Pivotal

# Spring Boot 2.2 Performance Improvements

Dave Syer
@david_syer
September 2019

Spring Boot 2.2

# Runtime

# Useful Benchmarks are the ones you run

## Why running super-artificial benchmarks?

- Testing different parts of the web stack
- Comparing two versions (e.g., a specific patch)
- **Measuring Framework overhead**
- Optimizing for good latency, stable/predictable behavior, THEN throughput
- Learning!

**Is this a realistic use case?**

**What are we comparing here?**

**Show me latencies, percentiles, error rates!**

**Is this really your app's bottleneck?**

**Rank your favorite features (developer productivity, latency, throughput, predictability...)**

**Run your own**

# Full Stack Benchmarks and Load Testers

**Non-exhaustive selection...**

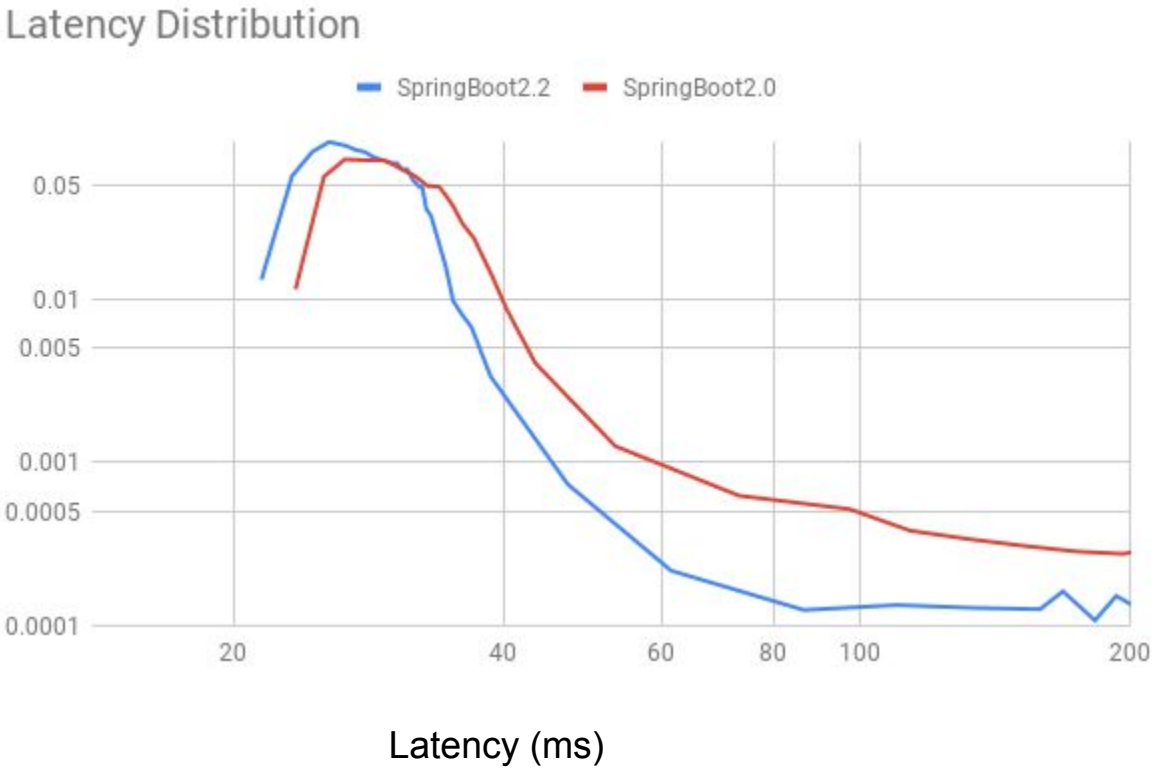
- Gatling: https://gatling.io
  - Scala DSL, good features for “realistic” test scenarios
  - Nice colourful visualizations
- Wrk2: https://github.com/giltene/wrk2
  - CLI. Nice histogram outputs in plain text.
  - Constant throughput HTTP load
- JMeter: https://jmeter.apache.org/
  - Very common in the wild
  - Many features, including non-web apps
- Apache Bench: https://httpd.apache.org/docs/2.4/programs/ab.html
  - CLI. Part of Apache HTTP. Installed from OS package manager.
  - Quick and dirty

