Performance Testing Framework

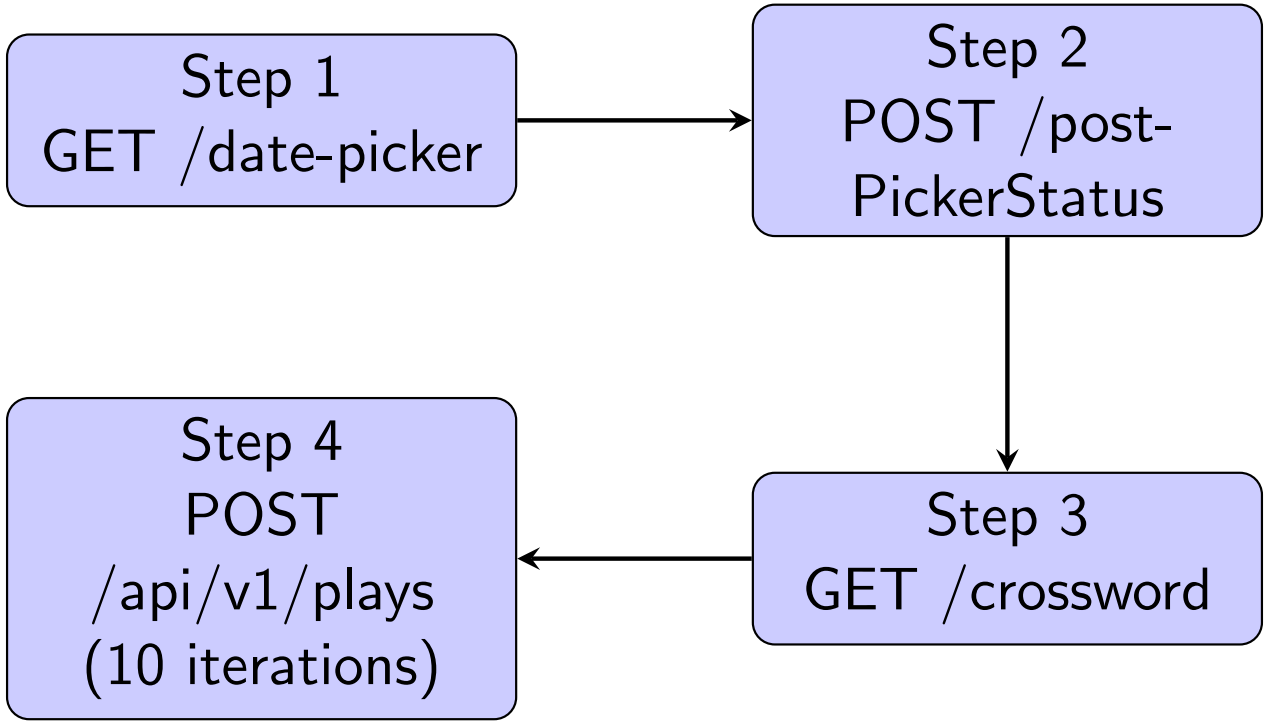
Java Implementation for Crossword API Load Testing

Performance Testing Team

December 30, 2025

Outline

API Flow: High-Level Architecture



Simulates complete user journey: Opening puzzle picker → Selecting puzzle → Playing the crossword

Step 1: Date Picker Page

Endpoint: `GET /date-picker`

Purpose:

- Fetches the puzzle date picker HTML page
- Extracts `loadToken` from embedded JSON params
- Token required for all subsequent API calls

Parameters:

- `set` - Puzzle series identifier
- `uid` - User identifier (random or fixed)

V3 Extension: Also fetches CDN resources:

- `date-picker-min.css`, `picker-min.js`
- Font Awesome CSS and WOFF2 fonts

Step 2: Post Picker Status

Endpoint: `POST /postPickerStatus`

Purpose:

- Notifies server that user is viewing the picker
- Validates the `loadToken`
- Records ad duration and verification status

Request Body:

```
{
    "loadToken": "<from step1>",
    "isVerified": true,
    "adDuration": 0,
    "reason": "displaying puzzle picker"
}
```

Step 3: Load Crossword

Endpoint: `GET /crossword`

Purpose:

- Loads the actual crossword puzzle page
- Extracts puzzle parameters and `playId`
- Uses hardcoded puzzle ID for consistent testing

Parameters:

- `id` - Puzzle identifier
- `set`, `picker`, `uid`, `loadToken`

V3 Extension: Also fetches:

- `crossword-player-min.css`
- `c-min.js` (crossword player script)

Step 4: Simulate Gameplay

Endpoint: `POST /api/v1/plays` (10 iterations)

Purpose:

- Simulates user solving the puzzle
- Sends state updates with progress

Play States:

1. **playState=1**: Game started
2. **playState=2** (iterations 2-9): In progress with mutations
3. **playState=4**: Game completed

State Management:

- `primaryState`: Current letter entries
- `secondaryState`: Fill status bitmap
- Random mutations simulate real user behavior

Why Java Over Python?

