Abstract: Asynchronous and non-blocking execution is a highly coveted attribute of high volume web services. Server side web services strive to make contractual agreements upon interface based APIs to other services, assuming their environments to be highly-available, recoverable, & loosely-coupled. However these attributes are often in direct disagreement with an APIs implementation given: (1) the underlying multi-threaded libraries an API must rely upon and (2) the outside data source whose assets must be operated upon before an API can complete. This hindrance often results in contractual agreements across services tying down & blocking the multithreaded resources of a given architecture. This bottleneck is only half solved by frameworks where asynchronous request processing is paired with either NoSQL or SQL databases whose data access must block an incoming thread in order to make durable mutations on an asset. Techniques such as caching & sharding in-memory data structures alleviate the bottleneck for modifications on data at the cost of added complexity; exposing a view of the data that cannot be atomically mutated, results in complex and eventually consistent architectures.

In this paper we introduce Baratine, an asynchronous web framework with an improved thread and data execution model. By only accessing data for a particular service on a single thread, Baratine provides an authoritative owner of data services to be operated on continuously, solving both (1) & (2) sources of the aforementioned bottleneck. These services can persist no state operating entirely in-memory, persist to Baratine's internal reactive document-style database or choose to persist to a traditional outside datasource. Baratine's internals provide high availability, recoverability, and thread context isolation to each service within its environment. Baratine puts forth an API level abstraction for developing services that match the POJO style of Java EE & Spring applications prevalent in today's architecture.

Introduction

Application requirements have changed dramatically in recent years. In the mid 2000s, large applications had tens of servers, seconds of response time, hours of offline maintenance and gigabytes of data. Today applications are deployed on everything from mobile devices to cloud-based clusters running thousands of multi-core processors. Clients expect millisecond response times, 100% uptime, and data is measured in petabytes. This change, the demand of Big Data & IoT evolution, in turn explains the meteoric rise of NoSQL databases, Redis as a KV store, and cached data grids as in-memory storage structures as they have provided a faster location for applications to store and retrieve data for their operations.

Along the lines of faster execution, innovative single-threaded approaches towards solving the blocking nature of multithreaded applications have also been considered. COST, or the configuration that outperforms a single thread, has revealed that "most of the published work on big data systems fetishizes scalability as the most important feature of a distributed data processing platform without directly evaluating their absolute performance against reasonable benchmarks". Thus horizontally scaling a system does not provide linear performance, but rather parallelizes the overheads that these systems themselves introduce.

Some single-threaded solutions, such as Node.js and Vert.x, have been able to achieve greater compute efficiency by implementing on asynchronous processing. However, while these frameworks provide high performance for single jobs, their lack of a durable thread independent data store & obtuse programming model prevent them from achieving a wider adoption rate by the programming community.

As these models attempt to implement what SOA initially defined and Microservices has redefined, their widespread adoption is heavily dependent on matching the ubiquitous Spring & Java EE style of programming. POJO-style abstraction, or more specifically interface-based APIs, are the dominant style of application development.

Baratine provides the programming model that has been missing; a threading and data execution path that allows developers to leverage interface-based programming while executing asynchronously. Thus Baratine's goal from since inception of development has been to provide a programming model that does not need to rely on a database for every read or update request. Baratine does this by keeping data in-memory, mutating data based on single-threaded connections, and persisting to its internal reactive style database at set intervals.

1 Baratine's Design

Baratine has a service-oriented architecture. Within a service, Baratine guarantees the following attributes to be true:

- Strong encapsulation boundary for code, objects, and thread context
- Single-threaded execution for high-performance
- Continuation-style asynchronous programming

These principles are incorporated into a Service, a POJO class annotated with @Service, to allow developers to build applications as a set of services. As a note, asynchronous programming is not a requirement, but rather encouraged for performance in Services.

The second cornerstone of Baratine's design is the internal reactive database, Kraken. From a developer's standpoint the database is

transparent. So much so that neither database schema nor database configuration is necessary for data driven services in Baratine. Instead, persistence is managed by implementing Baratine's Vault interface or by annotating methods with an @Modify annotation.

