Testy wydajnościowe z Gatling

Łukasz Andrzejewski

Contents

Chapter 1

Wprowadzenie do testów wydajnościowych

1.1 Różnice między testami funkcjonalnymi a wydajnościowymi

Testowanie oprogramowania jest istotnym etapem procesu tworzenia aplikacji, mającym na celu wykrycie błędów, niezgodności oraz zagwarantowanie, że produkt końcowy spełni oczekiwania użytkowników. Wśród wielu typów testów wyróżnia się dwie podstawowe kategorie: testy funkcjonalne i testy wydajnościowe. Każda z nich bada zupełnie inne aspekty działania systemu i wymaga odmiennych metod oraz narzędzi.

Testy funkcjonalne skupiają się na sprawdzeniu, czy oprogramowanie realizuje określone funkcje zgodnie z dokumentacją wymagań. Obejmują one testowanie interakcji użytkownika z systemem, logiki biznesowej, walidacji danych, nawigacji oraz integracji między różnymi komponentami aplikacji. Przykładowo, w aplikacji e-commerce test funkcjonalny może obejmować proces dodawania produktów do koszyka, składania zamówienia, czy rejestracji nowego konta. Celem tych testów jest zapewnienie, że każda funkcjonalność działa dokładnie tak, jak została zaprojektowana, niezależnie od liczby użytkowników czy warunków technicznych.

Testy funkcjonalne są często przeprowadzane w sposób manualny lub automatyczny, z użyciem narzędzi takich jak Selenium, TestComplete czy Cypress. Testerzy sprawdzają poprawność działania przy różnych danych wejściowych, oczekując konkretnych wyników. Kluczowym elementem tych testów jest więc porównanie rzeczywistych rezultatów z oczekiwanymi, zgodnymi z wymaganiami.

Testy wydajnościowe, z kolei, analizują, jak system zachowuje się pod względem szybkości, stabilności, skalowalności oraz zużycia zasobów. Ich głównym celem nie jest sprawdzenie funkcjonalności, lecz tego, jak efektywnie i niezawodnie aplikacja działa w warunkach rzeczywistego lub zwiększonego obciążenia. W tym zakresie wyróżnia się kilka podtypów testów, takich jak testy obciążeniowe (load testing), stresowe (stress testing), czy testy długoterminowej stabilności (soak testing).

Przykładem może być sytuacja, w której setki lub tysiące użytkowników jednocześnie korzystają z platformy sprzedażowej podczas dużej promocji. Testy wydajnościowe pozwalają ocenić, czy serwer poradzi sobie z nagłym wzrostem ruchu, czy aplikacja nie zacznie działać z opóźnieniem, a także czy nie dojdzie do awarii systemu. Do tego typu testów wykorzystuje się specjalistyczne narzędzia, takie jak Apache JMeter, Gatling, LoadRunner czy k6, które pozwalają symulować duże obciążenie i analizować różne metryki techniczne.

Różnice między tymi dwoma rodzajami testów wynikają nie tylko z ich celu, ale także z podejścia do testowania i wykorzystywanych narzędzi. Testy funkcjonalne koncentrują się na „co" system robi, podczas gdy testy wydajnościowe analizują „jak dobrze" system to robi w różnych warunkach. Obie formy testowania są niezbędne — testy funkcjonalne pomagają upewnić się, że system działa zgodnie z wymaganiami, a testy wydajnościowe sprawdzają, czy jest w stanie sprostać rzeczywistemu użytkowaniu.

