**Gatling** is a powerful open-source load and performance testing tool for web applications. Built primarily on Scala (though it also supports writing tests in Java and Kotlin), Gatling is designed to be highly performant, with a focus on asynchronous and non-blocking IO. This makes it possible to simulate many concurrent users (virtual users) on relatively modest hardware.

---

## 1. Why Gatling?

- **High performance**: Gatling uses an asynchronous, event-driven approach, enabling it to handle more concurrent users than many other load testing tools on the same hardware.
- **Scala, Java, Kotlin DSLs**: You can write Gatling scripts in Scala by default, but it also supports Java and Kotlin, making it easy to adopt in diverse tech stacks.
- **HTML reports**: Gatling automatically produces rich graphical reports showing response times, throughput, percentiles, and more.
- **Extensibility**: Gatling can be extended to handle different protocols (e.g., HTTP, JMS) and can be integrated into CI/CD pipelines for continuous performance testing.

---

## 2. Getting Started

### 2.1 Installation

1. **Download the Gatling bundle**:
   - Visit Gatling's official website and download the latest open-source bundle (e.g., gatling-charts-highcharts-bundle-<version>.zip).
2. **Unzip** the downloaded file to a preferred directory.
3. **Run Gatling**:
   - On Linux/macOS:

```
cd gatling-charts-highcharts-bundle-<version>/bin
./gatling.sh
```

   - On Windows:

```
cd gatling-charts-highcharts-bundle-<version>/bin
gatling.bat
```

## 2.2 Project Structure

Once unzipped, the Gatling bundle contains several folders:

- bin/ – Scripts to run Gatling.
- conf/ – Configuration files (e.g., gatling.conf).
- lib/ – Gatling's library files.
- results/ – Default folder for storing simulation results.
- user-files/ – Holds your simulation files (simulations/) and resources (resources/).

If you prefer a Maven or Gradle setup, you can integrate Gatling as a plugin/dependency in your existing projects.

- Maven Plugin Reference
- Gradle Plugin Reference

---

# 3. Gatling Test Script Basics

Gatling scripts (simulations) are typically written in **Scala** and follow a certain structure. Below is a simplified overview:

1. **Package and Imports**:

```
package simulations

import io.gatling.core.Predef._
import io.gatling.http.Predef._
import scala.concurrent.duration._
```

2. **HTTP Protocol Configuration**:

```
val httpProtocol = http
  .baseUrl("https://example.com") // Base URL
  .acceptHeader("text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8")
  .userAgentHeader("Gatling Test")
```

3. **Scenario Definition**:

```
val scn = scenario("Basic Simulation")
  .exec(
    http("Homepage")
      .get("/")
      .check(status.is(200))
  )
```

4. **Simulation Setup**:

```
setUp(
  scn.inject(
    atOnceUsers(10)  // Inject 10 users immediately
  )
).protocols(httpProtocol)
```

## 3.1 Key Concepts

- **Scenario**: Defines the user journey or steps a virtual user will follow.
- **Exec**: Wraps individual requests or chains of requests.
- **Check**: Assertions on the response, e.g., verifying HTTP status codes or response bodies.
- **Injection Profiles**: How users are injected into the system (e.g., atOnceUsers, rampUsers, constantUsersPerSec, etc.).

---

# 4. Core DSL Components

## 4.1 HTTP DSL

Gatling's DSL for HTTP provides a straightforward way to describe requests:

- **http("requestName"):** Name of the request, used in reports.
- **get("/path")** or **post("/path"):** The HTTP method and endpoint.
- **Request body** (for POST/PUT/PATCH):

```
.body(StringBody("""{ "name": "test" }""")).asJson
```

- **Headers**:

```
.header("Content-Type", "application/json")
```

- **Checks**:

```
.check(status.is(200))
.check(jsonPath("$.id").saveAs("userId"))
```

