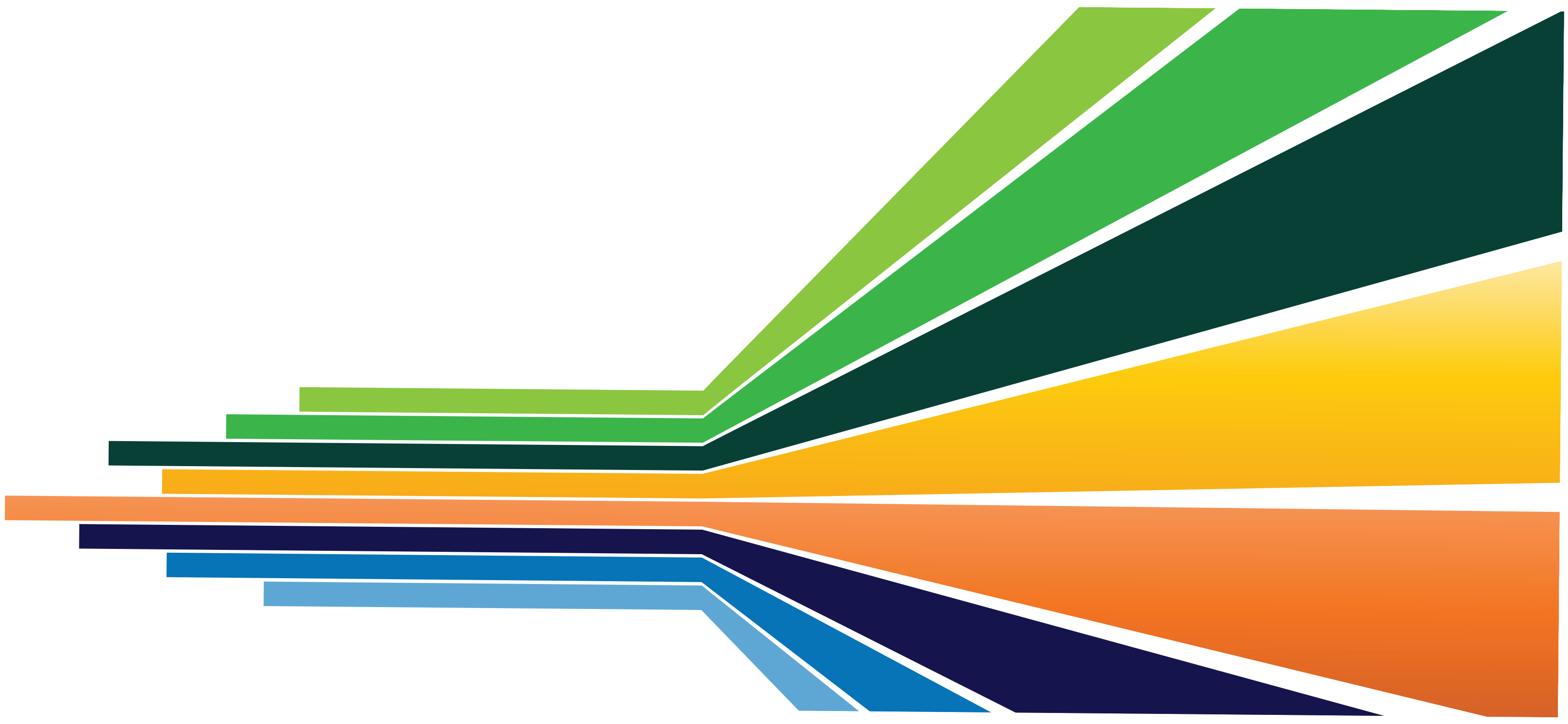
Baratine + Lucene: Exposing Lucene as a Web Service

Abstract: We present Baratine as an asynchronous facade that can be placed in front of an existing library with no modifications to the library's code base. Thus accomplishing two tasks: (1) exposing the library as a web service available to any language and (2) simplifying the requirements to have a nonblocking scalable (shardable + able to be partitioned) web service.

We have done this to illustrate how Baratine's POJO platform provides an API-centric approach to building high performing microservices.

Through performance benchmarks we show that a good purpose system built in Baratine can compete with a special purpose system built around a specific library.



