Revisiting Scala

How microservices in Scala compare with that of other programming languages.

by Glenn Engstrand

I use this blog and accompanying github repo to explore various technologies and programming languages by using them to implement the same news feed microservice, a digital Rosetta Stone of sorts. Three years ago, I developed such a microservice in Scala. While there was much merit to the service, it had a lot of code complexity and lackluster performance when compared to the other microservices here. I decided to try again in Scala using some different, often older, technology and programming paradigm choices. Were there any real advantages with the new version?

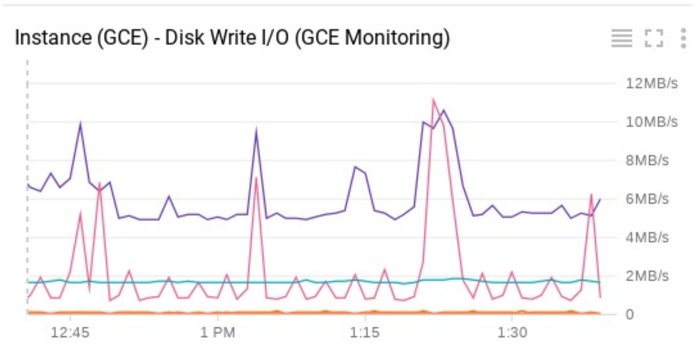
The Scala microservice that I wrote back then is in a folder under the clojure-news-feed repository called server/feed2. The service integrated with Finatra and was organized primarily around a technique known as the inheritance based Cake Pattern.

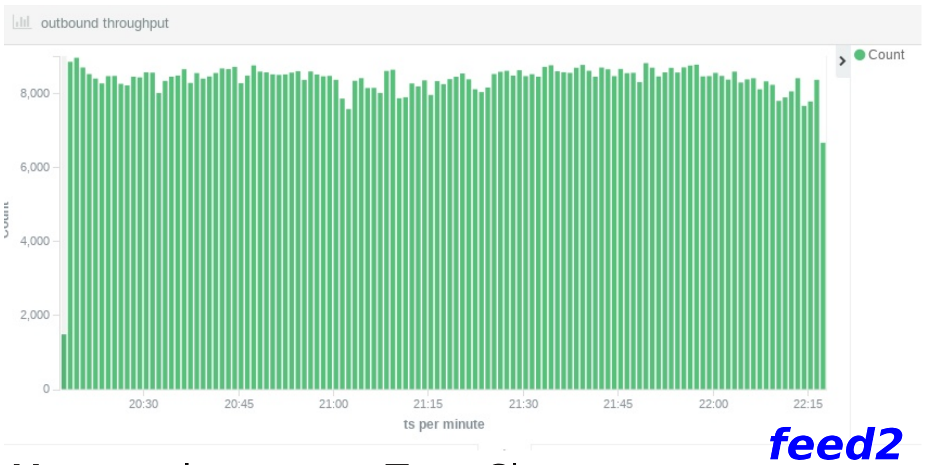
The Cake Pattern uses traits with self type annotations and mixes them into entity classes at object creation time using the “with” keyword. I chose the Cake Pattern at that time because it reminded me of the mixins in Ruby. I have always been intrigued by mixins because they afford multiple inheritance while avoiding the “dreaded diamond” problem.

Finatra is an open source web service framework developed by the folks at Twitter. It is based on another open source project called Finagle which, in turn, is based on yet another open source project called Netty. You will find many of the most popular and well engineered projects layered like this. Finatra is almost six years old.

This time around, I decided to do things differently with the server/feed6 project. Instead of Finatra, I integrated the service with Scalatra. Instead of the Cake Pattern, I used Type Classes. Instead of JDBC, I used Doobie. I also modernized the API itself with Swagger.

Scalatra is an open source web service framework based on Jetty. It is almost nine years old and is similar, in principle, to the Ruby based Sinatra project. Jetty is an open source servlet container technology brought to you by the makers of the Eclipse IDE.





Type Classes are not actually classes at all. It is a programming technique that combines the use of traits, currying, and implicits in order to implement ad-hoc polymorphism.

feed2

Most engineers see Type Classes as a way of separating out data structures from algorithms but I like to think of it as a Scala-idiomatic form of Dependency Injection. The concept of Type Classes first appeared in another functional programming language called Haskell, the first version of which was released in 1990.

