**Interactive Learning Exercises - Step by Step**

**🎯 Exercise 1: Understanding the Request Flow**

**Goal: Trace exactly how a request flows through your system**

**Step 1: Setup Monitoring**

# Open 3 PowerShell windows:

# Window 1: Monitor Apache error log

Get-Content "D:\Software\ApacheHttpd\Apache24\logs\error.log" -Wait -Tail 0

# Window 2: Monitor Apache access log

Get-Content "D:\Software\ApacheHttpd\Apache24\logs\access.log" -Wait -Tail 0

# Window 3: Make requests (this window)

**Step 2: Make a Simple Request**

# In Window 3, run:

curl http://localhost/load/view -v

# Watch what happens in Windows 1 & 2

# You should see:

# - Access log: Shows the request

# - No errors in error log (if working correctly)

**Step 3: Analyze the Response**

# Look for these elements in the response:

# 1. HTTP status code (should be 200)

# 2. Response body (your application response)

# 3. Set-Cookie header (if sessions are working)

# Question: Which server handled your request?

# Answer: Check the response JSON for "serverId" field

**Step 4: Understanding the Path Translation**

Your request: http://localhost/load/view

Apache matches: ProxyPass /load

Apache forwards: http://localhost:8080/load/load/view

Spring receives: /load/load/view

Spring processes: context(/load) + controller(/load) + method(/view)

**🤔 Think About**: Why do we need /load/load in the backend URL?

**🎯 Exercise 2: Load Balancing Algorithm Testing**

**Goal: See how different algorithms distribute requests**

**Step 1: Create a Request Counter Script**

# Create this script: test-distribution.ps1

$serverCounts = @{}

$totalRequests = 30

Write-Host "Testing load distribution with $totalRequests requests..."

for ($i = 1; $i -le $totalRequests; $i++) {

try {

# Make request (new session each time)

$response = Invoke-WebRequest -Uri "http://localhost/load/view"

$content = $response.Content | ConvertFrom-Json

$server = $content.serverId

# Count requests per server

if ($serverCounts.ContainsKey($server)) {

$serverCounts[$server]++

} else {

$serverCounts[$server] = 1

}

Write-Host "Request $i → $server"

Start-Sleep -Milliseconds 100

} catch {

Write-Host "Request $i failed: $($\_.Exception.Message)"

}

}

Write-Host "`nFinal Distribution:"

$serverCounts.GetEnumerator() | Sort-Object Name | ForEach-Object {

$percentage = ($\_.Value / $totalRequests) \* 100

Write-Host "$($\_.Key): $($\_.Value) requests ($([Math]::Round($percentage, 1))%)"

}

**Step 2: Test Current Algorithm**

# Run the script

.\test-distribution.ps1

# Expected result with byrequests:

# Roughly even distribution (33% each server)

**Step 3: Change Algorithm and Test**

# In your httpd.conf, change:

ProxySet lbmethod=byrequests

# to:

ProxySet lbmethod=bytraffic

# Restart Apache

net stop Apache2.4

net start Apache2.4

# Run the test again

.\test-distribution.ps1

**🤔 Think About**:

* Did the distribution change?
* Why might bytraffic behave differently than byrequests?

**🎯 Exercise 3: Sticky Sessions Deep Dive**

**Goal: Understand session affinity behavior**

**Step 1: Test Session Stickiness**

# Create this script: test-sticky-sessions.ps1

Write-Host "Testing sticky sessions..."

# Create a session

Write-Host "Creating session..."

$response1 = Invoke-WebRequest -Uri "http://localhost/load/view" -SessionVariable session

$data1 = $response1.Content | ConvertFrom-Json

Write-Host "First request → Server: $($data1.serverId), Session: $($data1.sessionId)"

# Make 5 more requests with same session

for ($i = 2; $i -le 6; $i++) {

$response = Invoke-WebRequest -Uri "http://localhost/load/tomcat" -WebSession $session

$data = $response.Content | ConvertFrom-Json

Write-Host "Request $i → Server: $($data.serverId)"

# Check if session is sticky

if ($data.serverId -ne $data1.serverId) {

Write-Host "❌ STICKY SESSION FAILED! Different server used."