1.2 Podstawowe pojęcia w testach wydajnościowych

Testy wydajnościowe są nieodzownym elementem oceny jakości systemów informatycznych, szczególnie w kontekście ich zachowania pod obciążeniem. Aby móc poprawnie analizować wyniki takich testów, niezbędne jest zrozumienie podstawowych pojęć i metryk, które opisują wydajność systemu. Oto najważniejsze z nich:

Throughput (przepustowość)

Throughput określa, ile operacji (np. żądań HTTP, transakcji) system jest w stanie przetworzyć w jednostce czasu, zazwyczaj wyrażanej w sekundach lub minutach. Jest to miara "wydajności ogólnej" systemu – im większy throughput, tym więcej pracy system wykonuje w danym czasie. Przykładowo, serwer obsługujący 500 żądań na sekundę ma throughput równy 500 RPS (requests per second).

Latency (opóźnienie)

Latency to czas, jaki upływa od momentu wysłania żądania do momentu, gdy jego przetwarzanie zostanie rozpoczęte przez system. Nie należy mylić tego pojęcia z czasem odpowiedzi. Latencja pokazuje, jak szybko system zaczyna reagować na żądanie – może być zależna od kolejek, obciążenia procesora, sieci itp. Wysoka latencja może oznaczać, że system jest przeciążony lub niewydolny na etapie przyjmowania żądań.

Response time (czas odpowiedzi)

Response time to całkowity czas od momentu wysłania żądania przez użytkownika do otrzymania pełnej odpowiedzi. Obejmuje zarówno latencję, jak i czas potrzebny na przetworzenie żądania oraz przesłanie odpowiedzi. Jest to kluczowy wskaźnik wpływający bezpośrednio na doświadczenie użytkownika. Na przykład, jeśli kliknięcie w przycisk „Kup teraz" powoduje odpowiedź po 8 sekundach, response time wynosi 8 sekund.

Percentyle

Percentyle pomagają zrozumieć rozkład czasów odpowiedzi wśród wszystkich wykonanych żądań. Najczęściej używa się percentyla 90 (P90), 95 (P95) i 99 (P99). Percentyl 95 oznacza, że 95% żądań zostało obsłużonych szybciej lub równo czasowi podanemu dla P95. Jeśli P95 wynosi 2 sekundy, oznacza to, że tylko 5% żądań trwało dłużej niż 2 sekundy. Percentyle są bardziej miarodajne niż średnie wartości, ponieważ uwzględniają nierównomierność wyników.

TPS (Transactions per Second)

TPS to liczba pełnych transakcji (czyli np. cały proces: dodanie do koszyka → przejście do kasy → płatność) przetworzonych w ciągu jednej sekundy. W odróżnieniu od RPS (requests per second), które mierzy pojedyncze żądania, TPS odnosi się do większych operacji składających się z wielu kroków. Jest szczególnie użyteczny przy testowaniu aplikacji biznesowych i e-commerce.

Użytkownicy wirtualni (Virtual Users, VUs)

Użytkownicy wirtualni to symulowani użytkownicy systemu, generowani przez narzędzia do testowania wydajności. Każdy użytkownik wykonuje określone operacje w aplikacji, często w pętli, odwzorowując rzeczywiste scenariusze użytkowania. Dzięki nim można sprawdzić, jak system radzi sobie z równoczesnym dostępem setek lub tysięcy użytkowników. Użytkownicy wirtualni nie są prawdziwymi osobami — to w pełni zautomatyzowane instancje klienta testowego.

1.3 Typy testów wydajnościowych

Testy wydajnościowe służą do oceny, jak aplikacja zachowuje się pod różnymi rodzajami obciążenia. W zależności od celu testowania oraz scenariusza użytkowania, wyróżnia się kilka typów testów: **load**, **stress**, **spike** i **soak**. Każdy z nich bada inne właściwości systemu i pozwala wykryć inne klasy problemów.

Load testing (testy obciążeniowe)

Cel: sprawdzenie, jak system działa przy przewidywanym, normalnym obciążeniu.

Test obciążeniowy polega na symulowaniu typowego ruchu użytkowników w celu oceny wydajności aplikacji w warunkach „codziennego" użytkowania. Dzięki temu można określić, czy czas odpowiedzi, przepustowość i zachowanie systemu spełniają wymagania przy standardowym obciążeniu – np. 500 użytkowników równocześnie wykonujących zakupy w e-sklepie.

