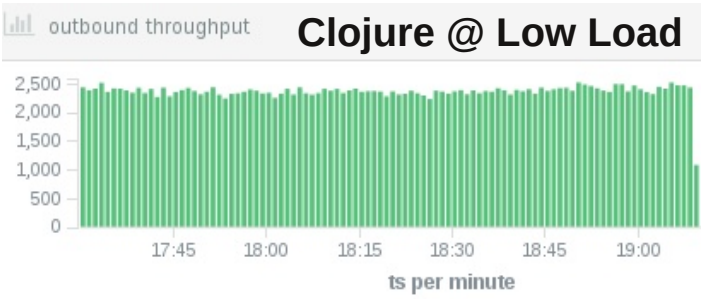
DropWizard vs Ring

The Java Framework Strikes Back

by Glenn Engstrand

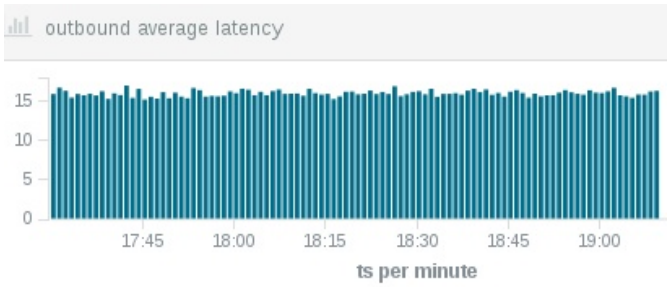


Let's start with the architecture for this micro-service. I decided to go with DropWizard version 1 (earlier versions were based on Java 7), Google Guice, and Swagger.

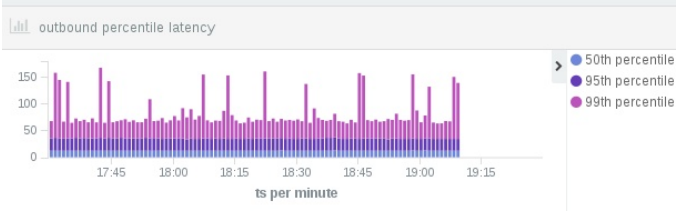
If you have read any of the recent blogs here, then you would know this. I take an emerging technology and use it to implement a rudimentary news feed micro-service. Then I test it under load and share the results. This is what I have done for Clojure and Scala. In today's blog, I return to Java, only this time using the functional programming features available in version 8.

Architecture

DropWizard is a Java framework for building RESTful micro-services. Though not as popular as Spring Boot, DropWizard is more popular than many similar Java frameworks, including the Vert.x project. I picked DropWizard over Spring Boot because it has a traditional open source community. There is no shepherding organization trying to upsell you into enterprise support or features.



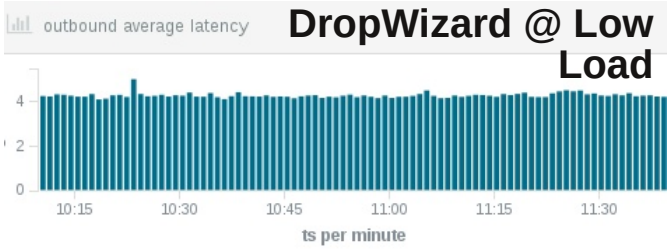
Because of this, DropWizard is actually an opinionated, convenient packaging of many other popular, time tested, best-of-breed open source Java projects including Jersey 2, Jetty, JDBI, and Jackson.



Guice is the most popular choice for DropWizard Shops.

Guice is Google's open source project for Dependency Injection. The biggest competing open source project for DI is Spring. There was available community integrations of DropWizard and Spring for versions prior to 1 but not for version 1. It seems to me that, for now, Guice is the most popular choice for DropWizard shops.

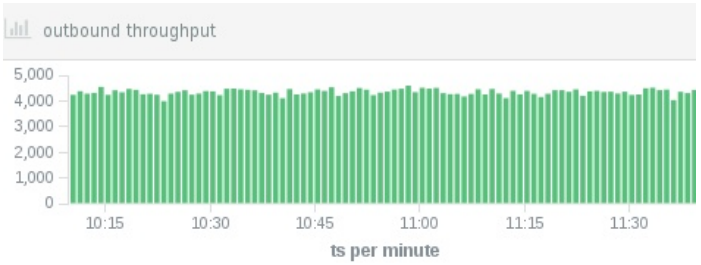
There has been, as of late, a bit of a resurgence of interest in Model Driven Software Development and the open source project Swagger is leading the pack on MDSD adoption. For this project, I wrote my own templates to generate both the APIs and the models using Swagger.



DropWizard @ Low Load

Not only am I going to talk about how I wrote this micro-service, I am going to document what I discovered when I tested this micro-service under load. In order to provide some kind of basis for comparison, I will be presenting the load test results of this project with those of the Clojure version of the same micro-service. Let's cover how those two projects are different.

Just in case you have not already read my previous blogs, this Clojure micro-service uses open source projects Ring, which integrates with Jetty to provide an application container, and Compojure to handle the request routing logic.