## 4.2 Checks and Assertions

- **Checks** are used inside requests to capture data from the response or validate response status/body.
- **Assertions** are defined at the end of the simulation to declare pass/fail criteria:

```
setUp(
  scn.inject(rampUsers(100).during(30.seconds))
).protocols(httpProtocol)
 .assertions(
   global.responseTime.max.lt(1000),
   forAll.successfulRequests.percent.gt(95)
 )
```

### 4.3 Feeder (Data-Driven Testing)

A **feeder** in Gatling provides data to the virtual users, allowing you to parameterize requests:

```
val csvFeeder = csv("data/users.csv").circular

val scn = scenario("Data-Driven Scenario")
  .feed(csvFeeder)
  .exec(
    http("Get user info")
      .get("/users/${userId}")
      .check(status.is(200))
  )
```

- **Types of feeders**: CSV, JSON, JDBC, custom Scala code, etc.
- **Feeder strategies**: queue, random, shuffle, and circular.

---

## 5. Advanced Topics

### 5.1 Sessions and Session Handling

Gatling maintains a **Session** object for each virtual user. This session can store variables, typically retrieved from feeders or checks.

- Access session data in your scenario:

```
.exec { session =>
  val userId = session("userId").as[String]
  println(userId)
  session
}
```

### 5.2 Custom Functions and Reusable Code

You can create **reusable code snippets** in Gatling by defining:

- **Functions** or **Methods** in your Scala simulation files.
- **Chains** of requests that can be .exec()ed multiple times.

For example:

```
def loginUser = exec(
  http("Login")
    .post("/login")
    .body(StringBody("""{ "username": "foo", "password": "bar" }""")).asJson
    .check(status.is(200))
)

val scn = scenario("Login and Fetch Data")
  .exec(loginUser)
  .exec(
    http("Get Dashboard")
      .get("/dashboard")
      .check(status.is(200))
  )
```

## 5.3 Parameterization & Correlation

**Correlation** is the process of extracting dynamic data from one request's response and using it in subsequent requests (e.g., tokens, IDs).

```
.exec(
  http("Step 1 - Get Token")
    .get("/auth")
    .check(jsonPath("$.token").saveAs("authToken"))
)
.exec(
  http("Step 2 - Use Token")
    .get("/profile")
    .header("Authorization", "Bearer ${authToken}")
    .check(status.is(200))
)
```

## 5.4 Looping and Conditional Logic

You can add loops and conditional logic:

```
// Loop example
.repeat(5) {
  exec(http("Looped request")
    .get("/loop-endpoint"))
}

// Conditional example
.doIf(session => session("condition").as[Boolean]) {
  exec(http("Conditional Request")
  .get("/conditional-endpoint"))
}
```

## 5.5 Complex Injection Profiles

Combine multiple injection steps:

```
setUp(
 scn.inject(
  nothingFor(4.seconds),
  rampUsers(10).during(5.seconds),
  constantUsersPerSec(20).during(15.seconds),
  rampUsersPerSec(20).to(50).during(10.seconds)
 )
).protocols(httpProtocol)
```

This sequence means:

1. Wait 4 seconds (no traffic).
2. Ramp up to 10 users over 5 seconds.
3. Maintain 20 users/second load for 15 seconds.
4. Ramp from 20 to 50 users/second over 10 seconds.

## 5.6 Assertions in Detail

To ensure your performance criteria are met, assertions are crucial:

```
.assertions(
 global.responseTime.max.lt(2000),
 global.successfulRequests.percent.gt(99),
 details("requestName").failedRequests.count.is(0)
)
```

Common assertion scopes:

- **global**: Across all requests in the simulation.
- **forAll**: For each request name.
- **details("requestName")**: For a specific request.

## 5.7 Distributed Testing

To handle extremely large load tests, you might run Gatling on multiple **load generators**:

1. **Master** node orchestrates the test.
2. **Slave** nodes receive instructions and generate load.
3. Gather results on the master node and aggregate them.

This can be done using continuous integration tools (e.g., Jenkins) or container orchestration (e.g., Kubernetes).