Zastosowanie:

- weryfikacja wydajności przy spodziewanej liczbie użytkowników,
- testowanie konfiguracji systemu i jego infrastruktury,
- wykrywanie wąskich gardeł (np. powolne zapytania do bazy danych).

Stress testing (testy przeciążeniowe)

Cel: sprawdzenie, jak system zachowuje się po przekroczeniu jego maksymalnych możliwości.

Test stresowy ma na celu symulację sytuacji ekstremalnej – zwiększania obciążenia ponad normalne wartości aż do momentu awarii systemu lub znacznego spadku wydajności. Pozwala to zidentyfikować maksymalny próg wydolności oraz sposób, w jaki system „się psuje".

Zastosowanie:

- określenie granicy wytrzymałości systemu,
- analiza odporności na przeciążenie,
- sprawdzenie, czy system wychodzi z awarii w kontrolowany sposób (np. restart usług, komunikaty błędów).

Spike testing (testy skokowe)

Cel: sprawdzenie reakcji systemu na nagły, gwałtowny wzrost liczby użytkowników.

Testy skokowe polegają na chwilowym, znaczącym zwiększeniu liczby użytkowników lub żądań w bardzo krótkim czasie, a następnie obserwacji, jak system reaguje – czy nadal działa poprawnie, jak długo przetwarza żądania i czy szybko wraca do normy po spadku obciążenia.

Zastosowanie:

- symulacja nagłego ruchu (np. promocje, kampanie reklamowe),
- testowanie autoskalowania,
- ocena odporności na „skoki" w ruchu.

Soak testing (testy długotrwałe, stabilnościowe)

Cel: ocena stabilności i zachowania systemu przy ciągłym obciążeniu przez długi czas.

Testy typu soak trwają wiele godzin, a czasem nawet dni, i polegają na utrzymaniu stałego, umiarkowanego obciążenia. Ich celem jest wykrycie problemów, które ujawniają się dopiero po dłuższym czasie działania – np. wycieki pamięci, stopniowy spadek wydajności, błędy w kolejkowaniu zadań.

Zastosowanie:

- sprawdzenie zarządzania zasobami (RAM, CPU),
- wykrycie degradacji działania po czasie,
- testowanie stabilności systemów krytycznych (np. bankowość, telekomunikacja).

1.4 Znaczenie testów wydajnościowych w cyklu życia oprogramowania

Testy wydajnościowe odgrywają istotną rolę w całym cyklu życia oprogramowania (SDLC – Software Development Life Cycle). Choć często postrzegane są jako działania końcowe, mają one znaczenie na wielu etapach rozwoju systemu – od planowania, przez projektowanie, aż po wdrożenie i utrzymanie. Ich głównym celem jest zapewnienie, że system nie tylko działa poprawnie, ale również działa **szybko, stabilnie i efektywnie**, nawet przy dużym obciążeniu.

Etap planowania i analizy wymagań

Już na wczesnym etapie projektu powinny być zdefiniowane wymagania niefunkcjonalne dotyczące wydajności:

- Jaki ma być maksymalny czas odpowiedzi?
- Ilu użytkowników jednocześnie ma obsługiwać system?
- Jak długo aplikacja ma działać bez restartu?

Testy wydajnościowe pomagają przekształcić te oczekiwania w konkretne, mierzalne cele, które będą weryfikowane w dalszych fazach.

Faza projektowania i architektury

Wydajność powinna być uwzględniona w decyzjach architektonicznych. Testy przeprowadzane w środowiskach testowych mogą pomóc w wyborze optymalnych rozwiązań (np. baz danych, systemów kolejkowych, mechanizmów cache). Na tym etapie można również projektować aplikację w sposób bardziej skalowalny i odporny na przeciążenie.

