УДК 004.05
DOI https://doi.org/10.32782/2521-6643-2023.2-66.5

Зацерковний Р. Г., доктор філософії,
доцент кафедри комп'ютерних наук
Львівського торговельно-економічного університету
ORCID: 0000-0001-6991-2866

Бабич В. І., старший викладач кафедри комп'ютерних наук
Львівського торговельно-економічного університету
ORCID: 0000-0003-1996-9332

Плеша М. І., кандидат фізико-математичних наук, доцент,
доцент кафедри комп'ютерних наук
Львівського торговельно-економічного університету
ORCID: 0000-0001-5321-9602

Хмілярчук Л. І., старший викладач кафедри комп'ютерних наук
Львівського торговельно-економічного університету
ORCID: 0000-0002-1753-6472

Швець О. М., кандидат економічних наук, доцент,
доцент кафедри комп'ютерних наук
Львівського торговельно-економічного університету
ORCID: 0000-0002-7175-2256

СИСТЕМА ТЕСТУВАННЯ ПРОДУКТИВНОСТІ API ПРИ ВИСОКИХ НАВАНТАЖЕННЯХ

В основах багатьох сучасних ком'ютерних систем лежать інтерфейси прикладного програмування, або API (Application Programming Interface). Інтерфейси прикладного програмування слугують мостом між різними елементами комп'ютерних систем, дозволяючи їм спілкуватися, взаємодіяти між собою та безперешкодно обмінюватися даними. Фактично API – набір протоколів, інструментів і визначень, які дозволяють різним програмам інтегруватись між собою, а також надають розробникам можливість отримати доступ до певних програмних функцій або наборів даних.

Важливим завданням в процесі розробки API є забезпечення їх коректної роботи в умовах високих навантажень. API можуть мати сотні, тисячі і навіть мільйони користувачів, для яких важливе забезпечення стійкості, масштабованості і надійності цих програмних засобів. Система, не є надійним чином протестована в умовах високих навантажень, може зазнати краху у випадку, якщо велика кількість користувачів захоче водночас використати один і той самий метод API, або отримати схожі дані. Така система також буде менш стійкою від DDoS-атак – видів хакерських атак, що "засмічують" комп'ютерну систему великою кількістю запитів, фактично блокуючи доступ до неї для реальних користувачів.

У цій статті представляється дослідження та розробка системи тестування, створеної для оцінки продуктивності API. В основі системи лежить Locust, інструмент для тестування продуктивності програм з відкритим програмним кодом. Цей інструмент призначений для моделювання та аналізу поведінки різноманітних додатків при навантаженні великою кількістю користувачів. Розроблена система тестування продуктивності використовує можливості Locust для того, щоб ретельно вивчити стійкість API-системи у реальному сценарії. Вона детально тестує локальну версію API, навантажуючи її так званим "роєм" віртуальних користувачів. Після цього Locust збирає дані про продуктивність API – кількість успішних та помилкових запитів; мінімальний, максимальний та середній час обробки запиту; а також детальний звіт, який може бути використаний для подальшого аналізу результатів. Отримані в результаті такого дослідження дані сприяють розумінню і, в подальшому, покращенню якості роботи API, пропонуючи цінні рекомендації щодо оптимізації продуктивності програми та виявлення "вузьких місць" (bottlenecks) у програмному коді.

Ключові слова: *API (Application Programming Interface), тестування, тестування продуктивності, навантажувальне тестування, Locust.*

Zatserkovnyi R. G., Babych V. I., Plesha M. I., Khmilyarchuk L. I., Shvets O. M. A system to evaluate the robustness of an API under high loads

Many modern computer systems extensively use application programming interfaces, or APIs. APIs serve as a bridge between different elements of computer systems, allowing them to communicate, interact with each other, and exchange data

© Р. Г. Зацерковний, В. І. Бабич, М. І. Плеша, Л. І. Хмілярчук, О. М. Швець, 2023

seamlessly. Formally, an API is a set of protocols, tools, and definitions that allow different programs to integrate with each other, and provide developers with the ability to access certain software functions or data sets.

An important task in the process of developing APIs is to ensure their correct operation under high loads. APIs can have hundreds, thousands, and even millions of users, for whom it is important to ensure the stability, scalability, and reliability of these software tools. A system that has not been reliably tested under high load conditions may fail if a large number of users want to use the same API method or retrieve similar data at the same time. Such a system will also be less resistant to DDoS attacks, a type of malicious hacker attack that "clogs" a computer system with a large number of requests, effectively blocking access to it for legitimate users.

