Автор Имя и логин в Слаке	s.g.danilov
Иллюстрации для статьи Ссылка на место, где хранятся оригиналы всех изображений из статьи	
ТЗ для дизайнера Предложите идею для обложки	
Обложка Ссылка на Дропбокс	**https://disk.yandex.ru/i/qixQi4GAhL6KJw**
Шер для соцсетей Предложите идею, если есть	Тезисы для анонса

Использование Gatling. Тестирование AMQP

Всем привет! С вами Сергей из команды тестирования производительности. Мы завершаем цикл статей о Gatling. Ранее мы уже рассмотрели работу с HTTP, JDBC, gRPC и Kafka, напоследок расскажу про AMQP.

Немного определений

Википедия дает такое определение протокола: Advanced Message Queuing Protocol — открытый протокол для передачи сообщений между компонентами системы. Основная идея в том, что отдельные подсистемы или независимые приложения могут обмениваться произвольным образом сообщениями через AMQP-брокер, который маршрутизирует, обеспечивает гарантированную доставку и распределяет потоки данных или подписку на нужные типы сообщений.

AMQP основан на трех понятиях:

1. Сообщение, **message.** Единица передаваемых данных, содержание которой никак не интерпретируется сервером. К сообщению могут быть присоединены структурированные заголовки.
2. Точка обмена, **exchange.** В нее отправляются сообщения. Точка обмена распределяет сообщения в одну или несколько очередей. При этом в точке обмена сообщения не хранятся.
3. Очередь, **queue.** Здесь хранятся сообщения до тех пор, пока их не заберет клиент. Он всегда забирает сообщения из одной или нескольких очередей.

Producer — клиентское приложение, которое публикует сообщения в **exchange.** **Consumer** — клиентское приложение, которое получает сообщения из очереди сообщений.

Тестовый сервис RabbitMQ

Для разработки скрипта Gatling нам необходим сервер с AMQP-брокером. В этой статье используем Rabbitmq как наиболее распространенный. Чтобы развернуть тестовый сервис, необходим установленный docker. С помощью docker-compose поднимем сервис:

```
version: "3.9"
services:
  rabbitmq:
    image: rabbitmq:3.9.7-management-alpine
    container_name: rabbitmq
    environment:
      - RABBITMQ_DEFAULT_USER=rabbitmq
      - RABBITMQ_DEFAULT_PASS=rabbitmq
    ports:
      - '5672:5672'
      - '15672:15672'
```

Запускаем:

```
docker-compose up
```

Теперь по адресу http://localhost:15672/ доступна консоль администратора. Логин и пароль для входа — *rabbitmq*.

Для скрипта публикации сообщений создаем очередь, в которую будем отправлять сообщения. Для этого нужно зайти в консоль, перейти на вкладку Queues, указать имя очереди *test_queue* и нажать AddQueue. Для скрипта публикации и чтения сообщений нужно по аналогии создать очередь *test_queue_out*.

Разработка скрипта публикации сообщений

Мы не будем разрабатывать проект с нуля, а используем уже готовый шаблон и с его помощью создадим проект myRabbitmq. Процесс создания мы описывали в первой статье цикла.

В результате получим готовый проект, но по умолчанию он для HTTP-протокола, поэтому его нужно переписать для AMQP.

Шаг 1. Обновим зависимости. В файле project/Dependencies.scala добавим (вместо <current version> подставить актуальную версию).

```
lazy val amqpPlugin: Seq[ModuleID]       = Seq(
  "ru.tinkoff" %% "gatling-amqp-plugin",
).map(_ % "<current version>" % Test)
```

В файле build.sbt добавим новую зависимость, чтобы после загрузки обновлений sbt можно было использовать AMQP-плагин.

```
libraryDependencies ++= amqpPlugin,
```

Загрузим новые зависимости в проект, запустив команду в консоли:

```
sbt update
```