Performance Advantages:

- **True multithreading** - No GIL limitations
- **Connection pooling** via OkHttp for efficient HTTP
- **Lower memory footprint** per thread
- **Predictable latency** - JIT compilation benefits

Scalability:

- Handle higher RPS (requests per second)
- More accurate wave-based timing
- Better suited for long-running load tests

Technology Stack

Build System:

- **Maven** - Dependency management & packaging
- **Java 17** - Modern language features
- **Maven Shade Plugin** - Uber JAR packaging

Key Dependencies:

Library	Version	Purpose
OkHttp	4.12.0	HTTP client with connection pooling
Gson	2.10.1	JSON parsing and serialization
OpenCSV	5.9	CSV result file generation
JCommander	1.82	CLI argument parsing
SLF4J	2.0.9	Logging framework

OkHttp: Why This HTTP Client?

Key Features:

- **Connection pooling** - Reuses HTTP connections
- **HTTP/2 support** - Multiplexing for efficiency
- **Automatic retries** on connection failures
- **Configurable timeouts** (connect, read, write)

Configuration:

```
OkHttpClient client = new OkHttpClient.Builder()
    .connectTimeout(30, TimeUnit.SECONDS)
    .readTimeout(30, TimeUnit.SECONDS)
    .writeTimeout(30, TimeUnit.SECONDS)
    .build();
```

Project Structure

Source Files:

- `ApiFlowV2.java` - Main entry
- `ApiFlow.java` - HTTP flow logic
- `ApiFlowV3.java` - With CDN fetches
- `WaveExecutor.java` - RPS control
- `CsvResultWriter.java` - Output
- `HtmlReportWriter.java` - Dashboard

Build Outputs:

- `api-flow-v2.jar` - Standard flow
- `api-flow-v3.jar` - With CDN

Usage:

```
java -jar api-flow-v2.jar \
  --rps 5 --duration 10 \
  --random-uid --html
```

HTML Dashboard: Overview

Key Metrics Cards:

- **Total Threads** - Number of test runs
- **Success Rate** - Percentage of successful flows
- **Avg Latency** - Mean response time
- **P95 Latency** - 95th percentile
- **Min / Max** - Latency range

Color Coding:

- **Green** - Success rate $\geq 95\%$
- **Yellow** - Success rate 80-95%
- **Red** - Success rate $< 80\%$

Dashboard: Charts

Three Visualization Types:

1. **Latency Distribution (Histogram)**
   - Bucketed response times
   - Shows distribution shape (normal vs skewed)
2. **Per-Wave Average Latency**
   - Line chart over time
   - Detects performance degradation
3. **Per-Thread Completion Time**
   - Individual thread latencies
   - Identifies outliers

Charts use Chart.js with dark theme styling

Dashboard: Per-Wave Summary Table

Columns:

Column	Description
Wave	Wave number (1 wave = 1 second)
Success	Successful/Total threads in wave
Step1-4 Avg	Average latency per step
Total Avg	Sum of all step latencies
P95	95th percentile for the wave

Expandable Details:

- Click wave to see individual thread results
- Includes UID, start time, per-step latencies
- Outliers highlighted ($> 2\sigma$ from mean)

Dashboard: Error Summary

Error Grouping:

- Errors grouped by error message
- Sorted by count (most frequent first)
- Expandable to see affected threads

Error Details Table:

- Wave and Thread numbers
- UID and Start Time
- Step latencies before failure
- Failed Step indicator

Visual Indicators:

- Red left border for error sections
- Count badge with error frequency
- Monospace font for error messages

Dashboard Screenshot

[Dashboard Screenshot Placeholder]

Add screenshot of the HTML dashboard here

Summary & Next Steps

What We Covered:

- 4-step API flow simulating user journey
- Java technology stack and library choices
- HTML dashboard format and metrics

Next Steps:

- Add actual test results
- Include performance benchmarks
- Compare V2 vs V3 flow performance
- Add CDN latency breakdown analysis

Questions?