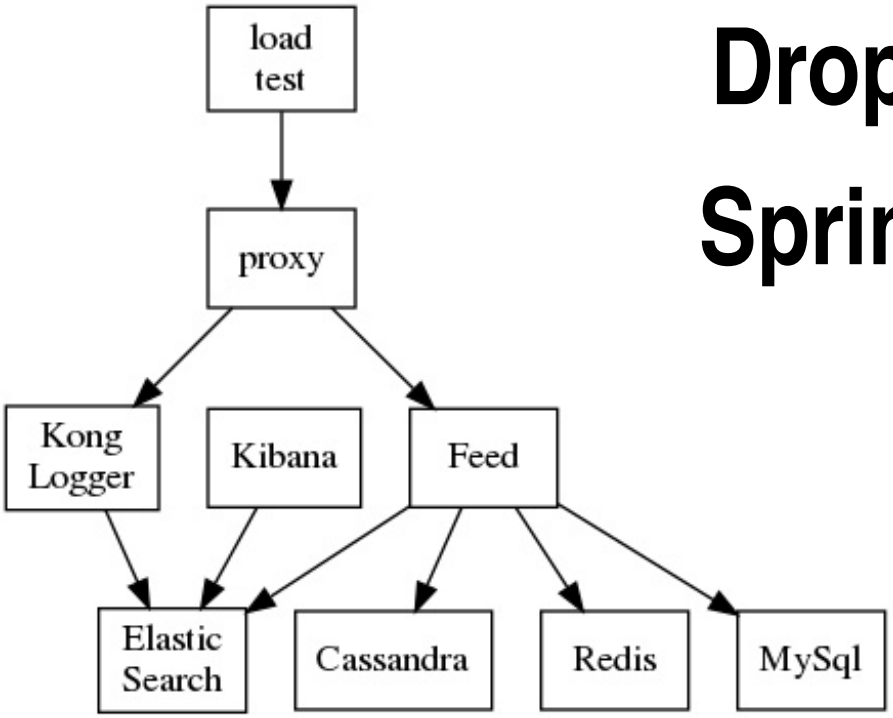


Dropwizard vs Spring Boot

by Glenn Engstrand

If you are planning on writing some microservices in Java, then you are most probably wondering whether to use Dropwizard or Spring Boot. If so, then this blog is for you. Although the title says Spring Boot, what I am really evaluating here is version 5 of the Spring Framework which includes Boot, Web, Core, Data, and Test. Whenever I evaluate a technology stack, I write a news feed microservice in that technology then compare that to feature identical microservices written in other technology stacks. Today, I will be comparing the news feed microservices written in Spring Boot vs Dropwizard.

There is a growing trend in Java to hide complexity with annotation based design patterns.