The ability to expose an existing application or library as a web service without any code modifications is ideal. With Baratine this can be accomplished in two tasks: implementing a service portion (SOA) and implementing the client library for communication. In this way, Baratine can transform an existing library or application into a standalone web service. The Baratine services communicate with the existing library, and the Baratine clients service requests from the outside world. Within this article we will explore exposing the Apache java library Lucene as a high performing web service.

Lucene is an open source project from the Apache Foundation defined as: “[Lucene is] a high-performance, full-featured text search engine library written entirely in Java. It is a technology suitable for nearly any application that requires full-text search, especially cross-platform.”

This example serves as a blueprint for bringing existing applications or libraries online as functioning web services. Our example transforms the Lucene library into a web service with the following traits:

- asynchronous messaging service
- a public WebSocket API for Lucene
- deployed in a standalone managed server.

This will create a high efficient search server.

Running the Example

1. Install maven baratine plugins https://github.com/baratine/maven-collection-baratine
2. Execute **mvn install**
3. Change to service directory and execute **mvn baratine:run**
4. Open lucene-plugin/client/src/web/index.html in latest browser

Project Contents and Overview

The Baratine Lucene project can be viewed at https://github.com/baratine/lucene-plugin. The example includes the Java services for Lucene, and a JavaScript client for browsers or for node.js. The project also uses the Angular.js framework to wire the backend services. This means that with just Baratine and a fronted framework, a high performing web service can be easily created. Majority of the work is done in the following files:

Clients:
-Baratine Lucene Client (lucene-client.js): Lucene method functionality from client, used for Browser results
-Baratine Javascript Client (baratine-js.js): Protocol library distributed with Baratine, which communicates with the service using HTTP or WebSockets and Baratine's JAMP Protocol

Service:
-Lucene Reader service (LuceneReaderImpl.java) and writer service (LuceneWriterImpl.java): Responsible for reading and writing search indexes, bridging between Lucene's synchronous/blocking API and Baratine's asynchronous/messaging.
- Public API service (LuceneFacadeImpl.java): Responsible for proxying method requests to Lucene library.

What we have done is publish a Lucene HTTP/WebSocket API using Baratine Services.

Two main tasks:

- Publish a HTTP/WebSocket Client API.
- Bridge between Lucene's synchronous library and Baratine's asynchronous services.

Baratine Service #1: LuceneFacade

LuceneFacade is the API for the published HTTP/WebSocket, accomplishing the first task. Its implementation is LuceneFacadeImpl. The published API is asynchronous, using Baratine's Result as a callback holder for the result. The implementation is single-threaded and non-blocking, eliminating the need for synchronization.

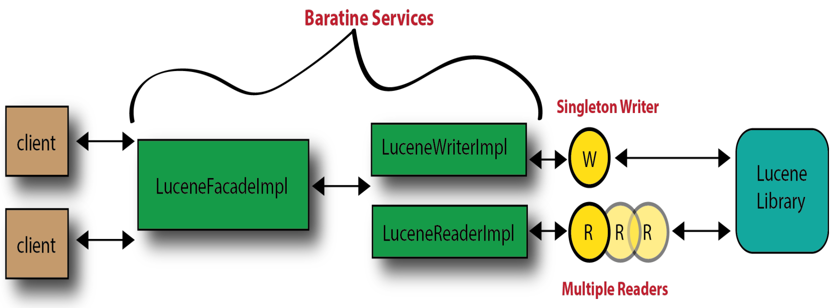
Baratine Service #2: LuceneReaderImpl

LuceneReaderImpl implements the search query service, which bridges between Lucene's synchronous library to Baratine's asynchronous APIs. Because Lucene's search is multi-threaded and blocking, LuceneReaderImpl uses Baratine's multi-worker support (@Workers). The multi-worker support is similar to a database connection pool. Users of LuceneReader see an asynchronous service without needing to know that is implemented as a multi-threaded service.

Baratine Service #3: LuceneWriterImpl

LuceneWriterImpl implements the index updating service. It is a single-threaded service that batches write requests to Lucene for added efficiency. As load increases, LuceneWriter becomes more efficient, because the write batch sizes increase automatically.

Not only does this illustrate how simple it is to encapsulate a method call as its own self contained service, it also prevents calls to these methods from blocking other client requests. Lucene does not provide an asynchronous API, however since Baratine services are asynchronous, we allow processing to happen while these methods are called continuously. A high level overview of the new architecture can be pictured as the following:



HTTP/WebSocket Client API

The client API is the Java interface of the service, in this case the LuceneFacade.java. Each service has its own address, using a URL syntax. Methods on the service are called like normal method calls. Baratine's Javascript library manages the details, providing a method interface for the protocol, which uses JSON method calls on HTTP or WebSockets.

