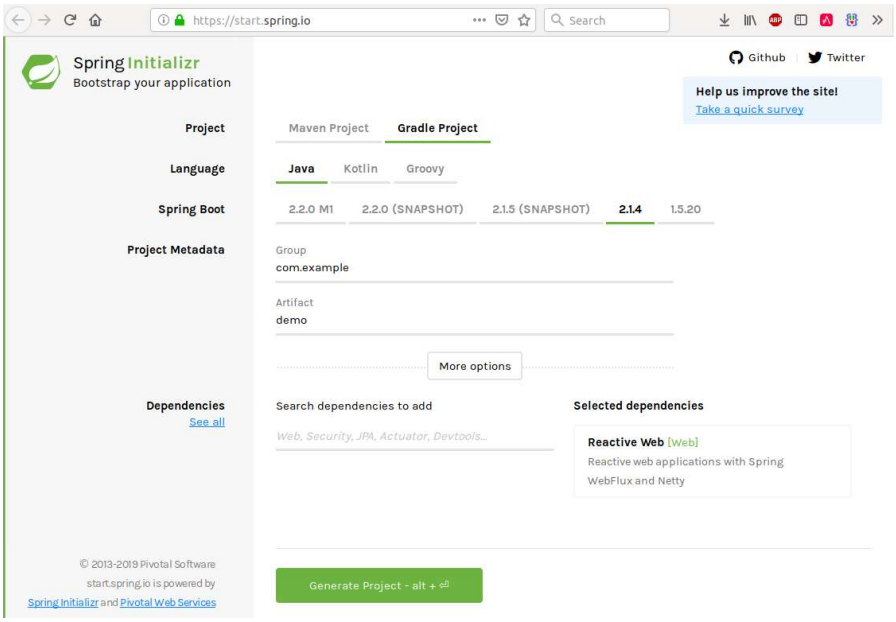
Spring Boot 2.0 Reactive Web Performance Metrics

Carvia Tech | April 07, 2019 | 3 min read | 750 views | spring-boot

Spring Boot 2.1 with Reactive Webflux support brings major performance improvements for non-blocking tasks i.e. network calls, mongodb repository calls, etc. In this article we will see how good the numbers are.

Step 1. Creating a new Spring Boot 2.1 Reactive Web Project

We will use https://start.spring.io/ to create a new gradle based spring boot 2.1.4 project with Reactive Web dependencies configured.



Using spring initializer to create project

Once the project is generated, you can expand the project to your desried workspace and import it into IntelliJ IDEA or Eclipse.

The `build.gradle` file will contain the below dependencies:

build.gradle

```
dependencies {
    compile('org.springframework.boot:spring-boot-starter-webflux')
    testCompile('org.springframework.boot:spring-boot-starter-test')
    testCompile('io.projectreactor:reactor-test')
}
```

Step 2. Create a component that will handle requests

Lets create a simple component with a method that serve ping requests:

/src/main/java/com/foo/HelloController.java

Similar Posts

Sendgrid Dynamic Templates with Spring Boot

Redis based rate limiter in Spring Boot

Custom TTL for Spring data Redis Cache

Spring Data ElasticSearch using HTTPS and Basic Auth

Generating UPI Payment QR Code in Java and Spring

Feign RequestInterceptor in Spring Boot

Send Gupshup SMS using Java API

RestTemplate with Basic Authentication Spring Boot

Disable SSL certificate validation in RestTemplate

How does Session handling works in Servlet environment

Book you may be interested in..

ebook PDF - Cracking Java Interviews v3.5 by Munish Chandel

Buy for Rs. 250

Book you may be interested in..

ebook PDF - Cracking Spring Microservices Interviews for Java Developers

Buy for Rs. 199

```
@Component
public class HelloController {

    public Mono<ServerResponse> ping(ServerRequest serverRequest) {
        return ServerResponse.ok()
                .contentType(MediaType.APPLICATION_JSON)
                .body(fromObject("{\n" +
                "  \"status\": \"ok\"\n" +
                "}"));
    }

}
```

Step 3. Create a Route Config

We will utilize reactive web router config to route calls for a given http pattern to the HelloController, as shown in the below self explanatary code.

/src/main/java/com/foo/RouteConfig.java

```
@Configuration
public class RouteConfig {

    @Bean
    public RouterFunction<?> routerFunction(HelloController testController) {
        return route(GET("/api/perf"), testController::ping);
    }
}
```

Step 4. Start the server

Now we are ready to start the service that we just created, we can use gradle to run the server using embedded Netty server.

Start the server

```
$ ./gradlew bootRun
```

This command will start the server on default server port i.e. `8080`

We can start server using alternative mechanism also, like creating an executable uber jar and then running it, or from the InteiiJ IDEA itself.

Step 5. Install wrk tool for performance testing

wrk - a HTTP benchmarking tool

wrk is a modern HTTP benchmarking tool capable of generating significant load when run on a single multi-core CPU. It combines a multithreaded design with scalable event notification systems such as epoll and kqueue.