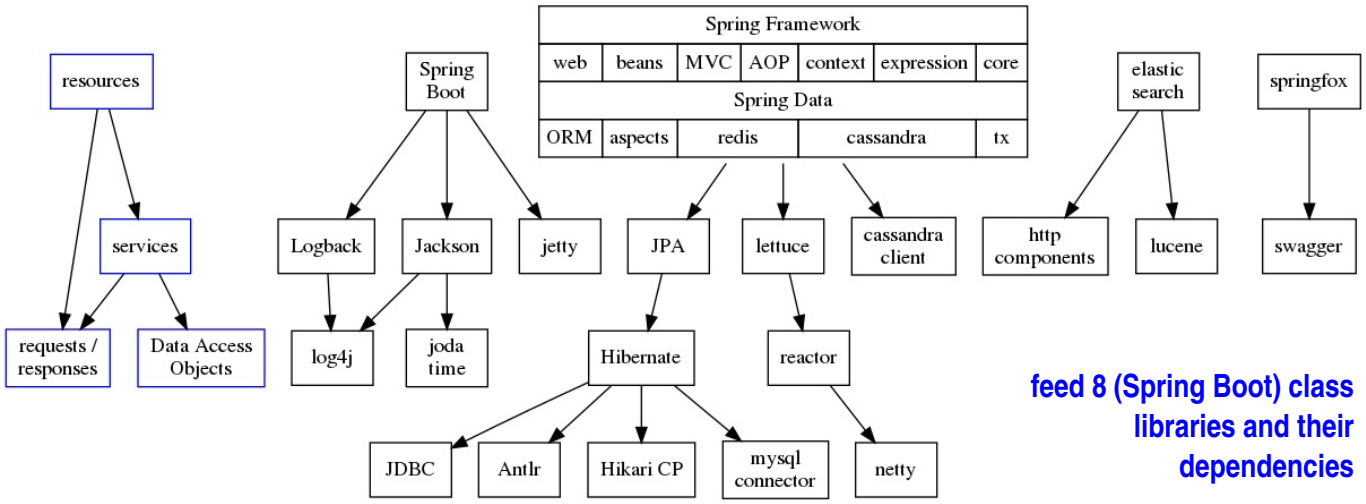
According to Google Trends, those two technologies were equal in popularity back when Spring Boot was first released in 2014. Since then, interest in Dropwizard has remained the same while Spring Boot interest has steadily increased. Perhaps this is because the Spring Framework is backed by Pivotal Software which (according to the S-1 that they filed prior to their April 2018 IPO) spent $221 million in sales and marketing in the previous year. The Dropwizard project is maintained by individual contributors with no marketing budget; however, they do get some sponsorship from Jetbrains. I remember attending various tech conferences over the years and seeing the occasional presentation for Spring Boot on the schedule but never for Dropwizard.

Since both technologies are for writing servlet container based microservices in Java, the architecture is very similar. Both microservices access MySql, Redis, Cassandra, and Elasticsearch since that is part of the functional requirements. I am a big believer in Model Driven Software Development and Swagger is a very popular technology for generating microservice boilerplate code with MDSD. I started using Swagger with the Dropwizard implementation and have continued to use it including the Spring Boot implementation.

It makes a lot of sense to develop microservices in such a way as to deploy and test them while they are running in a Kubernetes cluster. For the actual load tests, I do use the cloud managed solutions from either Amazon or Google. I use minikube when developing and testing each microservice on my laptop. As a minimum requirement for adoption, each candidate technology has to be able to function in those environments.

Architecture

An important part of evaluating any new microservice technology is to investigate how it performs under load and that means having the capability to measure its performance. I used to use Kafka to capture performance data but it is not easy to deploy Kafka on Kubernetes (although it is getting easier). Since then, I have switched over to measuring performance with an API gateway approach. The load test application makes its calls to the microservice through a full reverse proxy that also makes asynchronous calls to another service with the performance data for the call to the microservice under evaluation. That second service batches up the performance data then periodically updates Elasticsearch with that data in a format that is easy for Kibana to analyze and present.



feed 8 (Spring Boot) class libraries and their dependencies

Originally, I used Kong as the API gateway. It used to do a great job at that but started underperforming once version 1 was released. I can only speculate as to why. Kong has recently pivoted to becoming a service mesh for Kubernetes. Perhaps it has lost focus on its original purpose of being an API gateway?

I had to replace Kong with a service that is a drop in replacement to how Kong was being used here. I ended up coding a 100 line custom proxy in golang. The Go programming language feels like a better, more modern C and it has lots of great support for highly concurrent HTTP servers and clients.