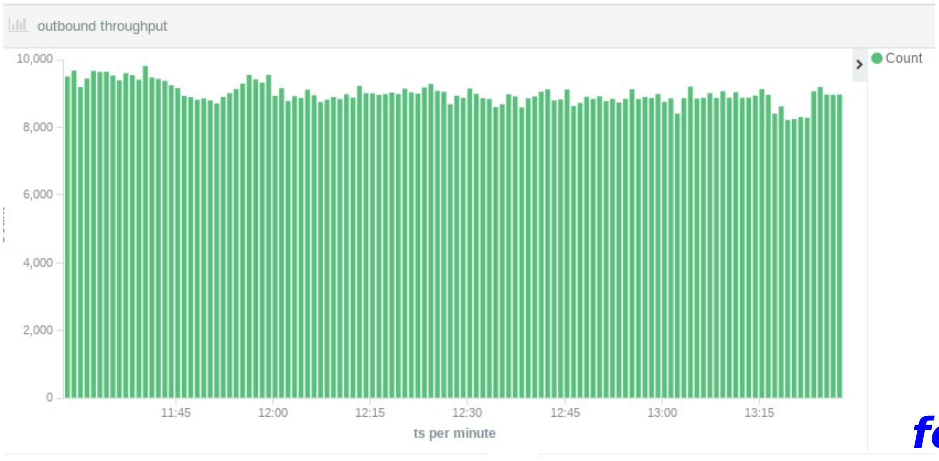
JDBC allows Java applications to connect to relational databases. Doobie is an open source project that allows Scala developers to use JDBC in a way that is more conducive to functional programming.

Swagger is a Model Driven Software Development system that accelerates microservice development by generating a lot of the API code from a YAML specification. Swagger was first published in 2010 but MDSD has been around since the 1980s.

There were some other differences too. The feed6 project depended on more recent versions of the drivers for the underlying data sources. The feed6 project used the ElasticSearch specific high level REST client whereas the feed2 project used a general purpose Apache HTTP client to communicate with ElasticSearch.

Let's look at some rudimentary code metrics on the two projects to see if we can make any valid comparisons as to complexity.

The feed2 project is composed of 2,724 lines of hand written Scala code in 19 files across 59 commits. On average, that is 143 lines of code per file. The biggest file has 594 lines of code.

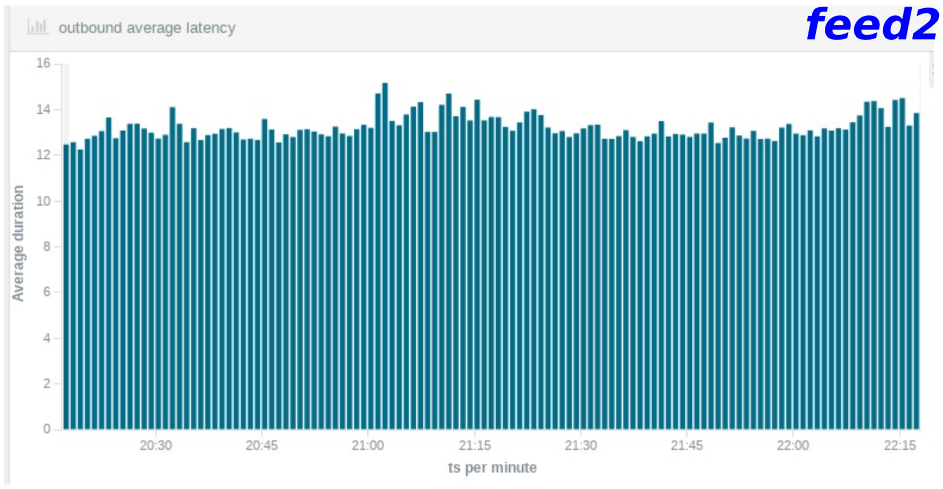


feed6

Copyright © 2018 Glenn Engstrand

The feed6 project is composed of 1,118 lines of Scala code in 28 files across 7 commits. Over half of that is generated by the Swagger templates. On average, that is 40 lines of code per file. The biggest file has 101 lines of code.

feed2



Lines of code may not be a guaranteed measure of complexity but in this case the comparison is valid. The feed6 project is much less complex than the feed2 project, mostly due to the use of Type Classes and Doobie.

What about the memory footprint of the two services?