# Wrk2

```
$ wrk -t2 -c100 -d30s -R2000 --latency http://gcp:8080
Running 30s test @ http://gcp:8080
 Thread calibration: mean lat.: 45.526ms, rate sampling interval: 174ms
...
  Latency Distribution (HdrHistogram - Recorded Latency)
 50.000%   29.44ms
 75.000%   33.92ms
 90.000%   43.42ms
 ...
 99.999%  654.34ms
100.000%  654.34ms

  Detailed Percentile spectrum:
...
```

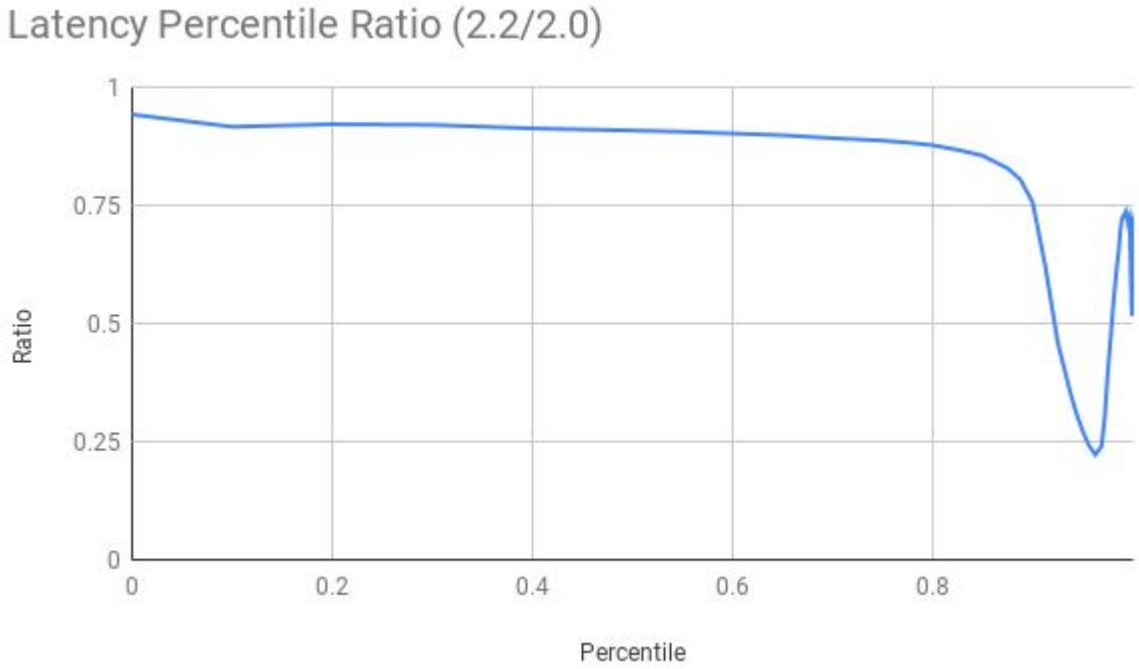
# Latency Distribution

Workload: Webflux JSON GET



# Latency Ratios

Workload: Webflux JSON GET



# Microbenchmarks

- JMH: https://openjdk.java.net/projects/code-tools/jmh/
  - Part of OpenJDK
  - Forking and threading
- MicrobenchmarkRunner: https://github.com/mp911de/microbenchmark-runner
  - JUnit integration with JMH
  - Run “test” in IDE and generate benchmark data

# MicrobenchmarkRunner

```
@Warmup(iterations = 1, time = 1)
@Fork(value = 1, warmups = 0)
@Microbenchmark
public class DispatcherBenchmark {

     @Benchmark
     public void test(MainState state) throws Exception {
          state.run("/foo");
     }

     @State(Scope.Thread)
     @AuxCounters(Type.EVENTS)
     public static class MainState {

...
}
```

# DispatcherHandler Benchmarks

Workload: Webflux JSON GET

| boot | class | method | count | errors | median | mean | range |
|---|---|---|---|---|---|---|---|
| 2.0 | WebFluxBenchmark | main | 111557.000 | ≈ 0 | 9702.130 | 10366.967 | 561.554 |
| 2.2 | WebFluxBenchmark | main | 140320.000 | ≈ 0 | 12871.144 | 13654.106 | 646.697 |