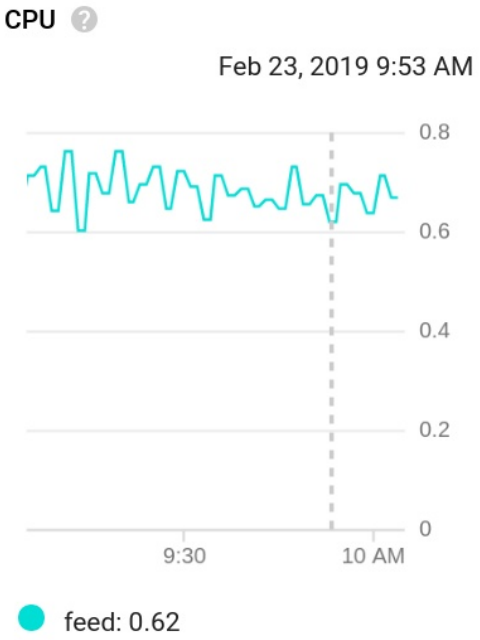
Design

Since both projects are started using similar Swagger templates and since both technology stacks are similar in architecture, it only stands to reason that there would be a lot of similarities in the design of both microservices. They are both organized into resources, services, and data access. Resource classes surface the API calls. Services are responsible for the business rules and for aggregating and caching data. Data Access Objects are used to access the underlying data bases. Both Dropwizard and Spring Boot use annotated interfaces where method intercepting proxies delegate functionality based on metadata.

Ever since Kong has pivoted from API gateway to service mesh, it has underperformed in terms of throughput.

There are plenty of differences too. While Dropwizard services can use Spring DI, this Dropwizard service uses Google Guice for Dependency Injection.

The Spring Boot service uses Spring DI which is included in Core. Spring is a little more opinionated than Guice in that it defaults to using the singleton pattern and supports the notion of component scans. Instead of a component scan, Guice depends on a module system where interfaces are explicitly bound to implementations during the configuration phase at time of system start. The Dropwizard framework is hard coded to use Jetty as the servlet container whereas Spring uses that component scan in order to figure out which servlet container to run under. It supports Tomcat, Jetty, and Undertow. When load testing the Spring Boot service, I used Jetty in order for it to be more comparable to the Dropwizard service. I also ran the load test with the Spring Boot service configured to use Tomcat and the difference in performance was negligible.

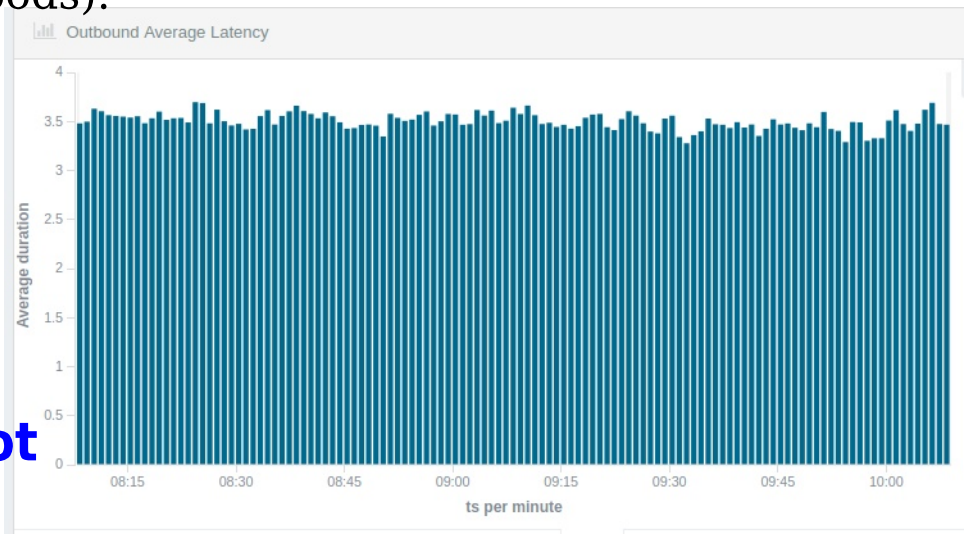


The most profound difference is in how the two technologies handle data access. Java applications tend to have a lot of plumbing code around data access, particularly with relational databases due to their stateful connections and due to the impedance mismatch between relational data and object orientation. Dropwizard addresses this with its own datasource factory and by incorporating another open source project called JDBI. Dropwizard db manages the pooling of JDBC connections. JDBI provides a relatively lightweight mapping between relational datasets and objects.

With non-relational databases, you are on your own and will most probably just use whatever libraries are available for Java.