Lucene API methods:

void indexFile(String collection, String path, Result<Boolean> result) throws LuceneException;

void indexText(String collection, String id, String text,Result<Boolean> result) throws LuceneException;

void indexMap(String collection, String id, Map<String,Object> map, Result<Boolean> result) throws LuceneException;

void search(String collection, String query, int limit, Result<List<LuceneEntry>> result) throws LuceneException;

void delete(String collection, String id, Result<Boolean> result) throws LuceneException;

void clear(String collection, Result<Void> result) throws LuceneException;

Clients then call in directly to these methods.

We know that a service is at a URL, so how do we create a client to talk to that service?

How to make clients

1. *Create ServiceRef*
2. *Call the lookup*
3. *Save the returned proxy*
4. *Call methods on the proxy*

Long-lived clients, like a Java app-server or Node.js service, can share a single thread-safe connection, because the protocol is asynchronous, using messages. Fast requests, like a search in a memory cache, can complete out-of-order before earlier slow requests, like Lucene search that calls a mysql database. Using a single connection can even improve efficiency, because multiple calls can be batched, which improves TCP performance.

JavaScript Client

For the Lucene example, the API includes methods to search for a text document and to add new documents to the search engine.

Baratine implements interfaces as classes allowing us to call methods directly on the interface we choose to expose. The Result of a call a callback that will return the result of the search or notify that the index has completed. Because Baratine is asynchronous, the call to search or indexText does not block.

Baratine supports creating a websocket connection from a client to the calling service. This allows full duplex communication over a single TCP connection, as websockets aim to bring native desktop responsiveness to web services.

The JavaScript client for Lucene follows the Java API. The following is extracted from the lucene.js file to show how the Baratine calls translate into Javascript.

First, a new connection is created with a new Jamp.BaratineClient at the server's HTTP URL, which is “http://localhost:8085/s/lucene”. Typically, the client will be a long-lived, used for many requests.

Creating a client:

1. Pass in service url to Baratine
   i. this.client = new Jamp.BaratineClient(url);
2. Call a lookup on the ServiceRef
3. Save the returned proxy from lookup
4. Call methods on the proxy

Search

```
{
  this.client.query(“/service”,
 “search”, [coll, query, limit],
onResult);
}
```

Search is straightforward, proxying requests directly to the backend.

Index

```
{
  this.client.send(“/service”,
“indexText”, [coll, extId, text]);
}
```

Indexing text is implemented as a non-blocking method.

Services are exposed by way of Baratine: creating a Bean which is managed by Baratine.

Java Client

The service can also be called from a Java client, such as an web-app using Lucene. As with the Javascript client, the Java client will typically be a long lived connection. Since the client itself is thread-safe, it can be used efficiently from a multithreaded application.

Async Java Client

```
@Inject
@Lookup("public://lucene/service")
  LuceneFacade _lucene;
```

When a multi-threaded, synchronous Java application uses a Baratine service such as the Lucene service, it will generally use a synchronous, blocking call to a synchronous version of the service API. The Baratine client proxy acts as a bridge from the synchronous client to the async Baratine service. The sync version of the lucene facade looks like:

LuceneFacadeSync

```
public interface LuceneFacadeSync
extends LuceneFacade
{
  List<LuceneEntry> search (String
collection, String query, int limit);
}
```

The client call looks like a plain Java method call, as in the code below. Since the ServiceClient is thread-safe and can be used for multiple Baratine services, it can be used as a singleton. In fact, because of Baratine's internal batching and messaging, it's more efficient to use a single client shared across threads.

Sync Java Client

```
ServiceClient client =
ServiceClient.newClient(url).build();

LuceneFacadeSync lucene;

lucene = client.lookup("remote:///
service").as(LuceneFacadeSync.class
);

List<LuceneEntry> result =
lucene.search(collection, query, limit);
```

As you can see, client creation is straightforward as Baratine's flexible architecture allows efficient API protocol design.

Lucene Server Implementation

The Lucene server has two tasks: publish the client API and act as a bridge from Lucene's multithreaded blocking implementation to Baratine's asynchronous architecture. The example uses three Baratine services to implement the service:

- Client API Facade Service
- Reader Service for searches
- Writer Service for search index updates

For the Lucene server, the reads and writes are split into two services because reads and writes behave differently, thus the services are customized to effectively support those differences.

