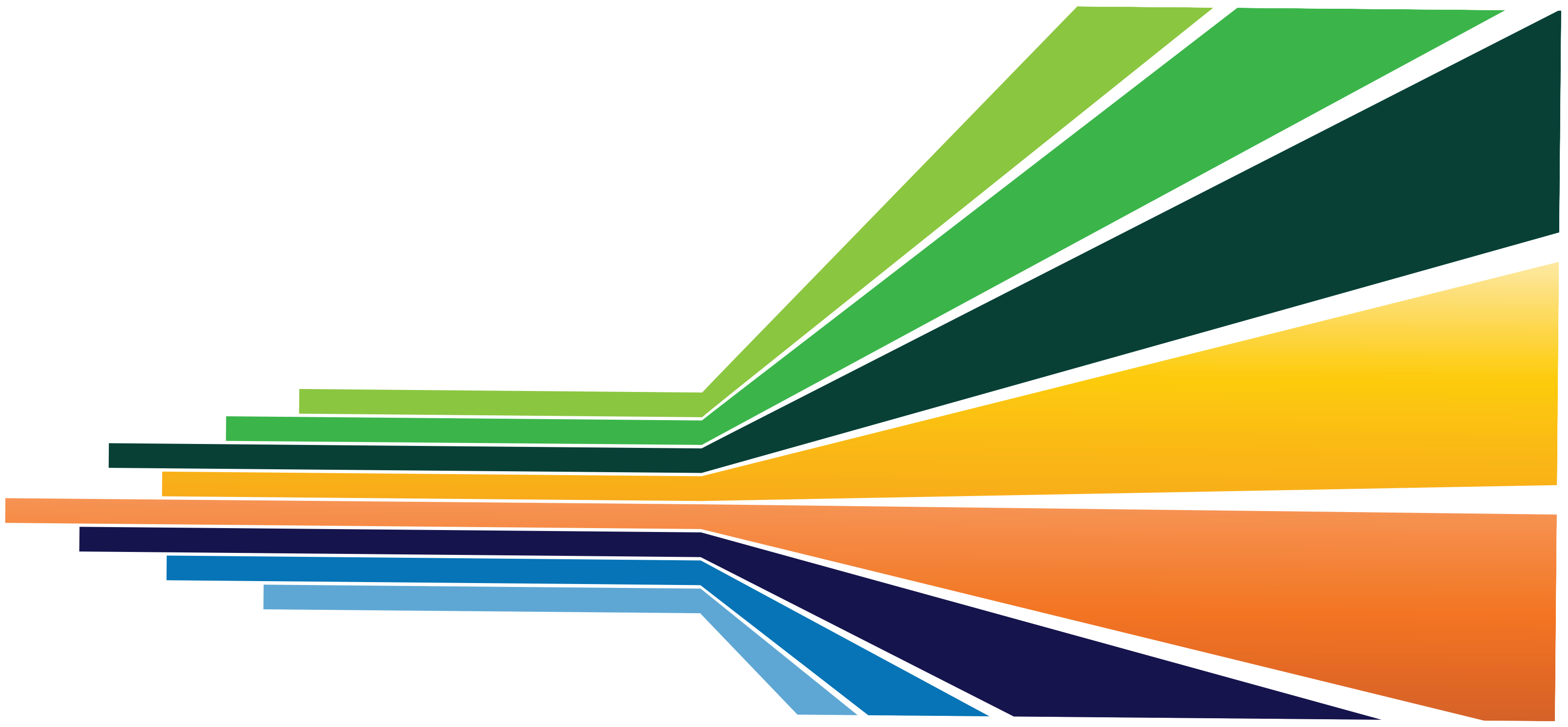
Baratine Auction Example

The Auction Application demonstrates how to use Baratine™ to build a Single Page Web Application.

Abstract: Baratine is a distributed in-memory Java service platform for building high performance web services. In Baratine, each service is responsible for its own data and operates on a single thread. In this way, an application's business logic can be split into a set of distinct services. Your API is transformed into a set of services.

Baratine simplifies the entire SOA backend of an architecture allowing greater collaboration and faster development cycles. In this walkthrough, we will cover a single-page auction example which is made up of ten microservices.



Article Table of Contents

======================
Part One: Baratine Schema
======================

Foregoing any further definitions, we have built a Baratine Auction Example to demonstrate many key concepts within Baratine's service orientated architecture. At an overview, Baratine internal services are utilized to create a single page application. Because this application is built on Baratine, inherently the application achieves:

1. ***Sharded services***: Named URLs provide a proxy into a service

2. ***Asynchronous Callbacks***: Method calls use Baratine's ***Result*** to form non-blocking callbacks, improving performance

3. ***Multiple Clients Available:*** Communication over Baratine's JAMP protocol allows clients of any language to communicate with services over HTTP, websockets or RPC.