Spring data takes a more heavyweight approach by providing repositories for both relational and non-relational databases. The repository for MySql calls JPA (Java Persistence API) using Hibernate as the implementation. I used the repositories for all of the underlying databases except for Elasticsearch because, at the time of this evaluation, the Spring Data repository for Elasticsearch depends on the native transport client instead of the high level REST client. The native transport client requires that you use a recent version of Elasticsearch which is too memory intensive to run on minikube (my laptop has only 6 GB RAM of which only 4 is available to run 9 pods).

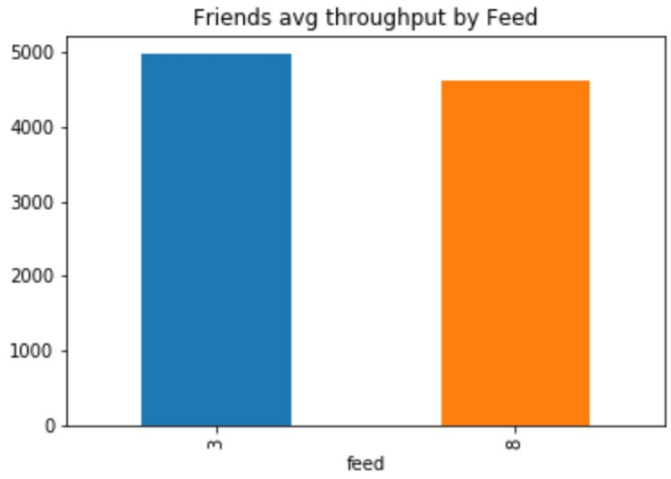
The Elasticsearch folks recommend the high level REST client anyway and will soon deprecate the native transport client.



Spring Boot

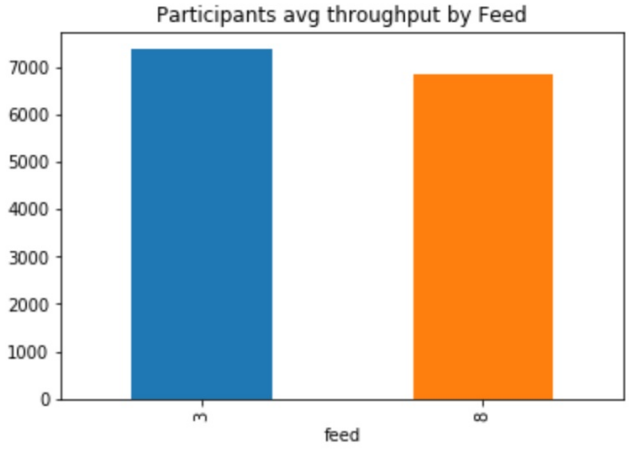
Unit Tests

There is an ongoing debate on how much code coverage that unit tests should strive for. Some believe that unit tests should just focus on covering services with mocked DAOs and let functional tests cover the rest. Others believe that unit tests should cover resource classes too. Dropwizard provides support for those who are in the latter camp. I side with the former so I didn't use the Dropwizard testing module. Spring test allows you to easily inject mocked dependencies via annotation with its test configuration annotation.



Like I said earlier, one of the biggest complaints against Java in general is its verboseness.

Lightweight (Dropwizard) apps load faster and are easier to debug than heavyweight (Spring Boot) apps.