Шаг 2. Введем переменные сервиса. В файле src/test/resources/simulation.conf хранятся дефолтные переменные для запуска. Добавим в него переменные, которые определяют наш RabbitMQ.

```
amqpHost: "localhost"
amqpLogin: "rabbitmq"
amqpPassword: "rabbitmq"
amqpPort: 5672
```

Шаг 3. Создадим Action для публикации сообщений. В директории cases создадим новый файл для объекта AmqpActions. В нем создадим действие, которое описывает публикацию сообщения в нашу очередь *test_queue*. Содержимое сообщения не статично и может изменяться в ходе теста. Параметр *#{messageId}* позволяет использовать значения, получаемые из Feeder.

```
package ru.tinkoff.load.myRabbitmq.cases

import io.gatling.core.Predef._
import ru.tinkoff.gatling.amqp.Predef._
import ru.tinkoff.gatling.amqp.request.{PublishDslBuilder,
RequestReplyDslBuilder}

object AmqpActions {

  val publishMessageToQueue: PublishDslBuilder = amqp("Publish to
exchange").publish
    .queueExchange("test_queue") // имя очереди, в которую отправляем
сообщения
    .textMessage("Hello message - #{messageId}") // текст сообщения
    .messageId("#{messageId}") // настройки сообщения
    .priority(0)

}
```

Шаг 4. Используем Feeders. Больше узнать о Feeders можно [из вводной статьи про Gatling.](#) В нашем примере используем фидеры [из подключаемой библиотеки](#)

gatling-picatinny.

Возьмем базовый фидер Gatling, который читает данные из CSV-файла. Создадим в resources директорию pools и в ней файл messageIds.csv. Запишем в этот файл первой строкой заголовок messageId и далее в каждой строке значения от 1 до 10. В результате файл messageIds.csv должен выглядеть следующим образом, значения от 1 до 10:

```
messageId
1
2
...
10
```

В директории myRabbitmq создадим новую директорию feeders, а в ней object Feeders.

```
package ru.tinkoff.load.myRabbitmq.feeders

import io.gatling.core.Predef._
import io.gatling.core.feeder.BatchableFeederBuilder

object Feeders {

  val idFeeder: BatchableFeederBuilder[String] =
csv("pools/messageIds.csv").random

  }
```

Шаг 5. Напишем сценарий теста. В CommonScenario описываем класс CommonScenario, в котором создаем сценарий — порядок выполнения определенных действий.

```
package ru.tinkoff.load.myRabbitmq.scenarios

import io.gatling.core.Predef._
import io.gatling.core.structure.ScenarioBuilder
import ru.tinkoff.gatling.amqp.Predef._
import ru.tinkoff.load.myRabbitmq.cases.AmqpActions
import ru.tinkoff.load.myRabbitmq.feeders.Feeders.idFeeder

object CommonScenario {
  def apply(): ScenarioBuilder = new CommonScenario().scn
}

class CommonScenario {

  val scn: ScenarioBuilder = scenario("Common Scenario")
    .feed(idFeeder) // вызов Feeder
```

```
    .exec(AmqpActions.publishMessageToQueue) // выполнение публикации
сообщения в очередь

}
```

Шаг 6. Опишем AMQP-протокол. В файле myRabbitmq.scala опишем протокол:

```
package ru.tinkoff.load

import io.gatling.core.Predef._
import ru.tinkoff.gatling.amqp.Predef._
import ru.tinkoff.gatling.amqp.protocol.AmqpProtocolBuilder
import ru.tinkoff.gatling.config.SimulationConfig.{getIntParam,
getStringParam}

package object myRabbitmq {

  val amqpHost: String = getStringParam("amqpHost")
  val amqpPort: Int = getIntParam("amqpPort")
  val amqpLogin: String = getStringParam("amqpLogin")
  val amqpPassword: String = getStringParam("amqpPassword")

  val amqpConf: AmqpProtocolBuilder = amqp
    .connectionFactory(
      rabbitmq
        .host(amqpHost)
        .port(amqpPort)
        .username(amqpLogin)
        .password(amqpPassword)
        .vhost("/"),
    )
    .replyTimeout(60000)
    .consumerThreadsCount(8)
    .usePersistentDeliveryMode

}
```

