ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

Lớp CQ20/3 – Nhóm C2

SEMINAR PERFORMANCE TESTING

CÔNG CỤ LOCUST

Học kỳ I – Năm học 2023 - 2024

MÔN HỌC KIỂM THỬ PHẦN MỀM

CHƯƠNG TRÌNH CHÍNH QUY

GIẢNG VIÊN HƯỚNG DẪN

Thầy Trần Duy Hoàng

Thầy Hồ Tuấn Thanh

Thành phố Hồ Chí Minh, tháng 11 năm 2023

MỤC LỤC

1. Thông tin nhóm và đánh giá

MSSV	Họ tên	Đánh giá đóng góp
20120406	Phạm Quốc Vương (nhóm trưởng)	25%
20120251	Trần Đức Anh	25%
20120385	Trần Hoàng Tín	25%
20120435	Lê Thị Ngọc Bích	25%
	Tổng	100%

2. Lý thuyết về Performance Testing[1]

2.1. Performance và phân loại

Một ứng dụng có Performance tốt là một ứng dụng cho phép người dùng cuối thực hiện một nhiệm vụ nhất định mà không bị chậm trễ hoặc khó chịu quá mức. Performance của hệ thống tốt hay không thường tùy thuộc vào quan điểm và cảm nhận của mỗi người dùng.

Các chỉ số về Performance được chia thành nhiều loại, nhưng có hai loại quan trọng:

- **Service-oriented:** đánh giá ứng dụng có cung cấp tốt dịch vụ cho khách hàng hay không.
  - **Availability:** Khoảng thời khả dụng của ứng dụng đến người dùng cuối.
  - **Response Time:** Thời gian phản hồi một yêu cầu từ người dùng của ứng dụng.
- **Efficiency-oriented:**
  - **Throughput:** Tỷ lệ các sự kiện của ứng dụng (application-oriented events) xảy ra. Ví dụ: Số lượt truy cập vào một trang web trong một khoảng thời gian nhất định.
  - **Utilization:** Tỷ lệ phần trăm dung lượng lý thuyết của một tài nguyên đang được sử dụng. Ví dụ: lượng băng thông mạng được tiêu thụ bởi lưu lượng ứng dụng (application traffic) và dung lượng bộ nhớ được sử dụng trên máy chủ khi có một nghìn khách truy cập.

2.2. Nguyên nhân và hậu quả của Performance kém

2.2.1. Nguyên nhân

- Hệ thống dùng công nghệ cũ, sự phụ thuộc và ràng buộc về phiên bản giữa các thành phần hệ thống gây khó nâng cấp lên công nghệ có hiệu năng cao.
- Các vấn đề về tương thích công nghệ trong hệ thống.
- Thiết kế ứng dụng không cân nhắc về hiệu suất.
- Sự phổ biến của ứng dụng và tăng số lượng người dùng, cũng như số lượng người dùng đồng thời không được tính toán trước.
- Hiệu suất ứng dụng khi triển khai thường khác xa so với dự kiến trong quá trình phát triển sản phẩm.

---

[1] Mọi thông tin lý thuyết nhóm đã tìm hiểu đều được trình bày vào báo cáo này; nội dung thuyết trình trên lớp sẽ được rút gọn nhiều so với bản báo cáo này.

- Không coi trọng Performance Testing. Hiệu suất chỉ được kiểm tra trước triển khai, không duy trì kiểm tra khi ứng dụng hoạt động thực tế.
- Không sử dụng các công cụ tự động hóa kiểm tra hiệu suất.

2.2.2. Hậu quả của hiệu suất kém

- Giảm năng suất.
- Mất khách hàng.
- Thiếu uy tín.
- Tiêu tốn nhiều nguồn lực để sửa chữa và cải thiện hệ thống, ảnh hưởng đến lợi nhuận.