`DispatcherHandler` is 30% faster in Spring Boot 2.2

# Spring Framework Changes

## WebFlux and MVC:

https://github.com/spring-projects/spring-framework/issues/22644

https://github.com/spring-projects/spring-framework/issues/22340

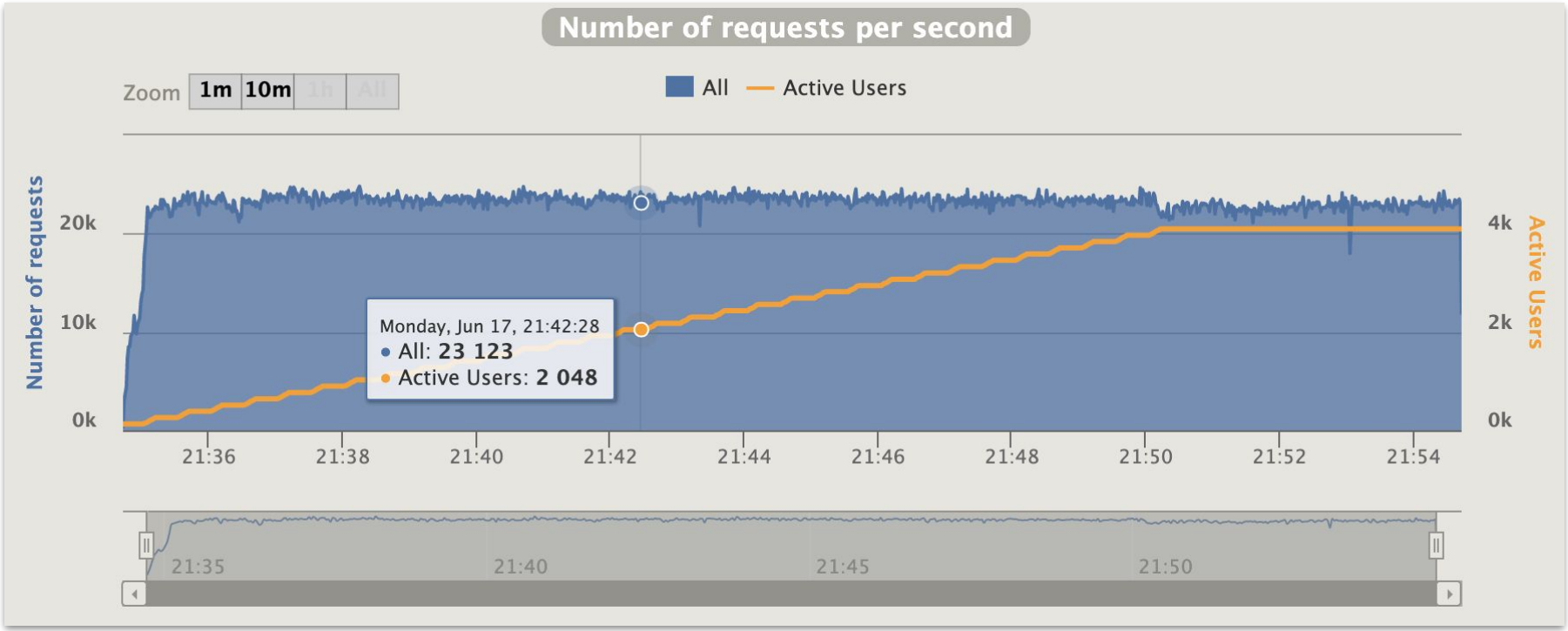
https://github.com/spring-projects/spring-framework/issues/22341

# Gatling

```
class JsonGet extends Simulation {

      val httpProtocol = http
       .baseUrl(baseUrl)

      val scn = scenario("Get JSON payload")
            .forever {
                  exec(http("jsonGet").get("/user/bchmark"))
            }


      setUp(
        scn.inject(
            incrementConcurrentUsers(increment)
              .times(steps)
              .eachLevelLasting(20 seconds)
              .separatedByRampsLasting(10 seconds)
              .startingFrom(0)
        )
      ).maxDuration(20 minutes).protocols(httpProtocol)

}
```