Writes benefit from a single writer thread, which improves efficiency under heavy load because it can batch multiple writes into a single commit.

Reads for Lucene can benefit from multiple reader threads. Lucene searches can potentially block on a database query, tying up the thread. With multiple threads, a separate thread can process a new search. Note that the multiple threads are only required because Lucene might use a slow, blocking service. If Lucene was memory-based, or asynchronous itself, a single reader thread would be more efficient because of CPU caching.

The implementation of the Lucene service is contained in five major files, which we'll describe briefly.

LuceneFacadeImpl.java

The client API is implemented by the LuceneFacade Baratine service. It primarily dispatches requests to the Reader and Writer services. The facade will have a bigger role when we use multiple services to partition the Lucene server in a following article. For this example, it is useful to focus the client API on simplicity, making sure the server works to make the clients easier.

LuceneIndexBean.java

Because the Lucene library is written as a singleton instance, the services for readers and writers share the same LuceneIndexBean. This design is based around Lucene's own design; we're using Baratine to work with the existing architecture, instead of trying to force Lucene to follow Baratine. If we were building a Baratine service from scratch, instead of adapting an existing library, we would likely choose a different architecture. The shared LuceneIndexBean singleton is be injected into the reader and writer services when they are initialized.

Since we are still using the underlying Lucene library, we can simply implement the methods from our LuceneFacade interface and add any helper methods for our own specific indexing (i.e. escaping special characters, using another data storage, limiting the size of commits, etc.)

LuceneWriterImpl.java

The Writer service takes requests from the facade and updates Lucene's indexes. It has a single worker thread, which writes updates to Lucene for as many requests as its inbox has. When its inbox is empty, it calls Lucene's commit method to complete the writes. Under heavy load, the inbox will have more requests, which leads to a bigger batch, improving performance.

LuceneReaderImpl.java

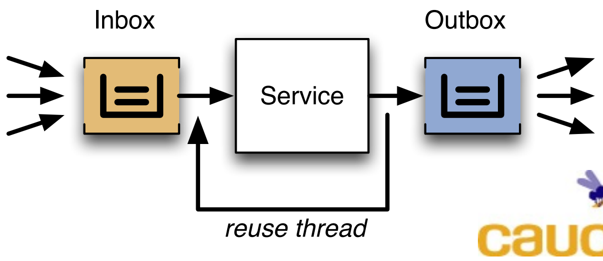
Because Lucene's searches can block waiting for a slow database, and because Lucene is multithreaded, the reader implementation uses a **@Worker** annotation to request multiple threads for the service. By annotating this service with ***@Workers(20)*** we provide a thread pool executor that can continuously dispatch threads to the reading service. Since the threads are only dispatched when needed, the multiple workers are low-cost. The multiple workers allows our Lucene service to serve multiple requests at once.

In general, this multi-worker feature should only be used for gateway services with an external blocking dependency, like a database connection or a REST call. Services designed for Baratine should use a single worker, because they should be designed as asynchronous, non-blocking services.

Baratine Architecture

What we have done is implement the Lucene API as a set of Baratine Services. Within Baratine, every ***service*** lives at its own unique URL and operates under a single-threaded, single-owner/single-writer contract. (The multi-worker bridge service used in the Lucene readers is an exception; it is used to bring external libraries into Baratine.) Requests to the specific service URL are queued to that service's ***inbox***, which guarantees an ordering of requests as they are processed.

The core Baratine building block looks like the following:



We can now summarize a Baratine service with the preceding picture and following points:

1. A Baratine service lives at a unique URL
2. A Baratine service has a single owning thread responsible for manipulating the service's data
3. Requests are queued into the service's inbox and processed in batches

Within Baratine, we have no need to add synchronization to our calls. Because each service is answered by only one thread, it is impossible for another thread to corrupt data that is being updated. This allows our classes to be POJO objects.

Performance and Comparison to Apache Solr

As some readers might have noticed, our example is similar to Apache Solr, which provides a server for the Lucene library. Solr is a good comparison because it's a familiar example and can be compared to directly.