} else {

Write-Host "✅ Sticky session working"

}

Start-Sleep -Seconds 1

}

**Step 2: Run the Test**

.\test-sticky-sessions.ps1

# Expected result:

# All requests should go to the same server

**Step 3: Test Without Sticky Sessions**

# Temporarily disable sticky sessions in httpd.conf

# Comment out this line:

# ProxySet stickysession=JSESSIONID

# Restart Apache and run test again

**🤔 Think About**:

* What happened when sticky sessions were disabled?
* How would this affect a real user's experience?

**🎯 Exercise 4: Failover Testing**

**Goal: Understand what happens when a server fails**

**Step 1: Create Session on Specific Server**

# Keep trying until you get server1

do {

$response = Invoke-WebRequest -Uri "http://localhost/load/view" -SessionVariable session

$data = $response.Content | ConvertFrom-Json

Write-Host "Got server: $($data.serverId)"

} while ($data.serverId -ne "server1")

Write-Host "Session created on server1. Session ID: $($data.sessionId)"

**Step 2: Stop Server1 (Port 8080)**

# Stop the Tomcat server running on port 8080

# (Use Ctrl+C in the terminal where it's running)

Write-Host "Now stop the server running on port 8080 and press Enter..."

Read-Host

**Step 3: Test Failover**

# Make request with existing session

Write-Host "Making request after server failure..."

try {

$failoverResponse = Invoke-WebRequest -Uri "http://localhost/load/tomcat" -WebSession $session

$failoverData = $failoverResponse.Content | ConvertFrom-Json

Write-Host "Request handled by: $($failoverData.serverId)"

if ($failoverData.serverId -ne "server1") {

Write-Host "✅ Failover successful! Request routed to healthy server."

}

} catch {

Write-Host "❌ Failover failed: $($\_.Exception.Message)"

}

**🤔 Think About**:

* How long did failover take?
* What happened to the session data?
* How could you minimize user impact?

**🎯 Exercise 5: Configuration Impact Analysis**

**Goal: Understand how configuration changes affect behavior**

**Step 1: Baseline Test**

# Test current configuration

$start = Get-Date

$response = curl http://localhost/load/view

$end = Get-Date

$responseTime = ($end - $start).TotalMilliseconds

Write-Host "Baseline response time: $responseTime ms"

Write-Host "Response: $response"

**Step 2: Change Timeout Settings**

# In httpd.conf, change:

BalancerMember http://localhost:8080 route=server1 retry=10 timeout=2

# to:

BalancerMember http://localhost:8080 route=server1 retry=10 timeout=10

# Restart Apache

**Step 3: Test Impact**

# Test again

$start = Get-Date

$response = curl http://localhost/load/view

$end = Get-Date

$responseTime = ($end - $start).TotalMilliseconds

Write-Host "New response time: $responseTime ms"

**Step 4: Force a Timeout**

# Simulate slow backend by adding delay to your Spring Boot controller

# Or use network simulation tools

# Then test how Apache handles slow backends

**🤔 Think About**:

* How do timeout settings affect user experience?
* What's the trade-off between quick failover and patience for slow requests?

**🎯 Exercise 6: Security Header Analysis**

**Goal: Understand security implications**

**Step 1: Check Current Headers**

# Make request and examine all headers

$response = Invoke-WebRequest -Uri "http://localhost/load/view" -Method GET

Write-Host "Response Headers:"

$response.Headers.GetEnumerator() | ForEach-Object {

Write-Host "$($\_.Key): $($\_.Value)"

}

**Step 2: Test HTTPS Redirect**

# Test HTTP to HTTPS redirect

$response = Invoke-WebRequest -Uri "http://localhost/load/view" -MaximumRedirection 0

Write-Host "Status Code: $($response.StatusCode)"

Write-Host "Location Header: $($response.Headers.Location)"

**Step 3: Security Header Testing**

# Check for security headers

$securityHeaders = @(

"Strict-Transport-Security",

"X-Content-Type-Options",

"X-Frame-Options",

"X-XSS-Protection"

)

$response = Invoke-WebRequest -Uri "https://localhost/load/view" -SkipCertificateCheck

foreach ($header in $securityHeaders) {

if ($response.Headers.ContainsKey($header)) {

Write-Host "✅ $header: $($response.Headers[$header])"

} else {

Write-Host "❌ Missing: $header"

}

}

**🤔 Think About**:

* Why is each security header important?
* What attacks do they prevent?