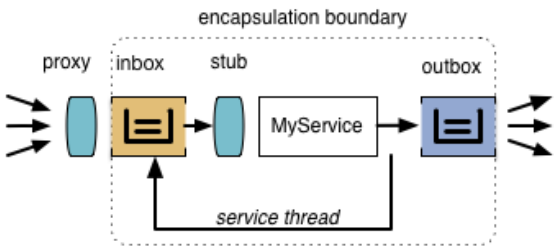
Baratine then allows each service to operate as the authoritative owner of the data it mutates. Because service calls are answered in batches on a single thread, it is not necessary to code explicit synchronized code blocks even in highly concurrent environments.

These two major internal components combine to make Baratine a fully reactive framework with no outside data source dependencies.

1.1 Service

The building block within Baratine is a @Service. A Service environment in Baratine guarantees the following:

- Strict encapsulation boundary
- An application service implementation
- An optional application API
- Thread safe concurrent proxy
- Service thread for synchronization, cpu affinity, and isolation
- Inbox queue to sequence and synchronize calls
- Proxy and stub to marshal calls to messages and messages to calls
- Outbox for outgoing messages.



Typically, only the proxy API and service implementation are visible to the application. The inbox queue, service thread, proxies, and stubs are all applied invisibly by Baratine. From an application perspective, the service looks like a single-threaded implementation called by a proxy using continuation-style methods.

Hello Service:

```
public class Hello
{
  public void hello(Result<String> result)
  {
    result.ok("hello");
  }
}
```

At the minimum, a service consists of an inbox, a POJO to process requests, and an outbox. The inbox is a ring-buffer queue that allows the service to process requests serially and in batches. A single thread retrieves requests from the inbox and invokes it upon the POJO's methods. The result is serialized to the outbox to be sent out in batches to recipients.

While this asynchronous single-thread model is highly encouraged for use, Baratine does allow for multi-threaded services through *worker* threads. A service becomes multi-threaded when annotated with @Workers on the class. @Workers accepts an argument designating the maximum number of threads for the service. This annotation provides a bridge between asynchronous services in Baratine and legacy systems they might communicate with.

1.2 Asynchronous single-threaded Services

Baratine's *Result* is a callback that allows methods to be asynchronously called. Instead

of returning when the method completes, a method issues a callback in the form of a Baratine Result. This Result is executed when a response comes back to the caller and allows methods to follow normal Java syntax with just a simple additional parameter.

When calling a method with a Result callback, the caller does not maintain state between a call and its Result because the Result is sent along with the call. The response includes the caller's Result, allowing the caller to execute on this response after the response has been examined. This design allows for high-performance RPC and is core to Baratine's model.

When using a Result, methods do not complete or use a *return* call. Instead, Results offer two primary ways of method completion: ok() and fail(). ok() is for a normal return. fail() is for exceptions.

```
public interface Result<X>
{
  default void ok(X value) { ... }
  default void fail(Throwable exn) { ... }

  void handle(X value, Throwable exn);
  ...
}
```

A Result may be completed right away or it may be passed to another service for it to be completed there. It can be in any position in a method's argument list granted that only one Result is used in a method. You cannot have multiple Results on a method.

1.3 Reactive Database Persistence

Baratine comes with a fully asynchronous document-style database named Kraken. Kraken combines with other Baratine components to provide *operational data* within an application.

Services maintain and operate on their data completely in-memory while Kraken handles the necessary persistence while Baratine reclaims memory through an LRU eviction model & GC.

Service inboxes are backed by a journal that is recoverable in the event of network or machine failure, allowing for atomic operations. This definition of data does not conform with traditional architecture where data is abstracted into an outside data entity and accessed sequentially through multiple threads, locking and blocking along the way. It is also what we define as *reactive database persistence*, because no locking/blocking is needed to operate durably on data.

Reactive database persistence is made possible in Baratine because only a single thread is ever mutating data and subsequently passing Results back to the callers. This unification of thread and data upon a single thread eliminates the need for synchronized code blocks even in highly concurrent environments.