This article presents the research and development of a testing system designed to evaluate API performance. The system is based on Locust, an open-source tool for testing software under high loads. The tool is designed to model and analyze the behavior of various applications when facing a large number of users. Our performance testing system uses the capabilities of Locust to thoroughly study the stability of the API system in a real-world scenario. It tests a locally running version of an API in detail by loading it with a so-called "swarm" of virtual users. After this process is complete, Locust collects data on the performance of an API – number of successful and failed requests; minimum, maximum, and average request processing time; as well a detailed report that can be used for further analysis of the results. The data obtained as a result of such an analysis contributes to understanding and, subsequently, improving the quality of API performance, offering valuable recommendations for optimizing application performance and identifying bottlenecks in code.

Key words: *API (Application Programming Interface), testing, performance testing, load testing, Locust.*

Постановка проблеми. В основах багатьох сучасних ком'ютерних систем лежать інтерфейси прикладного програмування, або API (Application Programming Interface). Інтерфейси прикладного програмування слугують мостом між різними елементами комп'ютерних систем, дозволяючи їм спілкуватися, взаємодіяти між собою та безперешкодно обмінюватися даними. Фактично API – набір протоколів, інструментів і визначень, які дозволяють різним програмам інтегруватись між собою, а також надають розробникам можливість отримати доступ до певних програмних функцій або наборів даних. Крім того, API полегшують інтеграцію різноманітних додатків, дозволяючи розробникам інтегрувати дані з різних зовнішніх джерел. За версією Postman, до найпопулярніших API належать засоби системи керуванням взаємовідносин з користувачами Salesforce, нотатника Notion, та месенджера WhatsApp [1].

Важливим завданням в процесі розробки API є забезпечення їх коректної роботи в умовах високих навантажень. API можуть мати сотні, тисячі і навіть мільйони користувачів, для яких важливе забезпечення стійкості, масштабованості і надійності цих програмних засобів. Система, яка не є надійним чином протестована в умовах високих навантажень, може зазнати краху у випадку, якщо велика кількість користувачів захоче водночас використати один і той самий метод API, або отримати схожі дані. Така система також буде менш стійкою від DDoS-атак – видів хакерських атак, що "засмічують" комп'ютерну систему великою кількістю запитів, фактично блокуючи доступ до неї для реальних користувачів [2].

Аналіз останніх досліджень і публікацій. На сьогодні у низці досліджень описана процедура тестування API, або ж запропоновані нові ідеї та системи. У статті [3], представлено запропонований інструмент автоматизованого тестування API, а також проведено огляд літератури з автоматизованого тестування API. Автори статті [4] наводять систематичний огляд літератури, пов'язаної з тестуванням API. В процесі було знайдено та класифіковано різноманітні проблеми та рішення, пов'язані з тестуванням RESTful API (API, що призначені для Web і дотримуються набору певних стандартів) та створенням модульних тестів. Повертаючись до тестування комп'ютерних систем під великим навантаженням, в статті [5] описано особливий підхід до такого тестування. Автори аналізують журнали виконання програми (лог-файли), щоб виявити "нормальну" поведінку програми і, відповідно, незвичну, аномальну поведінку. Деякі автори описують процес тестування конкретних програмних систем – наприклад, автори у своїй роботі [6] застосовують методологію load testing (тестування з навантаженням) для перевірки програмно-визначених мережевих контролерів (SDN). Однак слід зазначити, що в дослідженнях рідко зустрічається тестування API власне на предмет стійкості в умовах високих навантажень.

Мета статті – дослідження та розробка системи тестування, створеної для оцінки продуктивності API. В основі системи лежить Locust, інструмент для тестування продуктивності програм з відкритим програмним кодом.

Виклад основного матеріалу. Багато видів комп'ютерного тестування стосуються перевірки окремих аспектів поведінки системи за різних умов. До таких видів відноситься *тестування продуктивності* (performance testing), т. зв. *навантажувальне тестування* (load testing), а також *стрес-тестування* (stress testing). Усі ці види тестування є невід'ємними частинами методології тестування програмного забезпечення, однак вони фокусуються на різних рівнях навантаження системи.