2.3. Performance Testing – khái niệm, phân loại và qui trình test

- **Performance Testing:** Kiểm thử hiệu năng - là kiểm tra về tốc độ, thời gian phản hồi, độ tin cậy, mức độ sử dụng tài nguyên, lượng người dùng tối đa, v.v của một phần mềm.
- **Mục đích:** Không phải là tìm ra các lỗi chức năng mà là để loại bỏ các tắc nghẽn về hiệu suất, xác minh ứng dụng hoặc hệ thống có khả năng chịu được tải trọng và duy trì hiệu suất mong muốn.
- **Phân loại:** các mô tả phân loại đồng thời mô tả hướng thiết kế kiểm thể
  - **Baseline Load Testing:** xác định các ngưỡng chịu tải tối đa của hệ thống.
  - **Stress Testing:** kiểm tra khả năng hoạt động của ứng dụng khi có lượng công việc cực lớn như xử lý lưu lượng truy cập cao hoặc xử lý rất nhiều dữ liệu, giúp xác định các trường hợp hệ thống crash và không phản hồi được nữa.
  - **Endurance (Soak) Testing:** kiểm tra khả năng phần mềm có thể xử lý lượng tải dự kiến trong một khoảng thời gian dài nhằm xác định các trường hợp rò rỉ bộ nhớ, cạn kiệt tài nguyên, hoặc giảm hiệu suất theo thời gian.
  - **Spike Testing:** kiểm tra phản ứng của phần mềm đối với các đột biến do có lượng tải lớn từ người dùng trong một khoảng thời gian ngắn.
  - **Volume Testing:** kiểm thử phi chức năng, kiểm tra khả năng chịu tải của phần mềm với một lượng dữ liệu nhất định bằng cách thay đổi kích thước cơ sở dữ liệu để kiểm tra hiệu năng của phần mềm.
  - **Scalability Testing:** xác định tính hiệu quả của phần mềm trong việc nhân rộng (ví dụ như tăng lượng người dùng) để hỗ trợ tăng tải cho người dùng, giúp lập kế hoạch bổ sung khả năng đáp ứng của hệ thống với nhu cầu tăng dần.
- **Qui trình kiểm thử**: gồm các bước tương tự như những gì đã học ở bài tổng quan

- **Xác định yêu cầu:** Xác định mục tiêu, kịch bản và yêu cầu hiệu suất của hệ thống.
- **Lập kế hoạch kiểm thử.**
- **Thiết kế kiểm:** Xác định các kịch bản kiểm tra, tải trọng và các chỉ số hiệu suất cần được theo dõi.
- **Chuẩn bị môi trường:** Thiết lập môi trường test với phần cứng, phần mềm và dữ liệu tương ứng.
- **Thực hiện kiểm thử:** Chạy các kịch bản kiểm tra với tải trọng đã xác định và ghi lại dữ liệu hiệu suất.
- **Phân tích kết quả:** Phân tích dữ liệu thu thập được để xác định sự ổn định, thời gian phản hồi và khả năng chống chọi của hệ thống.
- **Báo cáo kết quả:** Tạo báo cáo chi tiết về hiệu suất của hệ thống trong quá trình test và gợi ý những điều cần làm để nâng cao hiệu suất trong tương lai.

2.4. Vì sao cần Performance Testing

- Để đảm bảo hiệu suất.
- Phát hiện vấn đề tiềm ẩn.
- Hỗ trợ ra chiến lược tối ưu hóa tài nguyên phần cứng, băng thông mạng, v.v.
- Đáp ứng các yêu cầu kỹ thuật khác xuất phát từ việc kinh doanh.