Baratine offers fine grained control into the life-cycle of services through the following annotations:

@OnInit : called to initialize the service
@OnActive : called when service is ready
@OnDestroy: called when service is going away
@OnLoad : called to load initial data into the service
@OnSave : called when @Modify methods have been called at least once

Alternatively, Baratine provides driver services for traditional databases, both JPA & JDBC implementations, that can be used with a service at the cost of performance.

2 Async Web Server

Baratine's web server is service-based, using Result to manage asynchronous calls. Applications can be written as single-threaded services, but should not block and instead use the Result to continue the request on a completion.

Baratine applications are a combination of non-blocking services, and blocking gateways to existing Java libraries. When the web service calls other services, such as an authentication service or a database service or a gateway service to a Java library, the continuation result allows for efficient reuse of the web-service's thread while the authentication or database is doing its work.

The web server supports two styles of configuration:

1. A Java DSL programmatic configuration for straightforward services
2. An annotation-based configuration for services using dependency injection

It also supports two styles of output generation:

1. View style, where the service returns a result object and a view resolver such as a JSON writer renders the output.
2. Direct output with writers and output streams.

2.1 RequestWeb

RequestWeb provides standard HTTP access, as well as access to application sessions and services. RequestWeb can be used to quickly code async REST services in a community standard format.

HTTP request and network information is available within the following methods:

```
public interface RequestWeb extends
Result<Object>
{
  String protocol();        // http or https
  String version();         // HTTP/1.1

  String method();          // GET, POST, etc.

  String uri();             // full HTTP URI
  String path();            // part of URI matching
the path pattern
  String pathInfo();        // trailing path for /*
patterns
  String path(String id);   // named path
pattern /{id}
  Map<String,String> pathMap();

  String query();           // query string
  String query(String key); // query param
  MultiMap<String,String> queryMap();

  String header(String key);
  String cookie(String key);

  String host();
  int port();

  String ip();
  InetSocketAddress ipRemote();
  InetSocketAddress ipLocal();

  SecureWeb secure();

  ...
}
```

As with using the explicit *Result*, requests will end once .ok or .fail has been called.

2.2 Templates & Views

Baratine Services are the authoritative view of their data, because of this, it naturally fits that a service should have ways to generate data as formatted output. There are two ways services can generate formatted output:

1. View rendering
2. Direct writing to an OutputStream or Writer.

Views can use standard formats like JSON and rendering engines like Mustache or Freemarker.

A web service returns an object in its request.ok method, sending the object to the view resolver. Based on the object, the view resolver chooses a view renderer. The matching view writes the object to the output using the output or writer methods of RequestWeb.

The View type is a standard view object that has a view template name and a map of values.

2.3 Session

Most clients interact with web services through the use of sessions. Sessions provide each client with an isolated and custom interaction. In Baratine, sessions are handled by session services. A session service instance is tied to a user as determined by the session cookie. To define a session service, use @Session annotation:

```
@Session
public class MySession {
  @Id // injects session cookie ID into this
field
  private String id;

  private String user;

  @Get("/login/{user}")
  public void login(@Path("user") String user,
RequestWeb request) {
    this.user = user;

    request.ok("user logged in as: " + user );
  }
  public void getUser(Result<String> result) {
    result.ok(user);
  }
}
```

Sessions encapsulate data with a vault/asset service. Because sessions are services, they use the continuation Result instead of method returns.

Alternatively to annotating a class, the session() method in RequestWeb with the session's API or the service address can provide a session instance.

2.4 Vault Services

Baratine handles CRUD data requests through the use of internal vault services. A vault is a collections of assets saved in a database. For example, a book vault's assets would be its books. Vault assets operate in-memory using the vault's thread and inbox.

Assets are the in-memory model for persistent objects and have the following attributes:

- Assets are encapsulated. Application code operates on fields without locking or transactions.
- Assets need a single writer to ensure atomic updates.
- The single writer is a Baratine service.
- Because of the single owning thread, asset services must be non-blocking for performance.

A book vault has an vault interface and a book asset implementation. The vault and its assets are a single Baratine Service.

The vault creates, finds and deletes assets. It acts across all assets in the vault.

The asset contains application logic, acting on the asset as an encapsulated object. Internally, asset loads and saves itself, because of the

single-writer principle. Single-writer is required to support atomic and consistent updates. Without it, data would be exposed to other request mutations before being saved.