---

## 6. Integration and Automation

### 6.1 Continuous Integration (CI)

- **Maven**: Use the Gatling Maven plugin to run tests via mvn gatling:test.
- **Gradle**: Similarly, use the Gatling Gradle plugin to run tests via gradle gatlingRun.
- **Jenkins, GitLab CI, GitHub Actions**: Integrate commands in your CI pipelines to automatically execute simulations and publish HTML reports as artifacts.

### 6.2 Docker

- **Official Docker images**: Gatling provides official Docker images for easy setup.

```
docker run -it -v $(pwd):/opt/gatling/project -v $(pwd)/results:/opt/gatling/results \
  gatling/gatling:latest \
  -s simulations.BasicSimulation
```

- Mount your simulation files in /opt/gatling/project and results in /opt/gatling/results.

---

## 7. Best Practices and Tips

1. **Start with smaller load**: Ramp up gradually to detect performance bottlenecks early.
2. **Parameterize requests**: Use feeders to avoid caching effects and realistic data.
3. **Correlate dynamic values**: Ensure tokens or session IDs are captured and reused properly.
4. **Utilize checks**: Validate responses so you don't load test error pages.
5. **Assertions**: Clearly define pass/fail criteria for SLAs (Service Level Agreements).
6. **Scalability**: Use distributed testing for large-scale tests.
7. **Version control your scripts**: Keep your Gatling scripts alongside your application code.
8. **Automate**: Run Gatling tests in CI/CD for continuous performance feedback.

---

## 8. Sample End-to-End Simulation

Below is a more complete example that ties everything together:

```
package simulations

import io.gatling.core.Predef._
import io.gatling.http.Predef._
import scala.concurrent.duration._

class EndToEndSimulation extends Simulation {

 // 1. HTTP Protocol
 val httpProtocol = http
  .baseUrl("https://example.com")
  .acceptHeader("application/json")
  .contentTypeHeader("application/json")
  .userAgentHeader("Gatling/EndToEnd")

 // 2. Feeder
 val userFeeder = csv("data/users.csv").circular

 // 3. Scenarios
 // Login scenario
 val loginScenario = scenario("User Login")
  .feed(userFeeder)
  .exec(
   http("Login Request")
    .post("/login")
    .body(StringBody("""{ "user": "${username}", "pass": "${password}" }"""))
    .check(status.is(200))
    .check(jsonPath("$.token").saveAs("authToken"))
  )

 // Profile scenario
 val profileScenario = scenario("User Profile")
  .exec(
   http("Fetch Profile")
    .get("/profile")
    .header("Authorization", "Bearer ${authToken}")
    .check(status.is(200))
  )

 // 4. Setup
 setUp(
  loginScenario.inject(rampUsers(50).during(10.seconds)),
  profileScenario.inject(rampUsers(50).during(10.seconds))
 ).protocols(httpProtocol)
  .assertions(
   global.responseTime.mean.lt(1000),
   forAll.successfulRequests.percent.gt(95)
  )
}
```

### Explanation

- **Login Request** uses a CSV feeder to pick up usernames and passwords, capturing an authentication token.
- **Profile Request** reuses the authToken from the session.

---

## 9. Advanced Protocol Support

Although Gatling is most commonly used for **HTTP** testing, it supports other protocols either out-of-the-box or through extensions:

1. **WebSockets**
   - Gatling provides a built-in WebSocket API that allows you to test real-time, two-way communication.
   - You can open a WebSocket, send messages, and check server responses.
   - Example:

```
exec(
  ws("Open WebSocket")
    .connect("/ws/endpoint")
)
.pause(1.second)
.exec(
  ws("Send Message")
    .sendText("Hello, WebSocket!")
    .check(wsAwait.within(10.seconds).until(1).regex(".*ack.*"))
)
.exec(ws("Close WebSocket").close)
```