Etap implementacji i testowania funkcjonalnego

W trakcie rozwoju aplikacji można wdrażać **ciągłe testowanie wydajności** (ang. performance testing in CI/CD). Umożliwia to wykrycie regresji wydajności – czyli sytuacji, w których nowy kod spowalnia działanie systemu. Testy te mogą być wykonywane cyklicznie w ramach automatyzacji, na przykład po każdym mergowaniu zmian.

Faza testów końcowych i przedprodukcyjnych

To kluczowy moment, w którym należy przeprowadzić pełne testy wydajnościowe:

- **Load testing** – sprawdzenie działania pod przewidywanym obciążeniem,
- **Stress testing** – analiza odporności na przekroczenie limitów,
- **Spike testing** – symulacja nagłych wzrostów ruchu,
- **Soak testing** – weryfikacja długoterminowej stabilności.

Ich wyniki pozwalają ocenić gotowość systemu do wdrożenia i zminimalizować ryzyko awarii po premierze.

Etap wdrożenia i utrzymania

Po wdrożeniu do środowiska produkcyjnego testy wydajnościowe nadal mają znaczenie – zwłaszcza w systemach o dużym ruchu i wysokiej dostępności. Można je wykorzystywać do:

- porównywania wyników testowych z danymi rzeczywistymi (monitoring),
- weryfikacji, czy nowe wersje nie wprowadzają regresji,
- planowania skalowania systemu na przyszłość.

W tym kontekście testy wydajnościowe wspierają **DevOps** i **SRE (Site Reliability Engineering)**, umożliwiając świadome zarządzanie wydajnością i kosztami operacyjnymi.

Chapter 2

Wprowadzenie do Gatling

Gatling to nowoczesne, wydajne narzędzie do testowania wydajności aplikacji webowych i API. Zostało zaprojektowane z myślą o wysokiej wydajności, elastyczności i łatwości automatyzacji, co czyni je popularnym wyborem w środowiskach DevOps i CI/CD. Gatling pozwala symulować setki tysięcy wirtualnych użytkowników z minimalnym użyciem zasobów, dzięki oparciu o asynchroniczne przetwarzanie i język Scala.

2.1 Architektura Gatlinga

Gatling zbudowany jest w oparciu o **reaktywną architekturę** i działa w modelu **asynchronicznym**, co odróżnia go od wielu narzędzi działających w sposób wątkowy (np. JMeter). Oto kluczowe komponenty architektury Gatlinga:

Core (silnik)

- Napisany w Scali i oparty o bibliotekę **Akka**, co umożliwia obsługę dużej liczby wirtualnych użytkowników bez tworzenia tysięcy wątków.
- Wykorzystuje model aktorów do przetwarzania zdarzeń w sposób skalowalny i niskokosztowy.

DSL (Domain-Specific Language)

- Gatling korzysta z własnego, przejrzystego **DSL-a w języku Scala**, umożliwiającego programistom opisanie scenariuszy testowych jako kodu.
- DSL jest zwięzły i intuicyjny, pozwala łatwo opisać złożone scenariusze: np. pętle, warunki, feeder'y z danymi wejściowymi.

Recorder

- Graficzne narzędzie do nagrywania ruchu HTTP/HTTPS, które pozwala łatwo wygenerować szkielet testów.
- Umożliwia użytkownikom nieznającym Scali stworzenie bazowego scenariusza poprzez „klikane" działania w przeglądarce.

Feeder'y danych

- Gatling wspiera odczyt danych wejściowych z różnych źródeł (CSV, JSON, bazy danych), co pozwala symulować unikalne żądania użytkowników.

Raportowanie

- Po wykonaniu testu Gatling generuje **szczegółowy raport HTML**, który pokazuje:
  - czasy odpowiedzi,
  - throughput (RPS/TPS),
  - percentyle (np. P95, P99),
  - liczbę błędów,

- rozkład czasowy żądań.