2.5. Tấn công Performance và kiểm thử phát hiện sớm

- **Tấn công từ chối dịch vụ**:
  - **Mô tả:** Phía tấn công gửi một lượng lớn yêu cầu truy cập tới hệ thống để gây quá giới hạn xử lý, làm người dùng thật không thể sử dụng dịch vụ.
  - **Giải pháp test:** Dùng Baseline Load Testing xác định ngưỡng tiếp nhận yêu cầu tối đa mà hệ thống chịu được để chủ động cài đặt giới hạn cho hệ thống. Dùng Stress Testing, Spike Testing để mô phỏng lại lượng tải cao và lượng tải đột ngột vào hệ thống để kiểm tra các ngưỡng đã cài đặt hợp lý hay chưa. Thực hiện lại các bài kiểm tra theo định kỳ để phát hiện lỗi và đảm bảo hệ thống hoạt động ổn định.

2.6. Chi phí thực hiện Performance Testing

- **Phần cứng**: Chi phí thuê hoặc mua phần cứng và/hoặc sử dụng Amazon Web Services, Microsoft Azure hay Google Cloud Platform để có các máy chủ mạnh mẽ hoặc các dịch vụ đám mây để tạo ra tải cao cho ứng dụng.
- **Công cụ và giấy phép**: Có thể cần sử dụng công cụ kiểm tra hiệu năng như Apache JMeter, LoadRunner hay Gatling. Một số công cụ sẽ yêu cầu giấy phép trả phí để sử dụng phiên bản đầy đủ hoặc tính năng cao cấp.
- **Nhân viên và chuyên gia**: Lương cho bộ phận QA/QC/Tester nếu công ty có đội ngũ riêng, hoặc chi phí thuê chuyên gia hoặc nhân viên giàu kinh nghiệm trong lĩnh vực này để hỗ trợ quá trình kiểm tra.
- **Thời gian và công suất lao động**: Thực hiện Performance Testing đòi hỏi thời gian và công suất lao động từ nhóm phát triển và kiểm thử. Việc chuẩn bị, thiết kế, triển khai và phân tích kết quả của các bài kiểm tra hiệu năng có thể tốn nhiều ngày hoặc tuần.
- **Môi trường kiểm thử**: Cần có một môi trường kiểm thử tách biệt để tiến hành Performance Testing mà không ảnh hưởng đến môi trường sản xuất. Điều này có thể yêu cầu việc triển khai các phiên bản riêng biệt của ứng dụng hoặc sử dụng các giải pháp ảo hóa để tạo ra môi trường giống với sản phẩm cuối.
- **Chi phí liên quan**: Chi phí điện, internet, lưu trữ dữ liệu và các yếu tố khác liên quan đến việc chạy Performance Testing.

2.7. Performance Testing và tương lai phát triển

- Còn tồn tại phần mềm là còn cần đảm bảo performance cho phần mềm. Chỉ khi không còn phần mềm nữa thì mới không còn Performance Testing.
- Xu hướng Automation và CI/CD.
- Xu hướng mang testing vào cả Software Development Life Cycle.
- Testing cho các công nghệ phức tạp như Cloud Computing, Internet of Things (IoT) và Machine Learning.

2.8. Performance Testing Best Practices

- Đặt mục tiêu về Performance càng sớm càng tốt.
- Có tỷ lệ phù hợp giữa kích thước phần cứng thử nghiệm và sản xuất.
- Lên kế hoạch đủ thời gian cho việc viết test script.
- Chạy các test quan trọng hai lần.
- Kiểm tra với cơ sở dữ liệu kích thước đầy đủ.

2.9. Công ty nổi tiếng trong lĩnh vực Performance Testing

- **Neotys** với công cụ Performance Testing tự động NeoLoad.
- **Micro Focus** với công cụ Performance Testing LoadRunner.
- **Apache Software Foundation** (tổ chức phần mềm phi lợi nhuận) với công cụ Performance Testing Apache JMeter.
- **BlazeMeter** với các giải pháp Performance Testing dựa trên cloud.
- **RadView Software** với công cụ Performance Testing tự động WebLOAD.

3. Công cụ Locust

3.1. Giới thiệu về công cụ

3.1.1. Tác giả/nhà phát triển Locust

- Jonatan Heyman,
- Carl Byström,
- Joakim Hamrén,
- Hugo Heyman,
- Và sự đóng góp từ nhiều tác giả khác.