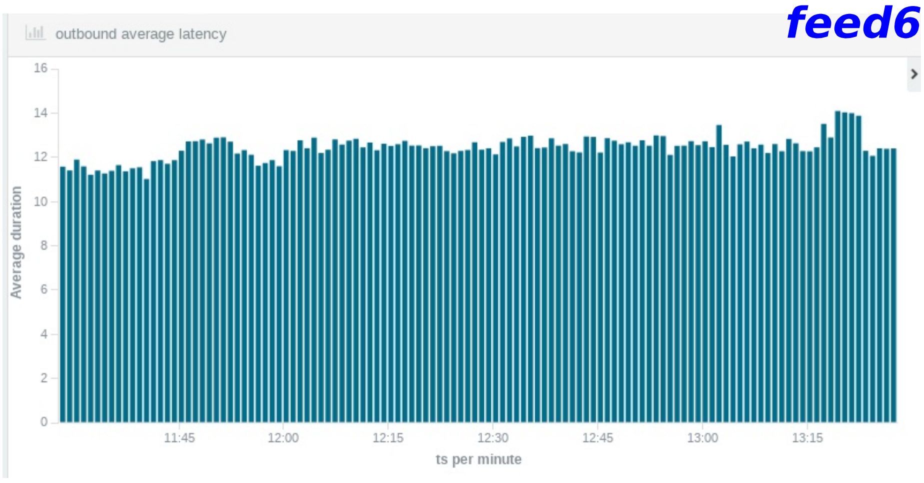
The inheritance based Cake Pattern in feed2 led to a design where you had entity classes with all of the abilities to communicate with the dependent data stores mixed in at object instantiation time.

I was following functional programming best practices with immutable objects so these somewhat heavyweight objects were being loaded into memory with each request. The data store aware traits have no direct knowledge of the specific entity classes that they were mixed in with so a memento pattern had to be used as well.

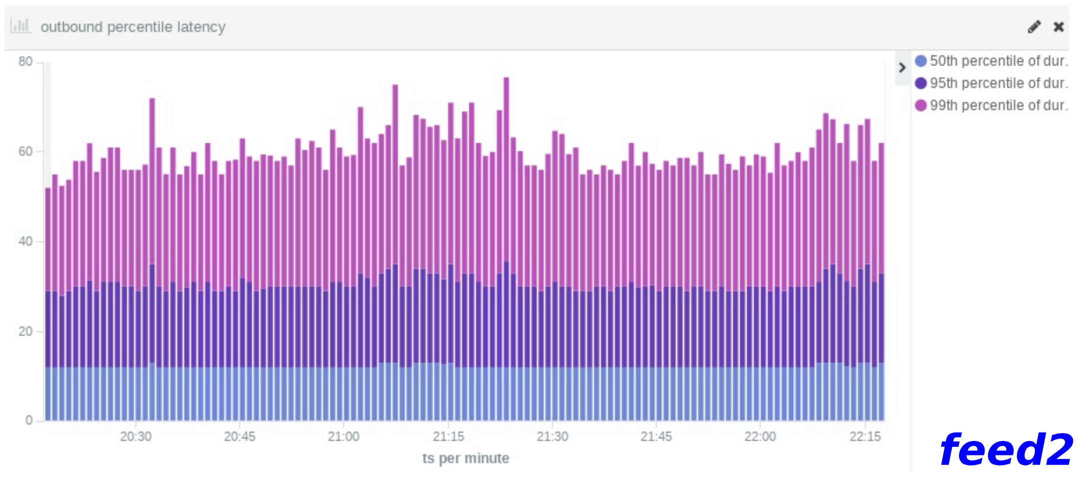
Alternatively, the Type Classes approach in feed6 led to a stratification of objects into resources, services, DAOs (Data Access Objects), and POJOs (Plain Old Java Objects). I used Type Classes to inject the DAOs and data source specific client drivers. Only the lightweight POJOs were created with each request. Everything else were singletons.

This yielded much lower Garbage Collection pressure in feed6 than in feed2.

feed6



Copyright © 2018 Glenn Engstrand



feed2

Were there any differences in the two services respective threading models that we should be made aware of?

Because Netty based services use asynchronous I/O to handle their socket connections, they do not pre-allocate a thread with each request. Instead, the requests gets handled in a worker thread managed by the NioServerSocketChannelFactory. This is a highly scalable approach in reactive systems because you don't need enough threads to handle all requests currently being processed; however, the agreement is that no blocking calls will be made in that worker thread.