2.2 DSL Gatlinga – jak wygląda?

Przykład prostego scenariusza w Gatling DSL:

```
class BasicSimulation extends Simulation {

  val httpProtocol = http
    .baseUrl("https://example.com")
    .acceptHeader("application/json")

  val scn = scenario("Basic Test Scenario")
    .exec(http("Request_1")
      .get("/api/data"))
    .pause(1)

  setUp(
    scn.inject(atOnceUsers(100)) // symulacja 100 użytkowników równocześnie
  ).protocols(httpProtocol)
}
```

Wyjaśnienie:

- `httpProtocol` – konfiguracja protokołu HTTP.
- `scenario` – definicja scenariusza użytkownika.
- `exec` – wykonanie konkretnego żądania HTTP.
- `inject` – strategia obciążenia (np. 100 użytkowników jednocześnie).
- `setUp` – konfiguracja uruchomienia testu.

Zalety Gatlinga

- Niewielkie zużycie zasobów (idealne do CI/CD).
- Przejrzysty, kodowalny DSL (łatwa automatyzacja).
- Bardzo szczegółowe raporty HTML.
- Wsparcie dla testów HTTP/REST, WebSocket, JMS.
- Integracja z Jenkins, Maven, Gradle, Dockerem.

2.3 Gatling Open Source vs Gatling Enterprise

Gatling występuje w dwóch głównych wersjach: Gatling Open Source oraz Gatling Enterprise (dawniej Gatling FrontLine). Obie wersje opierają się na tym samym silniku testowym i zapewniają wysoką wydajność oraz wsparcie dla zaawansowanego DSL-a opartego na Scali. Różnią się jednak zakresem funkcji, integracjami, poziomem wsparcia technicznego oraz przeznaczeniem – Open Source skierowane jest głównie do indywidualnych użytkowników i mniejszych zespołów, natomiast wersja Enterprise odpowiada na potrzeby firm wdrażających testy w dużych środowiskach produkcyjnych lub CI/CD.

Gatling Open Source to w pełni funkcjonalne, darmowe narzędzie umożliwiające lokalne tworzenie, uruchamianie i analizowanie testów wydajnościowych. Użytkownicy mają dostęp do bogatego DSL-a w Scali, mogą pisać własne scenariusze, generować szczegółowe raporty HTML oraz korzystać z narzędzia do nagrywania testów (Gatling Recorder). Open Source sprawdza się bardzo dobrze w małych projektach, projektach typu proof of concept, lub w przypadku indywidualnych testerów i deweloperów, którzy chcą mieć pełną kontrolę nad kodem testów.

Gatling Enterprise to rozwiązanie komercyjne, które rozszerza możliwości wersji Open Source o funkcje przeznaczone dla organizacji pracujących z dużą skalą i złożonością testów. Umożliwia rozproszone wykonywanie testów na wielu węzłach, zarządzanie

nimi z poziomu graficznego interfejsu użytkownika oraz integrację z systemami CI/CD (takimi jak Jenkins, GitLab, Bamboo). Enterprise oferuje także monitoring w czasie rzeczywistym, zarządzanie użytkownikami, historię testów i zaawansowane alerty. Dodatkowo, firmy korzystające z tej wersji mają dostęp do wsparcia technicznego i konsultacji ze strony twórców Gatlinga.

Podczas gdy Gatling Open Source świetnie sprawdzi się w mniejszych projektach i w pracy lokalnej, Gatling Enterprise jest wybierany przez zespoły, które potrzebują centralnego zarządzania testami, lepszej integracji z istniejącą infrastrukturą i profesjonalnego wsparcia. Wersja komercyjna szczególnie dobrze sprawdza się w środowiskach korporacyjnych, gdzie niezawodność i automatyzacja na dużą skalę są priorytetem.