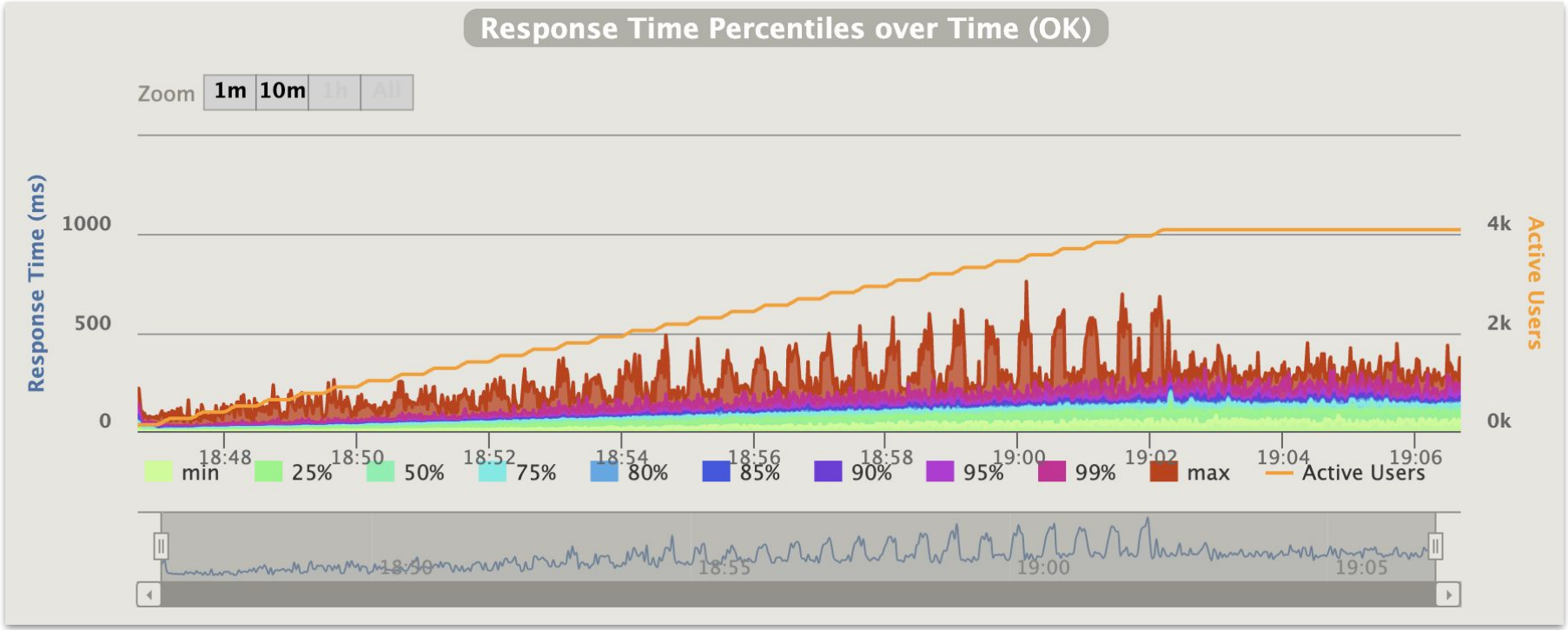
# Spring Web Performance Benchmark

Workload: MVC plain text echo



# Spring Web Performance Benchmark

Workload: reactor-netty plain text GET



# Spring Boot 2.1.0 vs. 2.2.0-SNAPSHOT

## Spring MVC:

- +20-30% req/sec depending on the use case
- Improved latency (75th percentile is now 95th percentile!)

## Spring WebFlux:

- +20-40% req/sec, more stable with concurrency
- Annotation model is now on par with Functional one
- Latency -10% overall, higher percentiles are much better
- A lot of it is building up on Reactor Netty's improvements