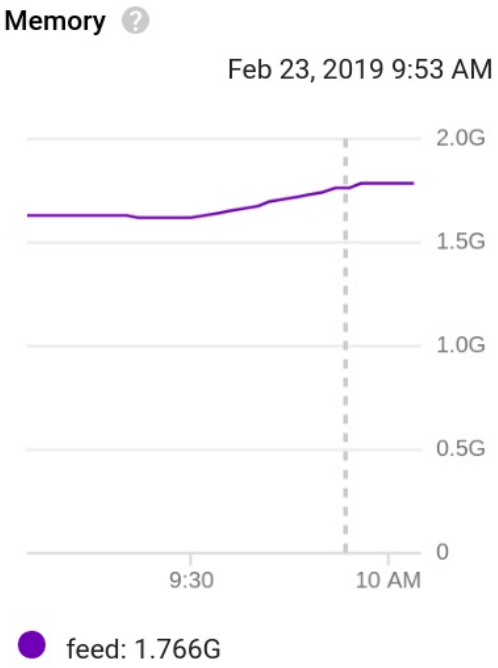


Code Size

It takes a lot of LoC (Lines of Code) to do almost anything in Java. How does this Dropwizard service compare against the Spring Framework service in terms of the size of each project respectively? At the time of this evaluation, the Dropwizard microservice is composed of 35 classes with a total of 3,266 LoC. The largest class is the Redis class (most probably because of the Hystrix integration) with 233 LoC. The Spring Boot microservice is composed of 41 classes with a total of 2,296 LoC. The largest class is a Swagger generated support class of 232 LoC.

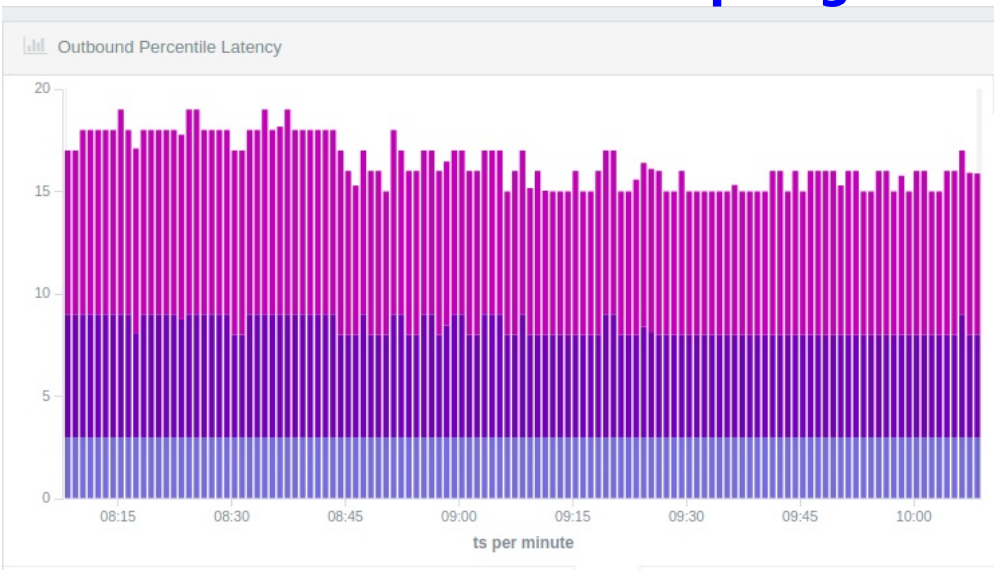
Remember that both projects start with some Swagger generated code. Let's remove both that code and the unit tests from consideration and look only at hand written code that implements each service. Dropwizard has 1,857 LoC and Spring Boot has 774 LoC.

How can the Spring Boot microservice implement the same requirements as the Dropwizard microservice in so much less code? Like I said early in the design section, both technologies use annotation based design patterns. Dropwizard uses it a little but the Spring Framework uses it a lot. In order to accomplish that, the Spring Boot microservice has a lot more dependencies on infrastructure based components. The uber jar for the Dropwizard microservice is 24 GB in size and holds 16 K classes. The uber jar for the Spring Boot microservice is 71 GB in size including 191 other jars totalling almost 45 K classes