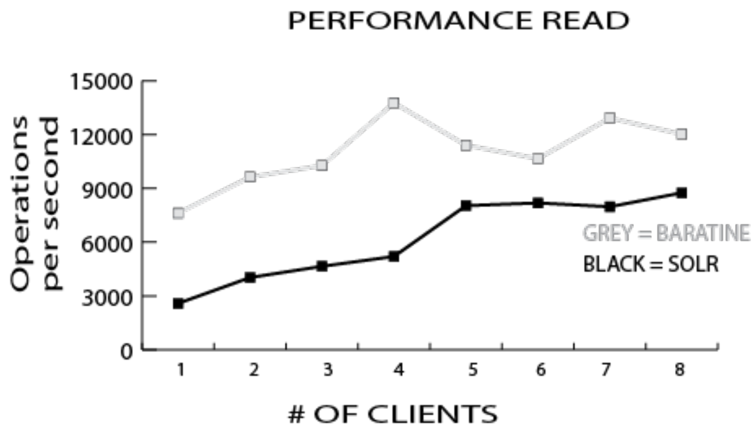
We benchmarked against Apache Solr with multiple numbers of clients. In testing against only ***reads***, Baratine was better or within 20% of Solr's performance. In a mixed ***read/ write*** benchmark, Baratine proved to be 3x faster than Solr.

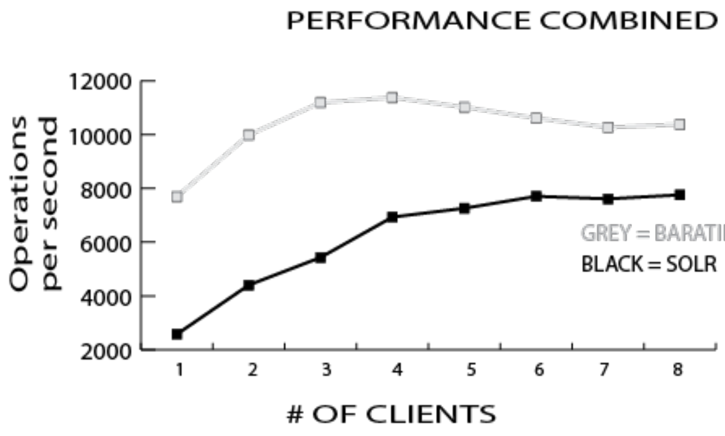
Testing was done with the following specs:

Intel(R) Core(TM) 2 Quad CPU
Q6700 @ 2.66GHz
Memory 4 G Speed: 667 MHz
HDD: ST3250410AS

java version "1.8.0_51"OS Linux deb-0 3.16.0-4-amd64 #1 SMP Debian 3.16.7-ckt11-1+deb8u2 (2015-07-17) x86_64 GNU/ Linux

The benchmark graphs are as follows:





As the results show, Baratine search (read) outperforms Apache Solr in a side by side comparison.

A mixed load of read/write requests, show competitive numbers.

Summary

What we have done for Lucene in this example can be done with any library or application. By wrapping a Baratine service as a facade into a library, we can transform any library (such as java.util) into an asynchronous service.

Many of the principles Baratine includes are reflected in the reactive manifesto (http://reactivemanifesto.org). Reactive applications are elastic, responsive, resilient, and message-driven. It's what is demanded by the Internet of Things where tens of thousands of devices are connecting to the same application. These principles are coveted by developers, yet difficult to implement in practice. Baratine's distinct POJO level abstraction defines a data and thread encapsulation level that absolves this difficulty. In this way, Baratine is a SOA implementation of a reactive platform that keeps developers programming in the object-oriented fashion they are used to. We believe that most new web applications will incorporate these principles while needing integration with their current systems. Thus, as we have shown in this paper, Baratine is a perfect fit for building both.

Baratine's value increases as you add functionality to the application. If for example, the decision was made to run live analytics on the queries being performed, a simple POJO class gathering statistics can be deployed on a Baratine Node and relay this information. It can provide this information as Near Real Time coming from the current updates of the BFS (Baratine File System), or it can precompute and batch these results to an outside source for consumption. Whichever the choice, Baratine's unified yet flexible architecture allows for the system to be designed specifically for the task at hand without limiting future potential. Because services always embellish the qualities of a needed web application, proof of concepts can go from white board APIs to deployed in a matter of minutes.

Baratine is currently in Beta (0.10), with a scheduled roadmap to have a production ready version launched Q4 2015. In this example, we've only touched on the capabilities of Baratine backing a Lucene web service.

Stay tuned for Part Two, as we tackle sharding and scaling this Lucene web service within Baratine!