**This is a very subjective benchmark, you won't get the same results in your apps!**

Spring Boot 2.2

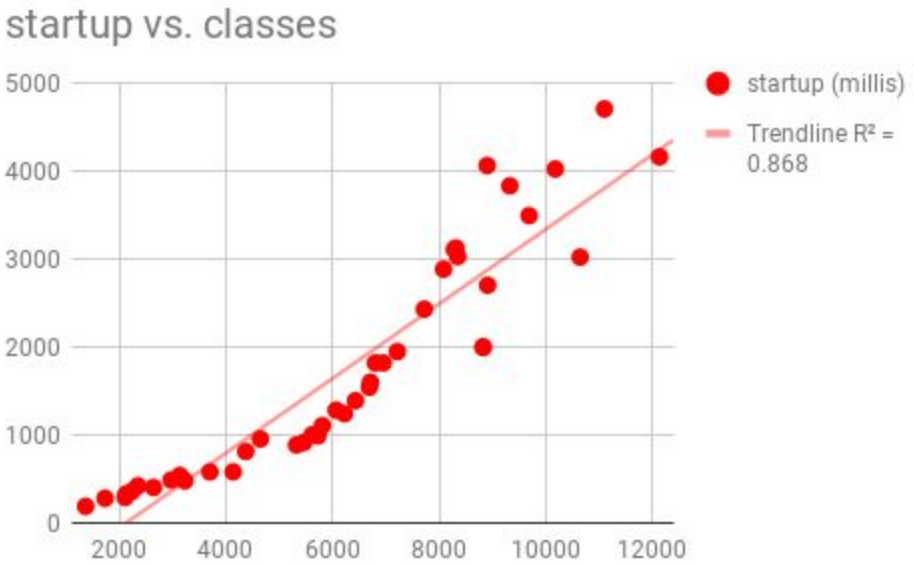
# Startup

# How Fast is Spring?

## TL;DR How can I make my app go faster?

- Exclude stuff from the classpath that you know you don't need
- Use `@Configuration(proxyBeanMethods=false)`
- Use the `spring-context-indexer`
- Use Spring 2.2 and Spring 5.2
- Use explicit `spring.config.location`
- Make bean definitions lazy by default (`spring.main.lazy-initialization=true` , `spring.data.jpa.repositories.bootstrap-mode=lazy` )
- Unpack the fat jar and run with an explicit classpath, use application main class
- Run the JVM with `-noverify`. Also consider `-XX:TieredStopAtLevel=1` . Java 11 or J9 use AppCDS.
- Import autoconfigurations individually
- Use functional bean definitions
- (Build a native image)

# More Classes Loaded -> More Time to Start



# Riff FunctionInvoker and Spring Cloud Function

Functional re-write https://github.com/projectriff/java-function-invoker

```
@SpringBootApplication
@EnableFunctionDeployer
public class JavaFunctionInvokerApplication
             implements ApplicationContextInitializer<GenericApplicationContext> {


      @Override
      public void initialize(GenericApplicationContext context) {
            context.registerBean(FunctionDeployerConfiguration.class,
                        () -> new FunctionDeployerConfiguration());
            context.registerBean(
                        "org.springframework.cloud.function.deployer.FunctionCreatorConfiguration",
                        ClassUtils.resolveClassName(
                                    "org.springframework.cloud.function.deployer.FunctionCreatorConfiguration",
                                    context.getClassLoader()));
...
}
```

# Riff FunctionInvoker and Spring Cloud Function

| container | cpus | startup(ms) |
|---|---|---|
| riff | 4 | 2817 |
| scf | 4 | 664 |
| riff | 2 | 4614 |
| scf | 2 | 653 |
| riff | 1 | **16782** |
| scf | 1 | 2121 |
| scf:n | 1 | **1091** |

15x faster with Spring Boot 2.2, Webflux, 1 cpu

# WebFlux and Micro Apps

Flux benchmarks from https://github.com/dsyer/spring-boot-startup-bench (startup time seconds)

```
class        method sample  beans    classes    heap     memory  median  mean   range

Netty Boot 2.2
MainBenchmark  main   demo  91.000   4392.000   8.121    49.800  0.760   0.770  0.009

Netty Boot 2.1
MainBenchmark  main   demo  101.000  4930.000  8.253    53.432  0.930   0.942  0.016

Netty Boot 2.0
MainBenchmark  main   demo  85.000   4806.000   9.519    55.198  0.964   0.980  0.018
```

# MVC and Micro Apps

MVC benchmarks from https://github.com/dsyer/spring-boot-startup-bench (startup time seconds)

| class | method | sample | beans | classes | heap | memory | median | mean | range |
|---|---|---|---|---|---|---|---|---|---|
| MVC Boot 2.2 | | | | | | | | | |
| MainBenchmark | main | demo | 114.000 | 4573.000 | 9.967 | 66.920 | 0.844 | 0.873 | 0.032 |
| MainBenchmark | main | **actr** | 208.000 | 5268.000 | **10.161** | **69.840** | **1.033** | 1.060 | 0.048 |
| MVC Boot 2.1 | | | | | | | | | |
| MainBenchmark | main | demo | 117.000 | 5185.000 | 11.135 | 71.880 | 0.988 | 1.016 | 0.037 |
| MainBenchmark | main | actr | 251.000 | 6008.000 | 12.975 | 78.007 | 1.328 | 1.342 | 0.019 |
| MVC Boot 2.0 | | | | | | | | | |
| MainBenchmark | main | demo | 109.000 | 5279.000 | 14.121 | 78.546 | 1.102 | 1.137 | 0.042 |
| MainBenchmark | main | **actr** | 242.000 | 6131.000 | **16.865** | **85.696** | **1.585** | 1.625 | 0.050 |