Тестування продуктивності оцінює роботу системи з точки зору продуктивності, швидкості відгуку, стабільності та масштабованості під очікуваним робочим навантаженням. Його основна мета – переконатися, що система відповідає заданим критеріям продуктивності. Тестування продуктивності може виконуватись в умовах, що близькі до стандартного рівня завантаженості системи. Однак окремі підтипи цього тестування – наприклад, навантажувальне тестування та стрес-тестування – фактично перевантажують комп'ютерну систему.

Навантажувальне тестування зосереджується на вивченні поведінки системи під впливом деякої заданої кількості одночасних користувачів, транзакцій або обсягів даних. Воно спрямоване на моделювання різноманітних сценаріїв використання системи, визначення її реальної пропускної здатності та обчислення ресурсів, які використовує система під деякими навантаженнями. Основна увага приділяється розумінню того, як система поводиться в пікових умовах, щоб переконатися, що вона може впоратися з високим навантаженням без значного погіршення продуктивності.

Стрес-тестування виходить за рамки навантажувального тестування, виводячи систему на межі її можливостей, фактично задаючи кількість одночасних запитів, на який вона не розрахована. Мета такого завідомо "провального тестування" – виявити точку зламу або граничні межі системи, і зрозуміти її поведінку при перевантаженні. Навантажувальне тестування допомагає виявити вразливості або слабкі місця, які можуть призвести до збою системи за надзвичайних обставин, таких як раптові сплески активності користувачів або ресурсів. Крім того, воно перевіряє, чи система адекватно реагує на перевантаження, належним чином відновлюється після перевантаження.

У випадку нашої системи найбільш актуальними видами тестування є тестування продуктивності (performance testing), а також навантажувальне тестування (load testing). Інакше кажучи, нас цікавить оцінювання швидкості роботи системи в цілому, а також перевірка роботи системи з кількістю користувачів, близькою до реальної. Більшість програмного забезпечення, і наша система в тому числі, покрита модульними тестами (unit tests) або інтеграційними тестами (integration tests), що перевіряють коректність та адекватність окремих елементів системи або ж усієї системи в цілому. Однак такі тести не можуть всебічно оцінити роботу системи з високою кількістю користувачів, адже зазвичай вони не паралелізовані і працюють лише з одним запитом водночас. Це і зумовлює необхідність розробки тестів з великою кількістю користувачів.

Для розробки тестів, що стосуються продуктивності, використовується середовище Locust. Ця комп'ютерна система – це інструмент для тестування продуктивності з відкритим вихідним кодом, призначений для оцінки масштабованості та продуктивності веб-додатків і API. Його основна мета – імітувати та генерувати різноманітні навантаження на систему, щоб проаналізувати, як системи поводяться в умовах високого трафіку трафіку. На відміну від багатьох традиційних інструментів навантажувального тестування, Locust дозволяє створювати тестові сценарії не за допомогою складних файлів конфігурації, а на основі коду на Python, що робить цей інструмент дуже гнучким і легким у налаштуванні.

Зазвичай навантажувальним тестуванням рекомендується проводити на локальних або тестових версіях комп'ютерної API-системи, щоб не навантажувати реальну систему без нагальної потреби. Припустимо, що такий API запущений на локальній машині, точніше, на хості localhost та порті 80 – традиційному розташуванні для web-серверів. Наступним кроком у тестуванні є інсталювання locust, і тут є два способи: за допомогою Pip, засобу для керування модулями на мові програмування Python:

```
pip3 install locust
```

Або ж за допомогою Docker – платформи, яка спрощує процес керування різноманітними додатками, "запаковуючи" програмну компоненту та усі її залежності в так званий "контейнер":

```
docker pull locustio/locust
```

Наступним кроком роботи з платформою Locust є розробка тестового файлу, що визначає поведінку користувачів. Така поведінка визначається у звичайному коді Python (в більшості схожих засобів тестування продуктивності використовується "domain-specific language" – особлива мова програмування, актуальна лише для роботи з конкретним засобом). Оскільки тестові файли є простими Python-файлами, для їх розробки можна використовувати будь-який зручний IDE (Visual Studio Code, PyCharm і т. д.), а для контролю їх версій – стандартну систему Git.