2. **Server-Sent Events (SSE)**
   - **SSE** is a unidirectional protocol where the server continuously sends data to the client over HTTP.
   - In Gatling, you can open an SSE stream and **wait** for specific events or patterns:

```
exec(
  sse("Open SSE").connect("/sse/stream")
)
.exec(
  sse("Wait for Event")
    .setCheck(
      sseCheck
        .regex("data: (.*)")
        .saveAs("eventData")
    )
  // You could wait for a certain number of messages or a timeout
)
.exec(sse("Close SSE").close)
```

3. **JMS**
   - Gatling has a **JMS** module for testing message-oriented middleware (e.g., ActiveMQ, RabbitMQ via JMS support).
   - You need to add the Gatling JMS dependency to your project and configure the JMS protocol with the connection factory:

```
import io.gatling.jms.Predef._

val jmsConfig = jms
  .connectionFactoryName("ConnectionFactory")
  .url("tcp://localhost:61616")
  .usePersistentDeliveryMode

val scn = scenario("JMS Test")
  .exec(
    jms("request")
      .sendQueue("testQueue")
      .textMessage("Hello JMS")
  )
  .exec(
    jms("response")
      .receiveQueue("testQueue")
      .check(simpleCheck(message => message.getText == "Hello JMS Reply"))
  )

setUp(scn.inject(atOnceUsers(10))).protocols(jmsConfig)
```

---

# 10. Advanced Load Modeling Techniques

## 10.1 Composite Injection Profiles

Gatling's injection DSL allows you to **layer** injection steps for more realistic load patterns. For example, simulating:

- **Warm-up** period with a moderate ramp,
- **Peak** traffic for a sustained period,
- **Soak** or **endurance** testing at a constant rate,
- **Spike** testing to see how the system responds to sudden surges.

```
setUp(
  scn.inject(
    // Warm-up
    rampUsers(50).during(30.seconds),
    // Peak
    constantUsersPerSec(20).during(2.minutes),
    // Spike
    rampUsersPerSec(20).to(100).during(10.seconds),
    // Soak
    constantUsersPerSec(100).during(5.minutes)
  )
).protocols(httpProtocol)
```

### 10.2 Throttling

Throttling lets you **control** the throughput (requests/second) rather than the number of concurrent users:

```
setUp(
 scn.inject(rampUsers(100).during(30.seconds))
).protocols(httpProtocol)
 .throttle(
  reachRps(20).in(10.seconds),
  holdFor(1.minute),
  jumpToRps(50),
  holdFor(2.minutes)
 )
```

This approach is especially handy for API testing where you want to ensure the target RPS (Requests per Second) stays within certain bounds.

---

## 11. Scalability and Distributed Testing

### 11.1 Distributed Mode (Open Source)

- Gatling itself (open-source) doesn't have a built-in cluster mode.
- However, you can **manually** orchestrate multiple Gatling instances (e.g., on AWS EC2 servers, Docker containers, Kubernetes pods) and combine the HTML results after the tests complete.
  - You'll need to manage test start times (for example, using Jenkins pipelines or custom scripts) so all your load generators begin simultaneously.
  - Each instance will produce its own report in the results/ directory. You can combine the raw logs using Gatling's Enterprise or community tooling.

### 11.2 Gatling Enterprise

- **Gatling Enterprise** is a commercial offering that provides:
  - Built-in distributed testing,

    - Real-time monitoring and reporting dashboards,
    - Seamless CI/CD integration,
    - Role-based access control, scheduling, and more.
- This is recommended if you require heavy load tests (thousands to millions of concurrent users), advanced analysis, or enterprise-grade support.

---

# 12. Debugging and Troubleshooting

## 12.1 Logging and Verbose Mode

- By default, Gatling logs are found in gatling-charts-highcharts-bundle-<version>/logs.
- Increase the log level in conf/logback.xml or conf/gatling.conf for more verbose output when troubleshooting:

```
<logger name="io.gatling" level="DEBUG"/>
```

- You can also use .extraInfoExtractor to capture additional info about requests and responses for debugging.

## 12.2 Print Session Data

Insert a simple session function to print variables:

```
.exec { session =>
  println("Current Session Data: " + session)
  session
}
```

This helps confirm that feeder data or extracted tokens are set correctly.