# WebFlux and Micro Apps

Flux benchmarks from https://github.com/dsyer/spring-boot-startup-bench

TODO: add some numbers from Spring Boot 2.0, and for MVC

```
class         method sample  beans   classes   heap    memory  median  mean  range
MainBenchmark  main  demo  93.000   4365.000  8.024  49.564  0.766  0.773  0.011
MainBenchmark  main  jlog  80.000   3598.000  6.141  43.006  0.667  0.679  0.019
MiniBenchmark  boot  jlog  28.000   3336.000  7.082  41.949  0.588  0.597  0.014
MiniBenchmark  mini  jlog  27.000   3059.000  5.487  38.953  0.534  0.545  0.018
MiniBenchmark  micro jlog  2.000    2176.000  4.608  32.886  0.336  0.345  0.013
```

# Manual Autoconfiguration

Blog: https://spring.io/blog/2019/01/21/manual-bean-definitions-in-spring-boot

```
@SpringBootConfiguration
@ImportAutoConfiguration({
    WebFluxAutoConfiguration.class, ReactiveWebServerFactoryAutoConfiguration.class,
    ErrorWebFluxAutoConfiguration.class, HttpHandlerAutoConfiguration.class,
    ConfigurationPropertiesAutoConfiguration.class, PropertyPlaceholderAutoConfiguration.class})
@RestController
public class DemoApplication {

  @GetMapping("/")
  public Mono<String> home() {
    return Mono.just("Hello World");
  }

}
```

# Spring Boot Features

Repo: https://github.com/dsyer/spring-boot-features

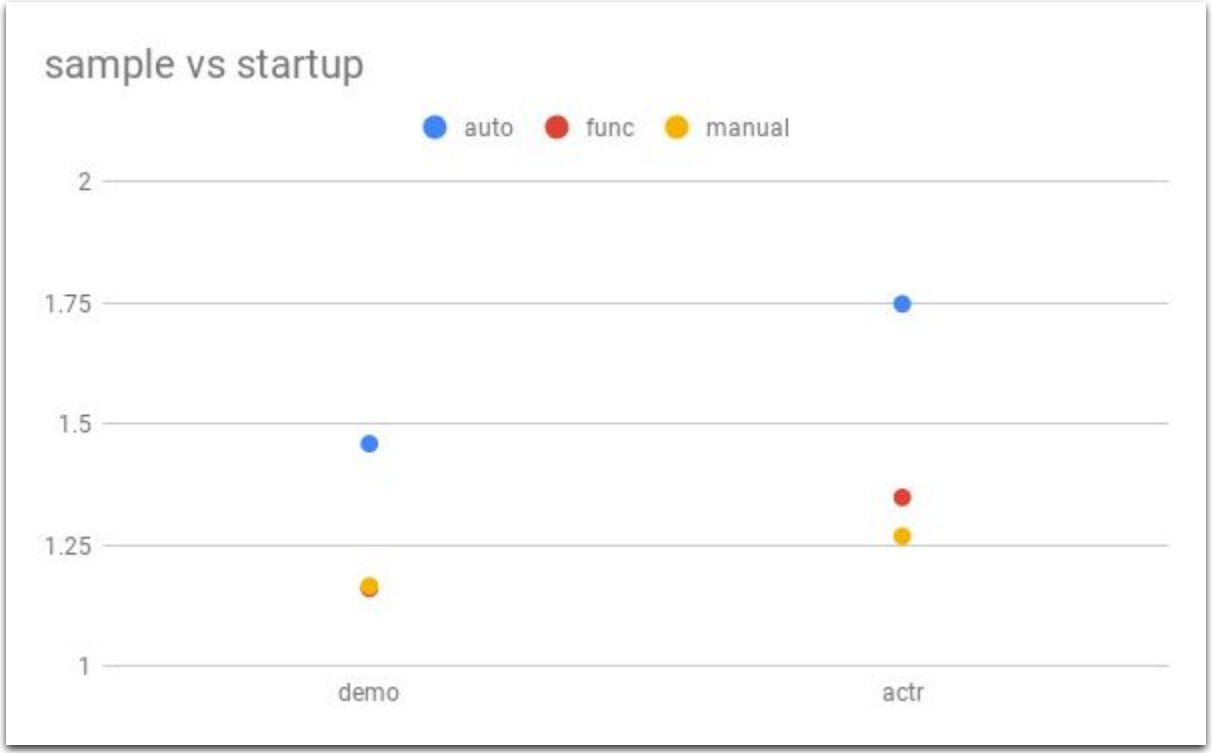
```
@SpringBootFeaturesApplication(WebFluxConfigurations.class)
@RestController
public class DemoApplication {

  @GetMapping("/")
  public Mono<String> home() {
    return Mono.just("Hello World");
  }

}
```