За версією документації Locust [8] приклад тестового файлу виглядає наступним чином:

```
import time
from locust import HttpUser, task, between
class QuickstartUser(HttpUser):
    wait_time = between(1, 5)
    @task
    def hello_world(self):
        self.client.get("/hello")
        self.client.get("/world")
    @task(3)
    def view_items(self):
        for item_id in range(10):
            self.client.get(f"/item?id={item_id}", name="/item")
            time.sleep(1)
    def on_start(self):
        self.client.post("/login", json={"username":"foo", "password":"bar"})
```

Як бачимо, тестовий файл є звичайним скріптом на мові програмування Python. Тому він може використовувати довільний код з будь-яких модулів, зовнішніх або вбудованих – наприклад, бібліотеку time. Клас QuickstartUser є похідним від HttpUser, і це вказує на те, що він здатний надсилати HTTP-запити до API або будь-якої іншої комп'ютерної системи, яку потрібно протестувати.

Рядок *wait_time = between(1, 5)* вказує на те, що інтервал між запитами, які надсилає система, обирається довільним чином в діапазоні від 1 до 5 секунд. (Це фактично симуляція поведінки реальних користувачів. Зазвичай в проміжку між різними запитами вони деякий час переглядають отриману ними інформацію, а не постійно навантажують систему.) Метод *on_start* викликається при створенні кожного віртуального користувача. А методи з анотацією *@task* – це програмний код, який запускається у так званому "грінлеті". Грінлет – це засіб асинхронного програмування в Python, що дозволяє водночас виконувати велику кількість завдань в програмному коді [9]. Таким чином, у прикладі з документації Locust два завдання: *hello_world*, що надсилає два статичні запити в кінцеві точки *hello* і *world*, та *view_items*, який надсилає десять запитів в кінцеву точку item з різноманітними значеннями параметру *item_id*.

Процедура виконання тестового файлу наступна: Locust створює деяку кількість віртуальних користувачів, яка зазначається в процесі виконання. Для кожного користувача спершу викликається метод *on_start*, а потім випадковим чином викликаються описані нами завдання, *hello_world* або *view_items*. В анотації методу *view_items* фігурує число 3 – це означає, що такий метод вибиратиметься втричі частіше, ніж зазвичай. Програма завершує роботу, коли вона вручну зупиняється користувачем, але можна задати і автоматичний критерій для її зупинки – наприклад, максимальну кількість запитів або максимальний час роботи. Приклад команди, що запускає Locust, наступний:

locust -f test.py --host localhost:80 --users 50 --spawn-rate 1 --headless -t 30m

Параметри цієї команди розшифровуються таким чином:

- *-f test.py* – розташування тестового файлу мовою Python, що визначає поведінку програми в цілому.
- *--host localhost:80* – місце розташування API, який тестується Locust.
- *--users 50* – кількість віртуальних користувачів, яких створить Locust. В даному випадку таких користувачів 50. Тобто водночас 50; цей процес, як було згадано раніше, організовується за допомогою так званих "грінлетів".
- *--spawn-rate 1* – швидкість створення користувачів. Locust починає з нуля, а потім додає користувачів зі швидкістю, яка залежить від *spawn-rate*. Число-значення параметру – кількість нових користувачів, що створюються щосекунди.
- *--headless* – параметр означає, що працює в режимі консольного додатку, а не web-інтерфейсу. Це дуже зручно у випадку, якщо потрібно автоматизувати роботу Locust, оскільки за замовчуванням його необхідно запускати вручну, з використанням спеціального сайту.
- *-t 30m* – критерій для завершення роботи програми. В даному випадку t вказує на те, що цей критерій – час (time), а 30m – на його тривалість, 30 хвилин.

У випадку прикладу документації, Locust завжди викликає одні і ті самі кінцеві точки API: *hello*, *world*, *item* з різноманітними ідентифікаторами, та *login* на початку роботи кожного користувача. Але зустрічаються випадки, коли потрібно повторювати саме ті запити, які використовують реальні користувачі – і в цьому випадку такий підхід не діятиме, оскільки в запитів бувають різні варіації та низка можливих параметрів. В такому випадку ми можемо використати метод відбору зразків, щоб точніше протестувати API, використовуючи зразки реальних запитів користувачів. Нехай низка запитів від користувачів – точніше кажучи, точні адреси кінцевих точок, до яких вони звертались – зібрані у файлі *sample.csv*. Тоді Python-файл може бути видозмінений таким чином:

```
from locust_plugins.csvreader import CSVReader
from locust import HttpUser, task, between
class CSVUser(HttpUser):
    wait_time = between(1, 5)
    csv_reader = CSVReader("sample.csv")
    @task
    def advance_through_csv(self):
        next_url = next(csv_reader)
        self.client.get(str(next_url[0]))
    def on_start(self):
        self.client.post("/login", json={"username":"foo", "password":"bar"})
```

У цьому прикладі Locust не звертатиметься до статичних кінцевих точок, а зчитуватиме дані з CSV-файлу. Таким чином, тестувальники можуть довільним чином змінювати кінцеві точки, які тестує засіб, без будь-якої зміни тестового файлу, що є зручним в умовах неперервної інтеграції (Continuous Integration).

