenkins and necessary plugins:
    a. Go to "Manage Jenkins" > "Manage Plugins"
    b. Install "Git plugin" and "Performance plugin"

2. Create a new Jenkins job:
    a. Click "New Item"
    b. Choose "Freestyle project"
    c. Name your project (e.g., "JMeter-Test")

3. Configure Source Code Management:
    a. In job configuration, go to "Source Code Management"
    b. Select "Git"
    c. Enter your Git repository URL
    d. Specify branch (e.g., */main)

Configuring JMeter Test Execution

1. Add build step:
    a. In job configuration, go to "Build"
    b. Click "Add build step" > "Execute shell"

2. Add JMeter execution command:

```
jmeter -n -t path/to/your/test.jmx -l results.jtl -e -o report
```

3. Add post-build action:
    a. Go to "Post-build Actions"
    b. Click "Add post-build action" > "Publish Performance test result report"
    c. Set "Source data files" to: **/*.jtl

Scheduling and Triggering Tests

1. Configure build triggers:
    a. In job configuration, go to "Build Triggers"
    b. Choose appropriate trigger (e.g., "Poll SCM" or "Build periodically")
2. Set up notifications:
    a. Go to "Post-build Actions"
    b. Add "E-mail Notification" to send results
3. Save and run the job:
    a. Click "Save"
    b. Click "Build Now" to test the setup