# Spring Fu

Repo: https://github.com/spring-projects-experimental/spring-fu

```
val app = application(WebApplicationType.REACTIVE) {
        beans {
                bean<SampleHandler>()
        }
        webFlux {
                port = if (profiles.contains("test")) 8181 else 8080
                router {
                        val handler = ref<SampleHandler>()
                        GET("/", handler::hello)
                        GET("/api", handler::json)
                }
                codecs {
                        string()
                        jackson()
                }
        }
}
```

# PetClinic Benchmarks




- Blue: out of the box Spring Boot.
- Yellow: manual autoconfiguration.
- Red: fully functional via Spring Init

# Spring Framework Changes

## Bean Factory:

https://github.com/spring-projects/spring-framework/issues/21457

# start.spring.io startup

Benchmarks using https://github.com/dsyer/spring-boot-startup-bench launcher

```
$ java -jar benchmarks.jar site.jar
```

2.1

```
Benchmark                Mode  Cnt  Score   Error  Units
MainBenchmark.jar        avgt   10  4.280 ± 0.061   s/op
MainBenchmark.launcher   avgt   10  4.420 ± 0.036   s/op
MainBenchmark.main       avgt   10  3.622 ± 0.035   s/op
```

2.2 with `-Dspring.main.lazy-initialization=true -Dspring.cache.type=none`

```
Benchmark                Mode  Cnt  Score   Error  Units
MainBenchmark.jar        avgt   10  2.251 ± 0.022   s/op
MainBenchmark.launcher   avgt   10  1.962 ± 0.048   s/op
MainBenchmark.main       avgt   10  1.767 ± 0.023   s/op
```