Після завершення роботи очевидним наступним кроком є аналіз отриманих результатів. Для цього використовується параметр під назвою csv. У випадку задання цього параметру з деяким значенням – наприклад, *--csv=sample* – після завершення роботи Locust створить чотири файли: *sample_stats.csv*, *sample_failures.*

csv, *sample_exceptions.csv* та *sample_stats_history.csv*. Вміст першого файлу – статистика щодо результатів тестів на продуктивність: кількість усіх запитів, невдалих запитів, статистичні дані щодо тривалості запитів тощо. Другий та третій файли стосуються невдалих запитів або ж винятків у самому Locust. А останній файл – історія статистики, яка вказує на те, як різноманітні статистичні дані змінюються в процесі роботи програми [10]. Окремі статистичні дані можна виокремити та зберегти, зчитуючи рядки CSV-файлу. Він міститиме дані, що стосуються кожної кінцевої точки API, а також для усього тесту в цілому. Наприклад, файл *sample_stats.csv* можливо розшифрувати таким чином:

```
with open("sample_stats.csv", "r") as sample_stats:
    last_line = sample_stats.readlines()[-1].split(",")
    total_requests = final_line[2]
    failed_request = final_line[3]
    median_runtime = final_line[4]
    average_runtime = final_line[5]
    minimum_runtime = final_line[6]
    maximum_runtime = final_line[7]
    requests_per_second = final_line[9]
```

В подальшому дані, отримані таким чином, можна представити кінцевому користувачеві у зручно оформленому вигляді:

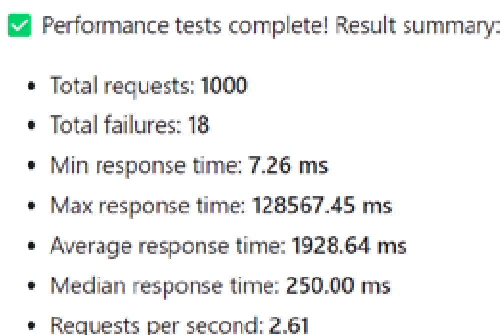

Рис. 1. Приклад подання результатів роботи тестів продуктивності на основі Locust.

Крім роботи в режимі консолі, Locust дозволяє працювати за допомогою візуального інтерфейсу. За замовчуванням цей інтерфейс є активним і розміщений за адресою localhost:8089, але адресу можна змінити, а інтерфейс – відключити.

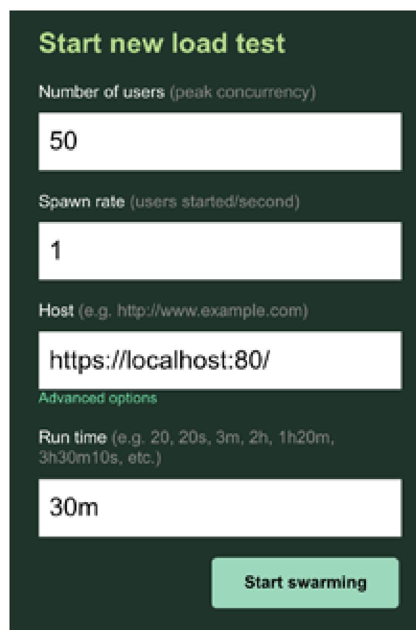

Рис. 2. Приклад візуального інтерфейсу засобу для тестування Locust. У цьому прикладі використовуються тісамі параметри, що вище описані в командному рядку.

Окрім надання можливості контролю параметрів програми Locust, web-інтерфейс дозволяє візуально проаналізувати роботу системи в режимі реального часу. У вкладці “Графіки” можна побачити низку показників: кількість запитів на одну секунду, середній час відгуку, а також кількість віртуальних користувачів.