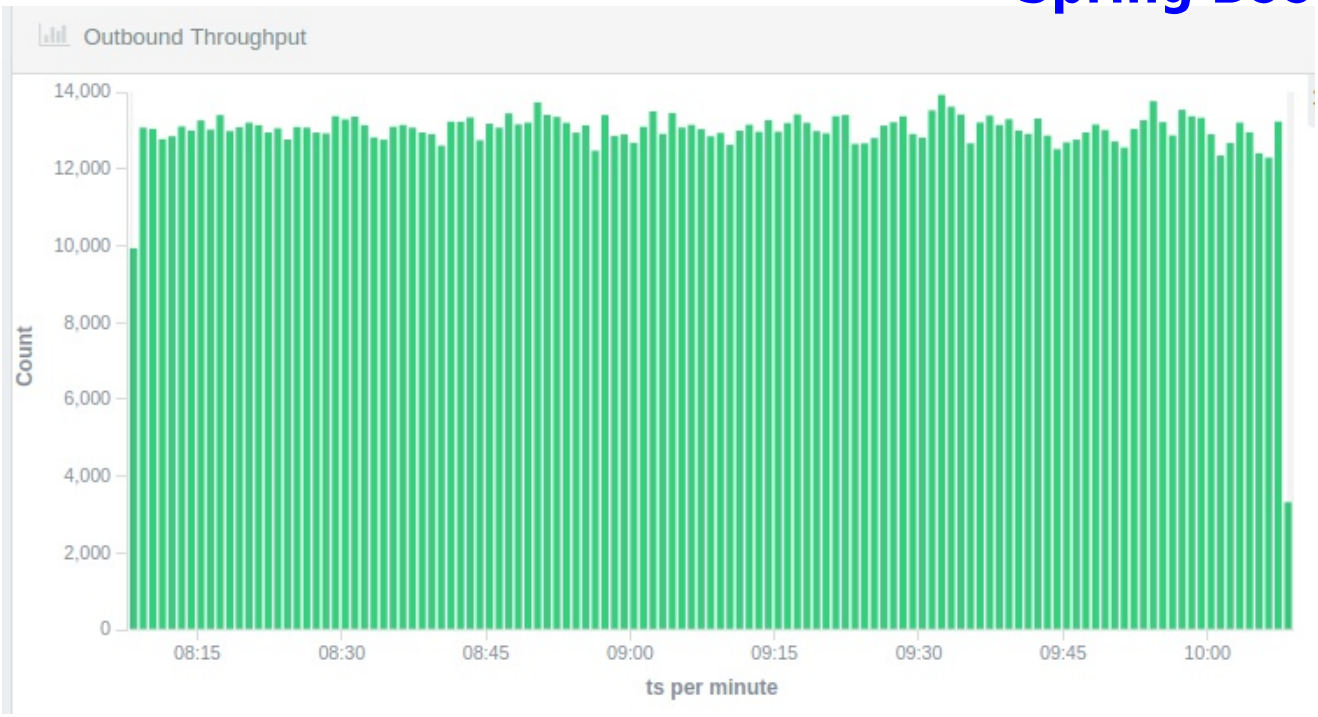
This is what I mean by lightweight vs heavyweight. Does it really matter? Lightweight apps load faster than heavyweight apps. A faster load time might lead to a shorter outage when Kubernetes starts restarting pods in order to reset a destabilized cluster.

Spring Boot



When exceptions gets logged in the code, stack traces for lightweight apps will be shorter than the stack traces for heavyweight apps. Since there are fewer lines of the application log to study, it might take less time for the developer to debug lightweight apps than heavyweight apps.

Spring Boot



How do the two microservices perform under identical load? I used my standard load test configuration which is 3 threads forever creating 10 users, friending each user on average 3 times then posting 10 outbound stories (of 150 words each) for each user. I had to rerun the Dropwizard test because of the need to establish a new baseline since I replaced Kong with that custom proxy. I let both tests run for 2 hours, then collect and analyze the performance data. Average throughput for outbound post requests to the Dropwizard microservice was 18,907 per minute. Latency was 4 ms on average and 10 ms on the 99th percentile. For Spring Boot, average throughput was 13,068 RPM and latency was 3.5 ms on average and 8 ms on the 99th percentile.

Load Test Results and Overall Summary

In conclusion, the Spring Framework is more enterprise focused than Dropwizard. The Spring Boot microservice had 30% less code and 60% less hand written code but also ran with 31% lower throughput. If managing complexity (for more predictable releases) is your primary concern, then consider choosing Spring Boot. If efficiency and resource utilization is more important (i.e. a smaller cloud bill), then go with Dropwizard. If you still want to use Spring Boot but in a more efficient way, then simply refrain from loading or using Spring Data and be okay with writing more code.