More Information

https://github.com/wg/wrk

Installation on Ubuntu

To install wrk on ubuntu 18.04 system, just run the below command

```
sudo apt install wrk
```

Step 6. Benchmarking

Now start the Spring Boot application and run the below command from terminal

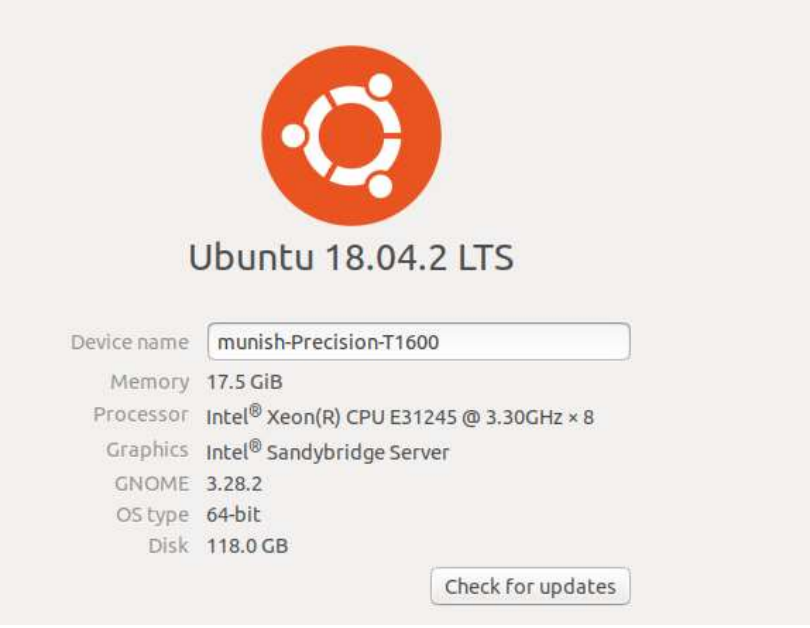
```
wrk -t8 -c40 -d30s http://localhost:8080/api/perf
```

This will launch 8 threads and keep 40 HTTP open connections for a duration of 30s and measure the results.


```
munish@munish-Precision-T1600:~$ wrk -t8 -c80 http://localhost:8080/api/perf
Running 10s test @ http://localhost:8080/api/perf
  8 threads and 80 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency     3.03ms    2.53ms  30.10ms   91.53%
    Req/Sec     3.79k   536.67     6.15k    79.88%
  302275 requests in 10.02s, 26.23MB read
Requests/sec:  30176.41
Transfer/sec:      2.62MB
```

Benchmark details

My machine is 16GB DDR3, Xeon E31245, 2011 Model Desktop



cpu-information

So we can see that a simple ping controller gives throughput of 31000 requests per second, which is not bad at all.

Step 7. Using h2load for HTTP Benchmarking

h2load to send 100000 requests using 32 concurrency level

```
$ h2load -n100000 -c32 -m10 --h1 http://localhost:8080/api/perf
```

Response

```
starting benchmark...
spawning thread #0: 32 total client(s). 100000 total requests
Application protocol: http/1.1

finished in 1.62s, 61670.27 req/s, 5.35MB/s
requests: 100000 total, 100000 started, 100000 done, 100000 succeeded, 0
failed, 0 errored, 0 timeout
status codes: 100000 2xx, 0 3xx, 0 4xx, 0 5xx
traffic: 8.68MB (9100000) total, 4.20MB (4400000) headers (space savings
0.00%), 1.91MB (2000000) data
                     min         max         mean         sd        +/- sd
time for request:      303us     38.16ms      4.68ms      3.23ms    83.42%
time for connect:      197us      3.18ms      1.60ms       864us    59.38%
time to 1st byte:     3.02ms     28.45ms     10.76ms      5.65ms    78.13%
req/s           :    1929.11     2451.89     2142.68      163.55    62.50%
```

We can see that with latest version of Spring Boot 2.1.4, we can get a throughput of `60K requests/second` on the same Ubuntu 18.04.2 server.

Download Code

You can download codebase repository from Github

https://github.com/cancerian0684/spring-boot-2-performance

Top articles in this category:

1. Testing web layer in Spring Boot using WebMvcTest
2. What is new in Spring Boot 2
3. AWS SDK 2.0 - S3 File upload & download in Java
4. SendGrid emails in Spring Boot
5. Sendgrid Dynamic Templates with Spring Boot
6. Basic Auth Security in Spring Boot 2
7. RestTemplate with Basic Authentication Spring Boot

Share article:

Facebook Twitter LinkedIn

Find more on this topic:

Spring Framework

Spring Framework - MVC, Dependency Injection, Spring Hibernate, Spring Data JPA, Spring Boot and Spring Cloud for Microservices Architecture.

Last updated 1 week ago