Рис. 3. Приклади графіків Locust. Тут верхній графік – середній час відгуку запитів, а нижній – кількість віртуальних користувачів. Як бачимо, вона починається з нуля і з часом лінійно збільшується, поки не досягне максимуму.

Висновки. У статті представлена розробка системи, створеної для оцінки продуктивності API, на основі інструменту Locust. Описано можливості Locust, приклади роботи над тестовими файлами мовою Python, та команди, що нестандартним чином змінюють поведінку системи в цілому. Отримані в результаті такого дослідження дані сприяють розумінню і, в подальшому, покращенню якості роботи API, пропонуючи цінні рекомендації щодо оптимізації продуктивності програми та виявлення “вузьких місць” (bottlenecks) у програмному коді.

Список використаних джерел:

1. Most Popular APIs this year | Postman API Network. URL: https://www.postman.com/explore/most-popular-apis-this-year.
2. What is a distributed denial-of-service (DoS) attack? | Cloudflare. URL: https://www.cloudflare.com/learning/ddos/what-is-a-ddos-attack/.
3. Isha, Sharma A., Revathi M. Automated API Testing. 2018 3rd International Conference on Inventive Computation Technologies (ICICT), Coimbatore, India, 15–16 November 2018. 2018. URL: https://doi.org/10.1109/icict43934.2018.9034254.
4. RESTful API Testing Methodologies: Rationale, Challenges, and Solution Directions / A. Ehsan et al. Applied Sciences. 2022. Vol. 12, no. 9. P. 4369. URL: https://doi.org/10.3390/app12094369.
5. Automatic identification of load testing problems / Z. M. Jiang et al. 2008 IEEE International Conference on Software Maintenance (ICSM), Beijing, China, 28 September – 4 October 2008. 2008. URL: https://doi.org/10.1109/icsm.2008.4658079.
6. Latah M., Toker L. Load and stress testing for SDN’s northbound API. SN Applied Sciences. 2019. Vol. 2, no. 1. URL: https://doi.org/10.1007/s42452-019-1917-y.
7. Locust.io. Locust – A modern load testing framework. URL: https://locust.io/.
8. Writing a locustfile – Locust 2.18.4 documentation. Locust Documentation – Locust 2.18.4 documentation. URL: https://docs.locust.io/en/stable/writing-a-locustfile.html#writing-a-locustfile.
9. greenlet. PyPI. URL: https://pypi.org/project/greenlet/.
10. Retrieve test statistics in CSV format – Locust 2.18.4 documentation. Locust Documentation – Locust 2.18.4 documentation. URL: https://docs.locust.io/en/stable/retrieving-stats.html.

References:

1. Most Popular APIs this year | Postman API Network. URL: https://www.postman.com/explore/most-popular-apis-this-year.
2. What is a distributed denial-of-service (DoS) attack? | Cloudflare. URL: https://www.cloudflare.com/learning/ddos/what-is-a-ddos-attack/.

3. Isha, Sharma A., Revathi M. Automated API Testing. 2018 3rd International Conference on Inventive Computation Technologies (ICICT), Coimbatore, India, 15–16 November 2018. 2018. URL: https://doi.org/10.1109/icict43934.2018.9034254.

4. RESTful API Testing Methodologies: Rationale, Challenges, and Solution Directions / A. Ehsan et al. Applied Sciences. 2022. Vol. 12, no. 9. P. 4369. URL: https://doi.org/10.3390/app12094369.

5. Automatic identification of load testing problems / Z. M. Jiang et al. 2008 IEEE International Conference on Software Maintenance (ICSM), Beijing, China, 28 September – 4 October 2008. 2008. URL: https://doi.org/10.1109/icsm.2008.4658079.

6. Latah M., Toker L. Load and stress testing for SDN's northbound API. SN Applied Sciences. 2019. Vol. 2, no. 1. URL: https://doi.org/10.1007/s42452-019-1917-y.

7. Locust.io. Locust – A modern load testing framework. URL: https://locust.io/.

8. Writing a locustfile – Locust 2.18.4 documentation. Locust Documentation – Locust 2.18.4 documentation. URL: https://docs.locust.io/en/stable/writing-a-locustfile.html#writing-a-locustfile.

9. greenlet. PyPI. URL: https://pypi.org/project/greenlet/.

10. Retrieve test statistics in CSV format – Locust 2.18.4 documentation. URL: https://docs.locust.io/en/stable/retrieving-stats.html.