2.4 Główne komponenty Gatlinga

Gatling to narzędzie do testów wydajnościowych oparte na kodzie, w którym każdy test jest de facto programem. Jego struktura jest czytelna i modułowa – składa się z kilku głównych komponentów, które razem pozwalają zdefiniować realistyczne, mierzalne scenariusze obciążeniowe. Poniżej przedstawiam najważniejsze elementy: Simulation, Scenario, Injection Profile oraz Checks.

Simulation

`Simulation` to główny komponent testu Gatlinga – klasa w języku Scala, która zawiera pełną definicję testu wydajnościowego. To właśnie w tym miejscu określamy, jakie scenariusze mają zostać uruchomione, jak mają być wstrzykiwani użytkownicy, jaki protokół ma zostać użyty (HTTP, WebSocket itp.) oraz jak wygląda konfiguracja testu (czas trwania, liczba użytkowników, tempo wzrostu obciążenia).

Przykład definicji klasy:

```
class MySimulation extends Simulation {
  // konfiguracja protokołu, scenariusz, injection i setUp
}
```

`Simulation` pełni rolę „planu głównego” – scala wszystkie pozostałe komponenty w spójną całość.

Scenario

`Scenario` (czyli scenariusz) to logiczny opis zachowania wirtualnego użytkownika – sekwencja kroków, które symulują działania użytkownika w aplikacji. Może to być logowanie, wyszukiwanie, przeglądanie produktów, dodawanie do koszyka i zakup. Każdy krok to zazwyczaj jedno żądanie HTTP, a między nimi można dodawać opóźnienia, pętle, rozgałęzienia warunkowe czy losowanie danych.

Przykład:

```
val scn = scenario("User Behavior")
  .exec(http("HomePage").get("/"))
  .pause(2)
  .exec(http("Search").get("/search?q=gatling"))
```

Scenariusze mogą być wielokrotnie wykorzystywane z różnymi profilami użytkowników i danymi.

Injection Profile

`Injection Profile` definiuje sposób i tempo, w jakim użytkownicy wirtualni są „wstrzykiwani” do scenariusza. Jest to kluczowe dla symulacji realistycznych warunków obciążenia. Gatling oferuje wiele strategii iniekcji, np.:

- `atOnceUsers(n)` – jednoczesne uruchomienie n użytkowników,
- `rampUsers(n) during (t)` – stopniowe zwiększanie liczby użytkowników przez t jednostek czasu,
- `constantUsersPerSec(n) during (t)` – stała liczba użytkowników na sekundę,
- `heavisideUsers(n) during (t)` – użytkownicy dodawani zgodnie z funkcją krzywej Heaviside’a.

Przykład użycia w `setUp`:

```
setUp(
  scn.inject(rampUsers(100).during(30.seconds))
)
```

Dzięki injection profile możemy symulować zarówno stałe obciążenie, jak i gwałtowne wzrosty lub długotrwałe użytkowanie.

Checks

`Checks` to asercje wykonywane na odpowiedziach serwera, które pozwalają sprawdzić, czy odpowiedź jest poprawna – nie tylko pod względem technicznym (np. kod 200), ale także logicznym (np. zawartość odpowiedzi JSON). Są kluczowe, by upewnić się, że aplikacja działa prawidłowo także przy wysokim obciążeniu.

Przykład:

```
.exec(http("GetProduct")
  .get("/api/product/123")
  .check(status.is(200))
  .check(jsonPath("$.name").is("Gatling Book")))
```

Dostępne są różne typy `checks` – m.in. na kod statusu, nagłówki, treść HTML, JSONPath, Regex. Błędy w `checks` są rejestrowane jako nieudane żądania w raporcie.

Podsumowując, Gatling opiera się na przejrzystej strukturze: `Simulation` definiuje test jako całość, `Scenario` opisuje zachowanie użytkownika, `Injection Profile` określa sposób obciążenia, a `Checks` weryfikują poprawność odpowiedzi. Takie podejście pozwala budować elastyczne, skalowalne i łatwe do utrzymania testy wydajnościowe.