3.1.2. Giấy phép

- Locust là phần mềm mã nguồn mở và được cấp phép theo giấy phép MIT.

3.1.3. Cơ sở của việc phát triển Locust

Nhà phát triển Locust cho biết, việc họ tạo ra Locust xuất phát từ những hạn chế của các giải pháp Performance Testing tồn tại trước khi Locust ra đời. Không có công cụ nào có khả năng kiểm thử tốt đối với trang web động với nhiều nội dung khác nhau cho từng người dùng.

Các công cụ lúc bấy giờ có giao diện phức tạp hoặc các tệp cấu hình dài dòng để khai báo các ca kiểm thử. Để khắc phục điều đó, Locust cung cấp một framework Python, cho phép tester mô phỏng lại hoạt động của người dùng bằng code Python đơn giản.

3.1.4. Locust và các tính năng

Locust lấy tên từ loài châu chấu, được biết đến với hành vi bầy đàn, đây là một công cụ Load Testing được viết bằng Python, có khả năng tạo kịch bản mô phỏng tải và rất dễ sử dụng; thường được dùng cho việc load testing các website, hệ thống API, hỗ trợ tìm ra số lượng người dùng đồng thời mà hệ thống có thể xử lí. Ý tưởng của nó là dùng một nhóm các Locust để giả lập các requests tới website.

3.1.4.1. Viết kịch bản kiểm thử bằng Python

Locust hỗ trợ việc định nghĩa các hành vi của người dùng như thực hiện vòng lặp, thực hiện một số hành vi có điều kiện hoặc thực hiện các tính toán, v.v, bằng ngôn ngữ lập trình Python.

Locust chạy mỗi người dùng logic trong một tiến trình nhẹ (light-weight process) tạo ra bởi thư viện gevent. Thay vì sử dụng các hàm callback để xử lý sự kiện, light-weight process giúp thực hiện đồng thời nhiều công việc mà không cần tạo ra nhiều luồng hoặc tiến trình nặng.

3.1.4.2. Hỗ trợ hàng trăm nghìn người dùng logic đồng thời bằng tính phân tán và tính mở rộng của Locust

Locust hỗ trợ Load Testing trên nhiều máy tính một cách rất dễ dàng. Với cơ chế hoạt động dựa trên sự kiện (event-based) và ít tiêu tốn tài nguyên, Locust cho phép mỗi quy trình có thể xử lý hàng nghìn người dùng logic đồng thời.

3.1.4.3. Web-based UI

Locust hỗ trợ giao diện web thân thiện cho người dùng. Nó hiển thị tiến trình của ca kiểm thử trong thời gian thực, hỗ trợ thay đổi tải tại thời điểm đang thực hiện kiểm thử. Ngoài ra, Locust cũng hỗ trợ giao diện dòng lệnh, phù hợp cho CI/CD testing.

3.1.4.4. Có thể test bất cứ hệ thống nào

Dù chủ yếu hoạt động với website/webservice, nhưng Locust có khả năng kiểm thử hầu hết các hệ thống hoặc giao thức (XML-RPC, gRPC, requests-based libraries, REST, WebSocket, Selenium, Kafka, v.v). Tester chỉ cần viết một Locust client tương ứng cho hệ thống cần để có thể thực hiện kiểm thử.

3.2. Một số chi tiết kỹ thuật thường gặp khi sử dụng Locust

3.2.1. Các lớp cơ bản

- **User:** đại diện cho một người dùng đang truy cập vào hệ thống, chứa các hàm được gắn task decorator đại diện cho một công việc cần thực hiện.
- **HttpUser:** khi tạo một instance của lớp HttpUser, về mặt cơ chế, Locust sẽ tạo một HttpSession instance, hỗ trợ các thao tác với ứng dụng web như tạo cookies, cache, sửa header, sử dụng các HTTP method, v.v.
- **TaskSet:** dùng định nghĩa tập hợp các task cần thực hiện; có hỗ trợ khả năng lồng các TaskSet vào nhau (nested TaskSet) giúp xây dựng cấu trúc phân cấp

của ngữ cảnh test; để dùng TaskSet, phải tạo lớp User hoặc HttpUser và định nghĩa cho thuộc tính **tasks = [tên_lớp_TaskSet1, tên_lớp_TaskSet2, …]** mới có thể sử dụng TaskSet.