4. ***In-memory***: Operational data within the application is stored in-memory and persisted to disk as a secondary storage

Baratine's programming model follows a single service ownership data methodology. A service shards its data among child nodes at unique URLs. In this sense, each self contained service in a Baratine cluster is responsible for storing and updating its own data. This interjects with the normal persistence layer most developers are used to working with.

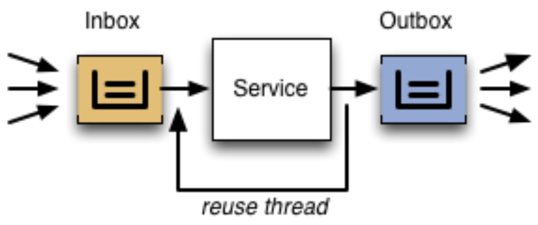


Fig. 1 Baratine Service Architecture

Baratine is a simple programming model that provides an ecosystem where the distributed and schema-free nature of web applications can flourish. Baratine moves business logic and processing onto a single node, whose jobs can share a single distributed file system for NRT data processing and updates. It has the ability to provide this information to a multitude of languages: anything that can speak JSON. Baratine can operate in-memory as well as persist to disk.

Running the Auction application

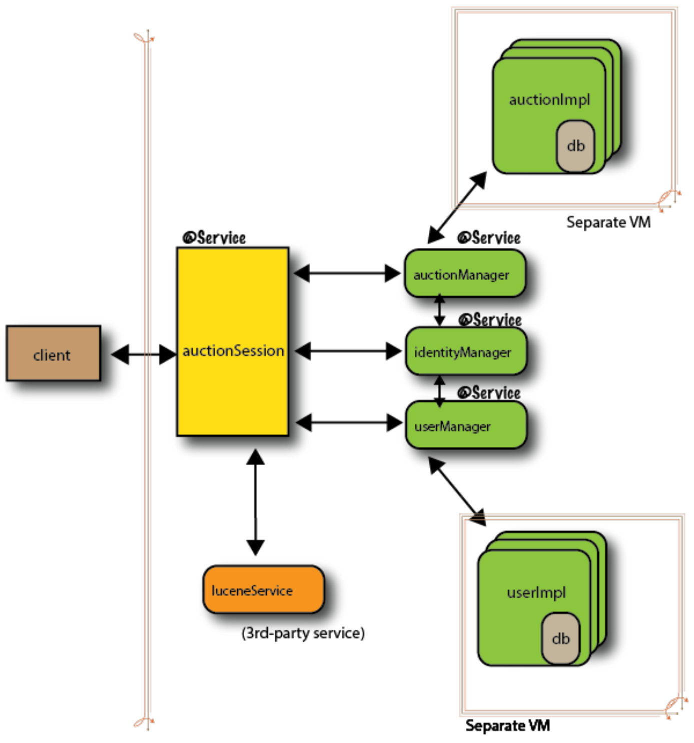
1. install lucene-plugin: https://github.com/baratine/lucene-plugin
2. execute ./baratine-run.sh
3. open latest browser and navigate to http://localhost:8085

==========================

Part Two: ***Architecture Overview***

==========================

The backend services are fully contained microservices. They are responsible for only the data they own and will query each other if business logic determines that data needs to be pulled from multiple services to fulfill a request.



=======================

Part Three: Service Structure

=======================

In order to see how Baratine simplifies creating fully encapsulated and high performing services, we can look at the following Java classes:

AuctionSessionImpl - implements a user session; invoked by the UI (index.html)
UserImpl - implements User; manages UserDataPublic class which contains user detail
UserManagerImpl - creates and manages users
AuctionImpl - implements Auction; manages AuctionDataPublic class which contains Auction detail
AuctionManagerImpl - creates and manages auctions

These classes utilize the following Baratine internal services:

DatabaseService - The database service gives access to the internal Kelp/Kraken database used to store & query
Store (key - value) - A simple key value store for accessing data within each service
Timer - allows you to schedule tasks to be run once or multiple times on a schedule
Event - publish events and to subscribe to events
WebSocket - Full Duplex communication over a single TCP connection

Baratine services begin with a named service URI, in the case of ***AuctionSessionImpl*** we can see the annotation beginning the class at:

```
@Service("session://web/auction-session")
public class AuctionSessionImpl implements AuctionSession
```

The ***@Service*** annotation marks the single-threaded applications inbox where requests are queued and batch processed. As the AuctionSession is the first stop coming from the client, it provides a unique user session back to that calling client. In this application, once a user has logged in (received an AuctionSession object), they can begin sending requests to be consumed by the backend service.

When a logged in user calls “create auction” from index.html, the following happens:

1 Index calls /auction-session with initial auction parameters proxying through the logged in auction session
2 The Auction Session passes the parameters to the Auction Manager’s create method
3 Auction Manager’s create method receives a uniqueID from the IdentityManager
4 An Auction object is created with the given uniqueID and indexed

5 Complete is called on the Auction’s Result, which propagates back to Index.html which is updated to reflect the new change

Methods within the Auction Example function in this manner, with a multitude of services being utilized in a non-blocking fashion. The services perform non-blocking batched updates, allowing for no inherent bottlenecks even as data is loaded and saved.