## 12.3 Pauses and Think Times

If your application uses rate-limiting or session-based logic, **pauses** can help mimic realistic user think time:

```
.pause(1.second, 3.seconds) // random pause between 1-3 seconds
```

If you see unexpected errors or timeouts, reduce concurrency or add pauses to isolate issues.

---

# 13. Monitoring and Observability

## 13.1 Application Monitoring

During a load test, you must monitor:

- **CPU, RAM, and GC** (on the application servers),
- **Database** (connections, locks, queries/second),
- **Network** (bandwidth, latency),
- **Logs** (errors, stack traces).

Tools like **Prometheus + Grafana**, **New Relic**, **Datadog**, or **Splunk** can help correlate Gatling load with application/server metrics.

## 13.2 Gatling Reports

After a test, Gatling automatically generates an **HTML report** in the `results/` folder:

- **Response Time Distribution** and **Percentiles** (50th, 75th, 95th, 99th),
- **Requests/second** (throughput),
- **Failures** (with reason codes),
- **Response Time Over Time** graph.

Review these reports to identify:

- **High-latency endpoints**,
- **Response time spikes**,
- **Errors** or timeouts.

---

# 14. CI/CD Integration in Practice

## 14.1 Jenkins Example

1. **Install Gatling** on your Jenkins server or use a Docker image with Gatling included.
2. **Store your simulations** in the same repository as your application code.
3. Add a **Jenkins pipeline** step to execute:

```
gatling.sh -s simulations.MySimulation
```

4. Archive **HTML report** artifacts from the `results/` directory.

You can also use the Jenkins Gatling Plugin to publish **trend reports**.

### 14.2 GitLab CI

1. Add Gatling to your CI environment (Docker image or an install step).
2. Create a **.gitlab-ci.yml** job:

```
stages:
 - test

performance_test:
 stage: test
 script:
  - ./gatling.sh -s simulations.MySimulation
 artifacts:
  paths:
   - results/**/*
  expire_in: 1 day
```

3. **Artifacts** contain the Gatling report for download/viewing.

---

## 15. More Advanced Tips & Tricks

1. **Use a build tool** (Maven/Gradle) to manage dependencies and run simulations consistently:
   - E.g., `mvn gatling:test` or `./gradlew gatlingRun`.
2. **Leverage custom Scala code** for complex data manipulations, advanced logic, or dynamic feeders (e.g., generating random JSON on-the-fly).
3. **Assertion strategies**:
   - Use percentile-based response time assertions (`percentile4` or `percentile95`) to define realistic SLAs.
   - Check error rates in addition to success/failure.
4. **Resource files** (e.g., images, JS, CSS) might artificially inflate response times. Consider ignoring or mocking static resources to focus on critical endpoints.
5. **Parameterize configuration** (like base URLs, environment variables, user credentials) so you can easily switch between dev, staging, and production environments.