- **SequentialTaskSet:** dùng định nghĩa lớp có các phương thức được decorate @task chạy theo thứ tự từ trên xuống dưới (phương thức nào được định nghĩa trước thì chạy trước).

3.2.2. Các hàm và phương thức

- **Các hàm wait_time:** dùng định nghĩa khoảng cách về thời gian giữa các task, giúp mô phỏng lại việc sử dụng phần mềm một cách thực tế, nếu không có khoảng nghỉ, nó sẽ trở thành stress testing. Gồm các hàm:
  - **between(min, max):** chọn ngẫu nhiên độ delay có giá trị từ min đến max (lấy biên).
  - **constant(wait_time):** delay với khoảng thời gian là hằng số wait_time.
  - **constant_pacing(wait_time):** đảm bảo thời gian chạy của task không nhỏ hơn **wait_time**, nếu task chạy xong bussines logic code trước thời gian wait_time thì phải đợi hết wait_time task đó mới được coi là hoàn thành.
- **Phương thức on_start và on_stop:**
  - **on_start:** chạy khi một User bắt đầu hoặc TaskSet được thực thi (khi bắt đầu chạy test), dùng để chèn code về business logic như đăng nhập vào ứng dụng, khởi tạo test data, v.v.
  - **on_stop:** chạy sau khi User ngừng chạy hoặc khi TaskSet ngừng thực thi (khi kết thúc chạy test), dùng để chèn code về business logic như xoá test data, đăng xuất khỏi ứng dụng, v.v.
  - Mỗi phương thức chỉ chạy một lần duy nhất trong ca kiểm thử.

3.2.3. Events và EventHook

- **Events:** gồm các phương thức xử lý sự kiện có sẵn của Locust như **test_start**, **test_stop**, **on_locust_init**, **request_failure**, **request_success**, **reset_stats**, **user_error**, **report_to_master**, v.v.
- **EventHook:** lớp hỗ trợ tạo ra các event do người lập trình tự định nghĩa.

3.2.4. Cấu hình Locust

- Khi thực thi Locust file, các thông số như địa chỉ trang web cần test, số lượng user logic, thời gian chạy test, v.v, đều có thể được cấu hình ở nhiều nơi khác nhau nếu sử dụng giao diện dòng lệnh để chạy test. Các vị trí và thứ tự ưu tiên giá trị cấu hình từ độ ưu tiên thấp đến độ ưu tiên cao tương ứng gồm:
  - ~/locust.conf (file config ở home directory - thư mục Users/username).
  - ./locust.conf (file config của project hiện tại).
  - File config được chỉ định địa chỉ bằng tham số --conf khi chạy Locust file trên terminal.
  - Các giá trị config được gán trong các biến môi trường.
  - Giá trị các tham số config được gán trực tiếp khi chạy Locust file với terminal.

3.3. Danh sách demo

3.3.1. Hướng dẫn cài đặt

- **Link video:** https://youtu.be/guGoFRN-y9E

3.3.2. Demo khái niệm, kỹ thuật cơ bản của Locust

- **Link video:** https://youtu.be/btuWxuFEEdw

3.3.3. Demo đơn giản với Fahasa

- **Link video:** https://youtu.be/rgPsDfsQ-zc

3.3.4. Demo phức tạp với Gmail (tích hợp Selenium vào Locust)

- **Link video:** https://youtu.be/lQYfxxuD_fE