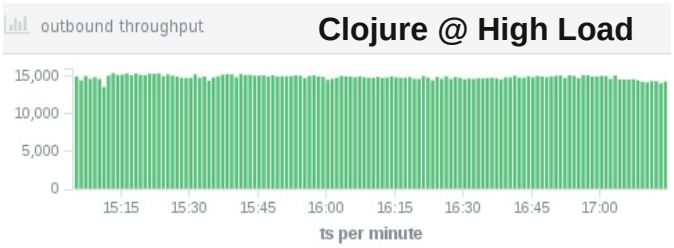


Probably the biggest and most obvious difference is the runtime environmental differences between the two programming languages. Though Clojure runs in the JVM, it is interpreted and this service is coded to always call through the reflection API. The Clojure app runs in Java 7 whereas the DropWizard app runs in Java 8 as an uber jar that gets JiT compiled.

Differences

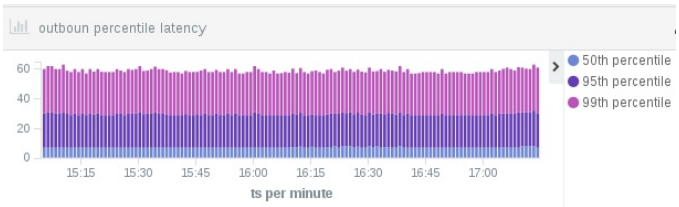
There is a trend amongst the modern Java frameworks, such as DropWizard and Spring Boot, to use Java annotations to control DI and to specify the REST interfaces.

For the DropWizard service, I used generic classes for all of the different data stores. Clojure is very light weight on its object orientation with no direct language support for generics.



As indicated previously, this DropWizard app uses Guice for DI. The Clojure app also uses DI but it implements DI in a naive way by simply evaluating a Clojure map that is in a separate file considered to be part of the configuration.

Though both apps return JSON, only the DropWizard app accepts JSON. The Clojure app POST handlers accept application/x-www-form-urlencoded data when creating entities.



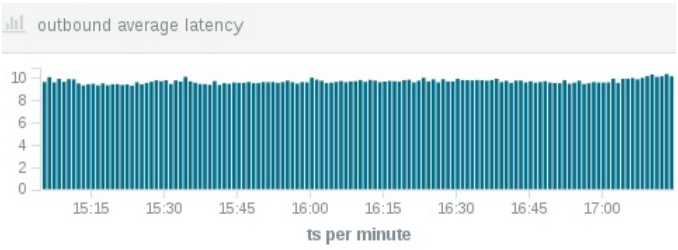
The DropWizard app runs natively in its EC2 instance whereas the Clojure app runs in a Docker container in host mode.

The reason why I picked the Clojure version of the micro-service to compare against the DropWizard version is because of all of their similarities.

Similarities

The most significant similarity between these two projects is that they both use Jetty as the application container.

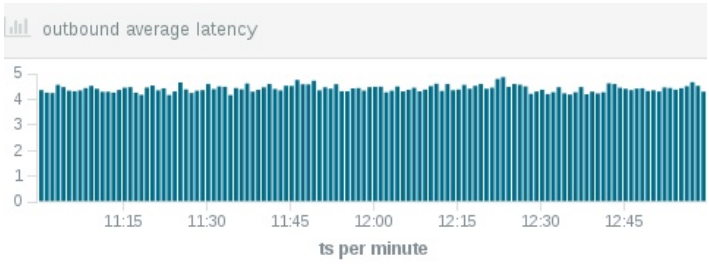
Another important similarity is that, while Java 8 is very different from Clojure, both micro-services are written in a functional programming style.



The DropWizard app parses JSON yet the Clojure app does not.

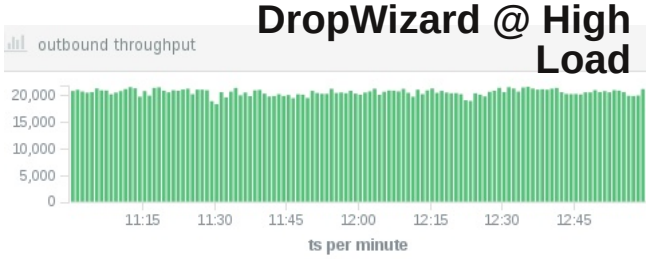
Though the API client libraries are not identical, both projects use the same technology stack when communicating to their various data stores. They also use the same versions of Elastic Search, Cassandra, MySql, Redis, and Kafka.

Now that we have compared DropWizard with Clojure in terms of code, technology, and project organization, how did the two micro-services compare in terms of performance under load? In order to answer that question, I used my standard load test. I ran two tests each per micro-service. The second test tripled the load from the first test. All tests lasted about two hours. Like all previous blogs regarding news feed performance, we are looking only at outbound post data.



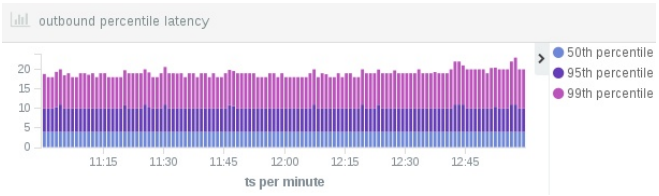
In terms of raw performance, DropWizard is the clear winner by being three times faster than Clojure across all performance metrics in all tests.

DropWizard @ High Load



In terms of scalability, DropWizard still out performed Clojure but the results were not as clear. In the first round of tests, the DropWizard service had 60% higher throughput than the Clojure service. That gap narrowed, when we tripled the load, to only 30%.

DropWizard is the clear winner.



Components