The ***AuctionSessionImpl*** takes advantage of Baratine’s CDI capabilities and injects services to be used from the other classes. Once these services have been injected, they can be called reliably within the Baratine environment:

```
@Inject
private ServiceManager _manager;

@Inject @Lookup("pod://user/user")
private UserManager _users;

@Inject @Lookup("pod://user/user")
private ServiceRef _usersServiceRef;

@Inject @Lookup("pod://auction/auction")
private AuctionManager _auctions;

@Inject @Lookup("pod://auction/auction")
private ServiceRef _auctionsServiceRef;
```

The ***UserImpl*** class is responsible for the business logic of a typical Auction user. It contains the following methods:

```
public void create(String userName, String password, Result<String> userId)
private boolean setUser(Cursor c)
public void authenticate(String password, Result<Boolean> result)
public void getUserData(Result<UserDataPublic> user)
```

As mentioned previously, Baratine Services are responsible for their data. In this case, each UserImpl object will be stored under a

pod://user/{id}

which is automatically sharded to a particular node in a cluster. This makes creating a user simple, we utilize a private instance of Baratine’s database and store created users into it.

```
@Override
@Modify
public void create(String userName, String password, Result<String> userId)
{
 _user = new UserDataPublic(_id, userName, digest(password));

 _db.exec("insert into users(id, name, value) values(?,?,?)",
          user.Id.from(o-> _id),
          _id,
          _user.getName(),
          _user);
 }
```

This store is now responsible for all data created by this ***UserImpl*** Service on that particular id.

Within the Auction example, Users are handled by a UserService named ***UserManagerImpl***. This separation from the User class itself, allows Baratine to continuously provide new users with unique ID's without any impact to the Auction Service itself. In this manner, Baratine allows you to offload portions of your business logic into a set of as many services that are needed.

========================
Part Four: ***<u>Loading & Storing</u>***
========================

A service handles loading and storing on two annotated methods: @OnSave & @OnLoad. The ***UserImpl*** Service places the annotations in the following fashion:

```
@OnLoad
public void load(Result<Boolean>
result)

@OnSave
public void save(Result<Boolean>
result)
```

@***OnLoad*** is called when the service is first instantiated to allow it to load its data. It attempts to load its previous state from the database from the {id} the service was passed.

@***OnSave*** is called when a service should save (persist) its data to disk. Saving can be triggered in two ways:

If a Service is "journaled", the service will call @OnSave once per journal rollover (in a batch).
If a Service is not journaled, the service will call @OnSave any time a method annotated with @Modify is called

@Modify signals to Baratine that a method is modifying service state and should be saved. Every time a method with ***@Modify*** is called, it is immediately proceeded by an ***OnSave***. When the ***OnSave*** is called, the inbox to that service is flushed and data is persisted.

By utilizing these two methods, developers can pre-load their objects with data previously saved or from another module entirely. In this particular example, OnLoad is used to pull previous users from the database, and OnSave is subsequently used to store the created users in Baratine's internal database. Because there is no persistence layer between Baratine and it's storage, the POJO methods become the schema for the data saved, and all of this can be managed within a single class.

=====================

Part Five: ***Configuration***

=====================

Baratine configuration is handled through the notion of named “pods”. The following two configuration files are used for the Auction Service Example:

auction.cf:

```
pod auction {
  archive "/usr/lib/pods/auction.bar";
 }

pod user {
  archive "/usr/lib/pods/auction.bar";
 }

pod web {
  archive "/usr/lib/pods/auction.bar";

  web "/" "web";
 }
```

conf.cf

```
server_port = 8085;

cluster {
  server port = $server_port;
  logger finest;
  logger finer "examples";
  logger off "com.caucho.amp.proxy";
  logger off "com.caucho.kraken.table";
 }
```

Applications deploy to Baratine pods, a virtual cluster of server nodes. Although the deploy command can automatically create a configuration file, developers can upload their own configured files.

The configuration file for a deployed service contains the following:

- pod configuration for the application
- code archive for the application

The pod can have multiple servers, specified by the type (solo, pair, triad, cluster), and can allocate servers automatically or statically, based on the configuration in / config/pods.

Client applications access services using the “pod:” scheme if inside Baratine, or with “/s/pod” if using HTTP.

================

Part Five: Summary

================

The Auction Service is a fully functioning web service for Baratine. The programming model within Baratine is unique, and these files and setup serve the purpose of better informing developers who are looking to create high performing microservices. With Baratine’s programming model many of the technical aspects that are inherently difficult to manage in such an environment (synchronization, sharding, reliability, low-latency) are abstracted away and developers can write straightforward code.

What’s more is that, as we have shown, an API can go from interface to fully functioning implementation all served by a Baratine backend and simple frontend framework (In this case Angular.js). Testing within an asynchronous environment is also easier to mange because services can be easily replicated into blocking calls to ensure the logic within a program. For more information head over to http:// baratine.io and get started today!