**🎯 Exercise 7: Performance Impact Measurement**

**Goal: Understand performance characteristics**

**Step 1: Measure Single vs Multiple Requests**

# Single request performance

$singleTimes = @()

for ($i = 1; $i -le 10; $i++) {

$start = Get-Date

Invoke-WebRequest -Uri "http://localhost/load/view" | Out-Null

$end = Get-Date

$singleTimes += ($end - $start).TotalMilliseconds

}

$avgSingle = ($singleTimes | Measure-Object -Average).Average

Write-Host "Average single request time: $([Math]::Round($avgSingle, 2))ms"

**Step 2: Concurrent Request Testing**

# Concurrent requests

Write-Host "Testing concurrent requests..."

$jobs = @()

for ($i = 1; $i -le 10; $i++) {

$job = Start-Job -ScriptBlock {

$start = Get-Date

Invoke-WebRequest -Uri "http://localhost/load/view" | Out-Null

$end = Get-Date

return ($end - $start).TotalMilliseconds

}

$jobs += $job

}

$concurrentTimes = $jobs | Wait-Job | Receive-Job

$jobs | Remove-Job

$avgConcurrent = ($concurrentTimes | Measure-Object -Average).Average

Write-Host "Average concurrent request time: $([Math]::Round($avgConcurrent, 2))ms"

**🤔 Think About**:

* Why might concurrent requests be slower/faster?
* How does load balancing affect performance?

**🎯 Exercise 8: Log Analysis Deep Dive**

**Goal: Understand what the logs tell you**

**Step 1: Generate Different Types of Requests**

# Make various requests to generate different log entries

# Success request

curl http://localhost/load/view

# Different endpoint

curl http://localhost/load/tomcat

# Non-existent endpoint (should be 404)

curl http://localhost/load/nonexistent

# Request with different user agent

curl -H "User-Agent: MyTestClient/1.0" http://localhost/load/view

**Step 2: Analyze Access Log Format**

# Look at recent access log entries

$logEntries = Get-Content "D:\Software\ApacheHttpd\Apache24\logs\access.log" -Tail 5

foreach ($entry in $logEntries) {

Write-Host "Raw log: $entry"

# Parse common elements

$parts = $entry -split ' '

Write-Host " IP: $($parts[0])"

Write-Host " Method: $($parts[5] -replace '["]', '')"

Write-Host " URL: $($parts[6])"

Write-Host " Status: $($parts[8])"

Write-Host " Size: $($parts[9])"

Write-Host ""

}

**Step 3: Error Log Analysis**

# Check for any errors

$errorEntries = Get-Content "D:\Software\ApacheHttpd\Apache24\logs\error.log" -Tail 10

Write-Host "Recent errors:"

foreach ($error in $errorEntries) {

Write-Host $error

}

**🤔 Think About**:

* What information is most valuable in the logs?
* How would you set up monitoring based on these logs?

**🔬 Advanced Understanding Challenges**

**Challenge 1: Custom Load Balancing**

Try to understand what would happen if you:

* Added weight factors to servers (loadfactor=50)
* Used different load balancing methods
* Added hot standby servers (status=+H)

**Challenge 2: Session Persistence**

Investigate:

* What happens to sessions during server maintenance?
* How could you implement shared session storage?
* When would you NOT want sticky sessions?

**Challenge 3: SSL/TLS Deep Dive**

Explore:

* SSL termination vs SSL pass-through
* Certificate validation process
* Impact of SSL on load balancing performance

**Challenge 4: Monitoring Integration**

Think about:

* What metrics would you monitor in production?
* How would you set up alerting?
* What dashboards would be most useful?

**🎓 Learning Assessment**

After completing these exercises, you should be able to:

1. ✅ Trace a request from browser to backend server