---

## 16. Putting It All Together: Advanced Example

Below is a **more complex** scenario that demonstrates several advanced features: WebSocket usage, dynamic feeders, correlation, and conditional logic.

```
package simulations

import io.gatling.core.Predef._
import io.gatling.http.Predef._
import io.gatling.http.request.builder.ws.Ws
import scala.concurrent.duration._
import scala.util.Random

class AdvancedSimulation extends Simulation {

 // 1. HTTP/HTTPS base protocol
 val httpProtocol = http
  .baseUrl(sys.env.getOrElse("BASE_URL", "https://example.com"))
  .acceptHeader("application/json")
  .contentTypeHeader("application/json")
  .userAgentHeader("Gatling/Advanced")

 // 2. Dynamic Feeder (generating random data inline)
 val dynamicFeeder = Iterator.continually(
  Map(
   "username" -> s"user_${Random.alphanumeric.take(5).mkString}",
   "password" -> "defaultPass123"
  )
 )

 // 3. Correlation: Extract token and reuse in both HTTP and WebSocket
 val loginChain = exec(
  http("Login Request")
   .post("/api/login")
   .body(StringBody("""{ "user": "${username}", "pass": "${password}" }"""))
   .check(status.is(200))
   .check(jsonPath("$.token").saveAs("authToken"))
 )

 val fetchProfileChain = exec(
  http("Profile Request")
   .get("/api/profile")
   .header("Authorization", "Bearer ${authToken}")
   .check(status.is(200))
   .check(jsonPath("$.userId").saveAs("userId"))
 )

 // 4. WebSocket Scenario: connect, send message, await response
 val wsScenario = scenario("WebSocket Messaging")
  .exec(
   ws("Open WebSocket")
```

```
      .connect("/ws/chat?token=${authToken}")
  )
  .pause(1.second)
  .exec(
    ws("Send Chat Message")
      .sendText("""{"type": "chat", "message": "Hello World!"}""")
      .check(wsAwait.within(5.seconds).until(1).regex(".*Hello World!.*"))
  )
  .exec(ws("Close WebSocket").close)

 // 5. Conditional Logic: if userId is certain type, fetch extra data
 val conditionalChain = doIfOrElse(session =>
session("userId").asOption[String].exists(_.startsWith("admin"))) {
  exec(
    http("Admin Data Request")
      .get("/api/admin/dashboard")
      .header("Authorization", "Bearer ${authToken}")
      .check(status.is(200))
  )
 }{
  exec(
    http("Regular User Data Request")
      .get("/api/user/home")
      .header("Authorization", "Bearer ${authToken}")
      .check(status.is(200))
  )
 }

 // 6. Combined Scenario
 val fullScenario = scenario("Advanced User Flow")
  .feed(dynamicFeeder)
  .exec(loginChain)
  .exec(fetchProfileChain)
  .exec(conditionalChain)

 // 7. Setup: Composite Injection with Throttling
 setUp(
  fullScenario.inject(
    nothingFor(2.seconds),
    rampUsers(10).during(10.seconds),
    constantUsersPerSec(10).during(20.seconds)
  ),
  wsScenario.inject(
    rampUsers(5).during(5.seconds),
    constantUsersPerSec(2).during(10.seconds)
  )
 ).protocols(httpProtocol)
  .throttle(
   reachRps(20).in(10.seconds),
   holdFor(30.seconds)
  )
  .assertions(
   global.responseTime.percentile4.lt(1500),
    global.successfulRequests.percent.gt(95)
  )
}
```

## Explanation

- **Random Feeder**: Generates random usernames to ensure each user session is unique.
- **Conditional Logic**: Checks if userId starts with "admin"; if so, it fetches an admin dashboard, otherwise fetches a regular user home page.
- **WebSocket** Scenario**: Opens a WebSocket with the authToken, sends a message, checks for a response, then closes.
- **Composite Injection**: Mixes ramp-ups and constant rates for the HTTP scenario and the WebSocket scenario.
- **Throttling**: Ensures we never exceed 20 RPS globally.
- **Assertions**: 4th percentile response time under 1500ms, and overall success rate above 95%.

---

## Final Thoughts

Gatling stands out for its **expressive DSL**, **high performance**, and **detailed reporting**. By iteratively building more complex scenarios, integrating with your CI/CD pipeline, and monitoring system health, you'll gain a deep understanding of your application's performance profile.

Whether you're **just starting** with simple tests or **moving into advanced** territory with WebSockets, SSE, and distributed load testing, Gatling's flexibility and scalability can handle a wide range of use cases.

**Key takeaways:**

1. **Model Realistic User Journeys**: Break down your application into typical user flows.
2. **Use Feeders and Correlation**: Drive requests with dynamic data and capture server outputs for subsequent requests.
3. **Continuously Validate**: Checks and assertions ensure you're testing for success, not just sending traffic.
4. **Automate**: Run Gatling tests in your CI/CD pipeline to catch regressions early.
5. **Scale**: For very high loads, distribute tests manually or use Gatling Enterprise.
6. **Monitor & Optimize**: Always pair load tests with server/application monitoring to pinpoint bottlenecks.