Шаг 7. Добавим нагрузочные тесты. В файле Debug.scala опишем тест для дебага, который запустит одну итерацию сценария.

```
package ru.tinkoff.load.myRabbitmq

import io.gatling.core.Predef._
import ru.tinkoff.gatling.amqp.Predef._
import ru.tinkoff.gatling.config.SimulationConfig._
import ru.tinkoff.load.myRabbitmq.scenarios.CommonScenario
```

```
class Debug extends Simulation {

  setUp(
    CommonScenario() // запускаем наш сценарий
  .inject(atOnceUsers(1)), // запускать будет один пользователь - одну итерацию
  ).protocols(
    amqpConf, // работа будет проходить по протоколу, который описан в конфигурации amqpConf
  ).maxDuration(testDuration) // максимальное время теста равно testDuration,если тест не завершится за меньшее время, он будет остановлен автоматически

}
```

Шаг 8. Запустим Debug-тест. Для запуска теста возьмем GatlingRunner или способ, указанный в шаге 8. Запуск Debug-теста. После выполнения скрипта можно посмотреть консоль RabbitMQ — количество сообщений в очереди увеличилось на 1.

```
package ru.tinkoff.load.myRabbitmq

import io.gatling.app.Gatling
import io.gatling.core.config.GatlingPropertiesBuilder

object GatlingRunner {

  def main(args: Array[String]): Unit = {

    // this is where you specify the class you want to run
    val simulationClass = classOf[Debug].getName // указывает имя теста Debug, либо какой-то другой, например, MaxPerformance

    val props = new GatlingPropertiesBuilder
    props.simulationClass(simulationClass)

    Gatling.fromMap(props.build)
  }

}
```

Шаг 9. Настроим публикацию и чтение сообщений. На этом этапе дополнительно добавим чтение сообщений из очереди. В файл AmqpActions запишем новое действие:

```
val publishAndReply: RequestReplyDslBuilder = amqp("Request Reply exchange test").requestReply
    .queueExchange("test_queue_out")
    .replyExchange("test_queue_out")
```

```
    .textMessage("""{"msg": "Hello message - #{messageId}"}""")
    .messageId("#{messageId}")
    .priority(0)
    .contentType("application/json")
    .headers("test" -> "performance", "extra-test" -> "34-#{messageId}")
    .check(
      bodyString.exists,
    )
```

Запишем вызов действия, которое публикует и читает сообщения, в сценарий CommonScenario.

```
val scn: ScenarioBuilder = scenario("Common Scenario")
    .feed(idFeeder)                           // вызов Feeder
    .exec(AmqpActions.publishMessageToQueue) // выполнение публикации
сообщения в очередь
    .exec(AmqpActions.publishAndReply)       // публикация в Exchange и
чтение из очереди
```

Заключение

Это заключительная статья в серии про Gatling. Весь цикл мы рассказывали шаг за шагом, как создать базовые скрипты на Gatling. Надеемся, получилось показать, что на самом деле Gatling — это не так сложно, как кажется на первый взгляд. Если интересно дальше изучать Gatling, рекомендуем Gatling Academy.

Спасибо всем, кто читает наши статьи!

Тезисы для анонса

Заключительная статья цикла про Gatling от команды тестирования Тинькофф. Сергей Данилов рассказывает как разработать скрипт Gatling на сервере с AMQP-брокером.

9 шагов с примерами кода, чтобы создать собственный скрипт.

Весь цикл наша команда тестирования рассказывала как создавать базовые скрипты на Gatling. Старались показать, что на самом деле Gatling — это не так сложно, как кажется на первый взгляд. Спасибо всем, кто читал!