Gatling Performance Testing – Notes & Practical Guide

1. What is Gatling?

Gatling is a **high-performance load and stress testing tool** written in Scala and built on **Akka** and **Netty**. It is designed to test HTTP-based systems with a strong focus on:

- Asynchronous, non-blocking execution
- High throughput with low resource usage
- Readable, code-based test definitions
- Rich HTML reports

Typical use cases: - Load testing REST APIs - Stress testing microservices - Soak testing backend systems

---

2. Gatling Architecture (High Level)

- **Simulation** – Entry point for a test
- **Scenario** – User journey definition
- **Protocol** – HTTP configuration (base URL, headers, auth, etc.)
- **Injection Profile** – How virtual users are injected
- **Assertions** – Pass/fail criteria

---

3. Project Setup (Maven)

```
<dependency>
  <groupId>io.gatling.highcharts</groupId>
  <artifactId>gatling-charts-highcharts</artifactId>
  <version>3.10.5</version>
  <scope>test</scope>
</dependency>
```

```
<plugin>
  <groupId>io.gatling</groupId>
  <artifactId>gatling-maven-plugin</artifactId>
  <version>4.9.5</version>
</plugin>
```

Standard directory structure:

```
src
 └── test
     ├── scala
     │   └── simulations
     └── resources
         └── gatling.conf
```

---

4. Basic Gatling Test Case (Simulation)

4.1 HTTP Protocol Configuration

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

4.2 Scenario Definition

```
val scn = scenario("Get Users")
  .exec(
    http("Get All Users")
      .get("/users")
      .check(status.is(200))
  )
```

4.3 Load Injection

```
setUp(
  scn.inject(
    rampUsers(100).during(30)
  )
).protocols(httpProtocol)
```

---

5. Common Gatling DSL Elements

HTTP Methods

```
.get("/path")
.post("/path").body(StringBody("{}"))
.put("/path")
.delete("/path")
```

Checks

```
.check(status.is(200))
.check(jsonPath("$.id").exists)
.check(bodyString.saveAs("responseBody"))
```

Feeders (Test Data)

```
val feeder = csv("users.csv").circular

.exec(
  http("Login")
    .post("/login")
    .body(StringBody("""
      { "username": "${username}", "password": "${password}" }
    """))
)
```

---

6. Assertions (Test Pass / Fail Criteria)

```
assertions(
  global.responseTime.max.lt(2000),
  global.successfulRequests.percent.gt(99)
)
```

---

7. Running Gatling Tests

Maven

```
mvn gatling:test
```

Specific Simulation

```
mvn gatling:test -Dgatling.simulationClass=simulations.GetUsersSimulation
```

---

8. Reports

After execution, Gatling generates:

- Response time distribution
- Requests per second
- Error percentage
- Percentiles (50th, 75th, 95th, 99th)

Reports are stored under:

```
target/gatling/
```

---

9. Why Integrate Gatling with Cucumber?

Cucumber allows:

- Business-readable scenarios (Gherkin)
- Collaboration with non-technical stakeholders
- Reusable step definitions

Gatling handles **performance**, Cucumber handles **behavior definition**.

---

10. Cucumber + Gatling Integration – Architecture

```
Feature File (.feature)
   ↓
Step Definitions (Scala/Java)
```

```
   ↓
Gatling Scenario
   ↓
Load Injection + Assertions
```

---

11. Dependencies for Cucumber Integration

```
<dependency>
  <groupId>io.cucumber</groupId>
  <artifactId>cucumber-scala</artifactId>
  <version>8.21.1</version>
  <scope>test</scope>
</dependency>

<dependency>
  <groupId>io.cucumber</groupId>
  <artifactId>cucumber-junit</artifactId>
  <version>8.21.1</version>
  <scope>test</scope>
</dependency>
```

---

12. Sample Feature File

```
Feature: User API Performance

  Scenario: Load test get users API
    Given base url is "https://api.example.com"
    When 100 users ramp up in 30 seconds
    Then GET "/users" should return 200
```

---

13. Step Definitions (Scala)

```
class UserApiSteps {

  var baseUrl: String = _
  var users: Int = 0
  var duration: Int = 0
```

```
  Given("base url is {string}") { (url: String) =>
    baseUrl = url
  }

  When("{int} users ramp up in {int} seconds") { (u: Int, d: Int) =>
    users = u
    duration = d
  }

  Then("GET {string} should return {int}") { (path: String, statusCode: Int) =>
    GatlingRunner.run(baseUrl, path, users, duration, statusCode)
  }
}
```

---

14. Gatling Runner Utility

```
object GatlingRunner {

  def run(baseUrl: String, path: String, users: Int, duration: Int, statusCode:
Int): Unit = {

    val httpProtocol = http.baseUrl(baseUrl)

    val scn = scenario("Cucumber Driven Scenario")
      .exec(
        http("Request")
          .get(path)
          .check(status.is(statusCode))
      )

    setUp(
      scn.inject(rampUsers(users).during(duration))
    ).protocols(httpProtocol)
  }
}
```

---

15. Best Practices

- Keep **Gatling logic independent** of Cucumber
- Use Cucumber only for orchestration
- Avoid heavy assertions inside Cucumber steps
- Parameterize users, duration, URLs

• Version control performance baselines

---

16. When NOT to Use Cucumber with Gatling

• Pure load tests with no business involvement
• Large-scale tests (100k+ users)
• CI pipelines focused only on metrics

---

17. Summary

• Gatling is code-centric and highly scalable
• Scenarios define user journeys
• Injection controls load behavior
• Assertions enforce SLAs
• Cucumber adds readability and collaboration
• Integration should remain lightweight

Next topics (optional): - JWT / OAuth performance testing - Correlation handling - Distributed Gatling execution - CI/CD integration (GitLab, Jenkins)