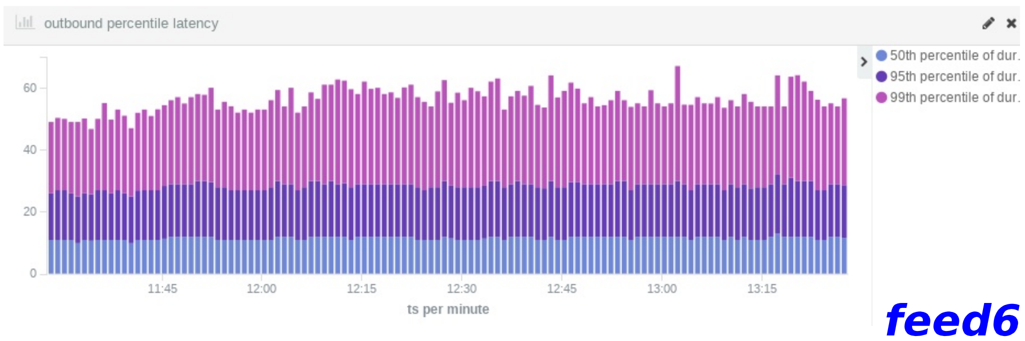
Jetty based services do pre-allocate a thread with each request. This means that you can make I/O blocking calls directly in code that is handling a request. If the I/O blocking call turns out to be slow, then it won't block the other requests being handled by this service.

That is not true for Netty based services.

If the API must return a result from a blocking I/O call, then it must wrap that call in a Future then preemptively wait for the Future to complete. The Future class from Java 8 won't work here. This leads to more code complexity and lower parallelism.

How did the two services performed under load?

I ran all of my load tests for the two services on the Google Kubernetes Engine. Each test run took at least two hours to complete. All API calls were proxied through Kong where the performance data was collected then sent to ElasticSearch and surfaced in Kibana (hence the screenshots).



feed6

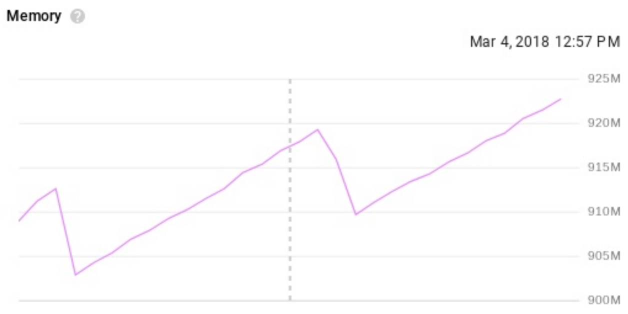
Copyright © 2018 Glenn Engstrand

I used StackDriver (acquired by Google in 2014) to monitor the utilization based metrics.

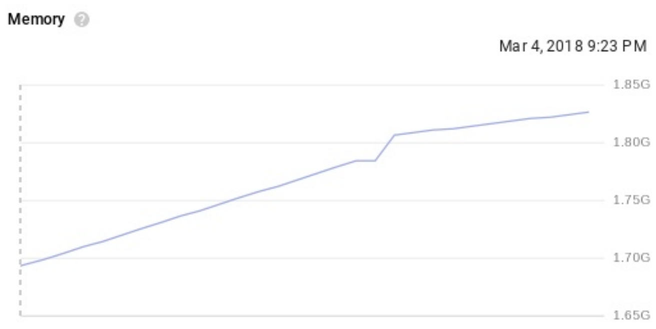
Average throughput for feed2 was 8,400 outbound post requests per minute with a mean latency of 13 ms, a median of 12ms, a 95th percentile of 19ms and a 99th percentile of 30ms. Average throughput for feed6 was 8,982 outbound post requests per minute with a mean latency of 12ms, a median of 12ms, a 95th percentile of 17ms and a 99th percentile of 28ms.

What conclusions can we draw from this investigation?

The feed6 service had 7% more throughput than feed2. The feed6 service also had 8% less latency than feed2. The feed6 project required about half of the RAM needed by feed2.



feed6



feed2

The feed6 service had significantly less complexity than the feed2 service. Lines of code is easy to calculate and compare with. The feed6 project was 59% smaller than feed2 in terms of code size.

The differences between feed6 and feed2 resulted in some nice improvements in complexity, performance and capacity but how did it effect the standing of the Scala services when compared to the other microservices?

While feed2 was the biggest memory hog, feed6 came in third behind feed4 and feed5. In terms of performance, Scala remains in third place behind feed3 and feed4.

The code complexity improvements in feed6 moved Scala from dead last to third place. It still had more lines of code than the feed (Clojure) project or the feed4 (node.js) project but less lines of code than the feed2, feed3 (Java), or feed5 (python) projects.

Copyright © 2018 Glenn Engstrand