Spring Boot 2.2

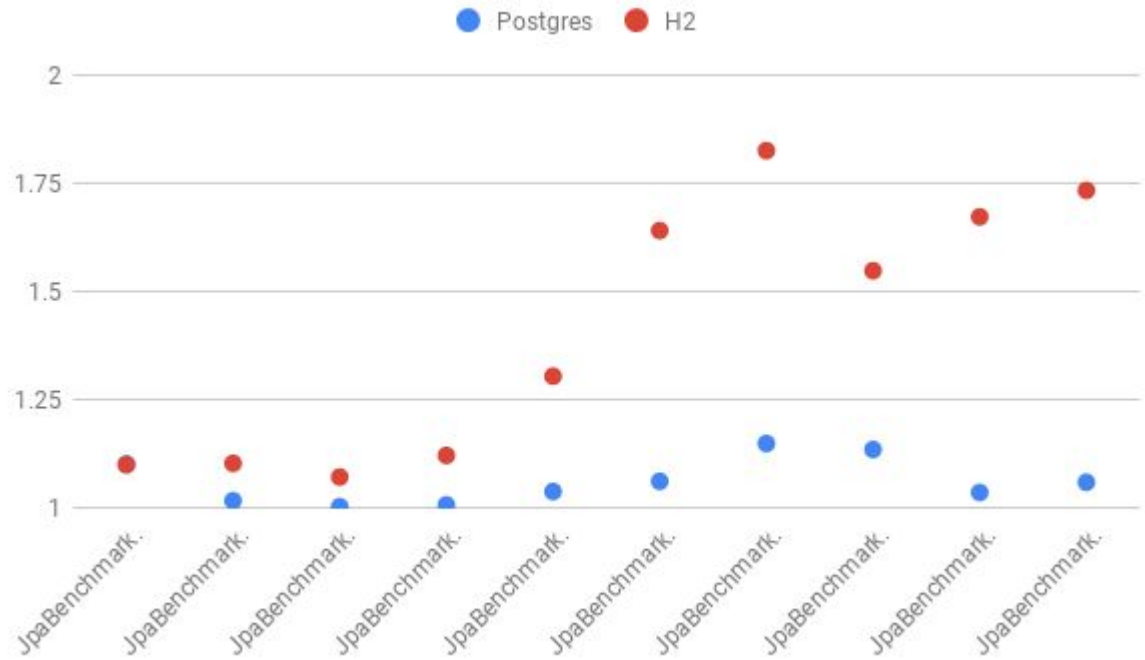
# Data Access

# Spring Data Repositories

- Bootstrap mode can be useful at dev time (`spring.data.jpa.repositories.bootstrap-mode=lazy` )
- Significant runtime performance improvements just before Spring Boot 2.2.0.M5
- Benchmarks at: https://github.com/spring-projects/spring-data-dev-tools/

# Spring Data JPA Benchmarks

Throughput Ratio After / Before Optimization



Spring Boot 2.2

# Test Containers

# Test Containers

- https://www.testcontainers.org
- Re-usable containers for development environment

```
@SpringBootTest({
   "spring.datasource.url=jdbc:tc:mysql:5.7.22:///petclinic",
   "spring.datasource.driver-class-name=org.testcontainers.jdbc.ContainerDatabaseDriver"
})
public abstract class MysqlIntegrationTest {
      ...
}
```

# Test Containers: Static Initializer

```
public static class Initializer
          implements ApplicationContextInitializer<ConfigurableApplicationContext> {

      private static MySQLContainer<?> mysql;
       static {
          mysql = new MySQLContainer<>().withUsername("petclinic").withPassword("petclinic")
                  .withDatabaseName("petclinic");
          mysql.start();
       }


       @Override
       public void initialize(ConfigurableApplicationContext context) {
          TestPropertyValues.of("spring.datasource.url=" + mysql.getJdbcUrl()).applyTo(context);
       }


   }
```

# Test Containers: Reusable Containers

Workload: PetClinic MySQL

| Spring Boot | Test Containers | App start (s) | JVM running (s) |
|---|---|---|---|
| 2.1.6 | 1.12.1 | 5.881 | 21.658 |
| 2.2.0 | reusable_containers | 2.055 | 4.812 |

```
static {
    mysql = new MySQLContainer<>().withUsername("petclinic").withPassword("petclinic")
            .withDatabaseName("petclinic").withReuse(true);
    mysql.start();
}
```

Spring Boot 2.2

# Graal VM

# Graal VM

- Oracle research project https://github.com/oracle/graal/
- OpenJDK + GC Engine, alternative to C1
- Community Edition (CE) and Enterprise Edition (EE)
- Polyglot programming model (Javascript, Ruby, R, Python, ...)
- Native images

# GraalVM GC: start.spring.io Project Generation

Benchmarks using JMH Microbenchmark Runner (projects per second)

| JVM | class | method | median | mean | range |
|---|---|---|---|---|---|
| Hotspot | ProjectGenerationIntegrationTests | projectBuilds | **57.674** | 65.802 | 5.223 |
| GraalCE | ProjectGenerationIntegrationTests | projectBuilds | **67.270** | 89.004 | 14.268 |
| GraalEE | ProjectGenerationIntegrationTests | projectBuilds | **74.396** | 92.541 | 13.535 |

Graal CE is faster but EE has an edge. The Graal measurements are more noisy.

# Native Images

- Spring 5.2 and Spring Boot 2.2 already have some features
- PetClinic:
  - Compile time: 10minutes(!)
  - Image size: 60MB
  - Startup time: 200ms
- More coming in 5.3/2.3. See presentation by Andy Clement and Sebastien Deleuze.

# All that Glitters

(Admittedly stupid) benchmark with wrk, hammering HTTP GET on localhost

```
$ wrk -t4 -c100 -d30 --latency http://localhost:8080
```

| | | Latency | |
|---|---|---|---|
| JVM | throughput(req/sec) | 50%(ms) | 99%(ms) |
| Hotspot | 116,000 | 0.661 | 16 |
| GraalCE | 63,927 | 1.140 | 24.43 |
| GraalEE | 119,116 | 0.520 | 24.71 |
| GraalCE (native) | 19,320 | 5.650 | 32.18 |
| GraalEE (native) | 22,139 | 4.060 | 22.68 |


**Transforming how the world builds software**