Application logic operates on the asset as an in-memory encapsulated object. Methods that update the data need a @Modify annotation to tell the asset to save its state.

IDs provide a way for assets to be uniquely saved in Baratine's database. IdAsset is a built-in 64-bit identifier, which displays as a base-64 string, and is designed to be unique across a Baratine cluster.

2.5 WebSockets

Websockets provide full-duplex communication channels across a single TCP connection. This is a perfect fit for Baratine's asynchronous architecture as traditional request/response architecture inefficiently blocks resources. The unification of Websockets and Baratine mean clients and servers are operating in a continuous harmony.

WebSockets are implemented with the ServiceWebSocket interface
or you can upgrade requests to WebSockets by calling RequestWeb.upgrade() and passing in a ServiceWebSocket instance:

```
public class EchoWebSocket implements
ServiceWebSocket<String,String> {
  public void open(WebSocket<S>
webSocket) throws Exception {
    System.out.println("opened websocket
connection");
  }

  public void next(String value,
WebSocket<String> webSocket) throws
Exception {
    webSocket.next("echoing: " + value);
  }
```

```
public class MyWebService {
  @WebSocketPath("/echo")
  public void doUpgradePath(RequestWeb
request) {
    request.upgrade(new EchoWebSocket());
  }
```

2.6 Pipes

WebSockets and Baratine match so well together that it introduces a problem for processing business logic. More precisely, messages can pass through a network and Baratine faster than they can be processed. Pipes are therefore a secondary messaging system when flow control is required.

A websocket clients for a chat system might freeze its network connection, but the system itself must not freeze. Flow control lets the producer avoid blocking when the consumer is blocked.

Pipes are unidirectional messaging between two services. Unlike service calls, they use application messages and require a subscription call to setup.

Flow control is managed by a credit sequence. The consumer adds credits, which the producer can use to send messages. For convenience, a prefetch can issue credits automatically for simpler applications.

Pipes are designed to resemble the JDK 9 Flow API and Reactive Streams. Because of this design, Pub/Sub services are easy to setup and revolve around the following:

1. A PipeBroker
2. A Pipe Subscriber
3. A Pipe Publisher

2.7 Injection

Baratine is a service oriented architecture, where services combine to create entire

loosely-coupled applications. However, services need to remain loosely-coupled while interacting with each other. Baratine provides the @Inject annotation for dependency injection to combine services without tying up the configuration. The configuration can occur cleanly during initialization and the lookup happens when you need it.

The two phases are registration and injection. For example, a book store might use a search service to find books. To get a copy of the SearchService, it would use injection:

```
public class MyBookStore
{
  @Inject SearchService _search;

  ...
  public void findBookByTitle(String title,
Result<Book> result)
  {
    _search.find("title", title, result.of());
  }
}
```

Before the search service can be injected, it needs to be registered. In Baratine, the registration can use the Web.bean() method like the following:

```
public class MyBookMain
{
  public static void main(String []args)
  {
    Web.bean(new
SearchServiceImpl("localhost", 6920))
       .to(SearchService.class);

    Web.include(MyRestBookStore.class);
    Web.start(args);
  }
}
```

The bean method registers an instance (or a class or producer), which is then bound to an API. In this case, we register a search service client that connects to localhost:6920 and attach it to SearchService. Any code that needs the search service, such as our bookstore above, can inject it using the interface.

Baratine's injection defaults to a singleton service. If multiple services inject the bean, only one copy of the bean is created. This default is different from some other injection, because Baratine is designed around services, and the other injections are designed around request objects.

Beans can be registered with the injection system with several options:

- A @Bean annotated producing class, which provides the instance.
- An instance, useful for programmatic configuration.
- A class, used with other injection.
- A provider, which programmatically instantiates the instance.

2.8 Event Services

Event-sourcing is a relatively new concept that evolves around two primary concepts:

1. Changes to application state are stored as a sequence of events
2. An event log can be used to reconstruct past state changes

To fit this specific use case, Baratine provides event services. The event service is an API-based publish/subscribe broker. Publishers send events using application APIs. Subscribers receive events with the same API.

When a subscriber wants to receive event updates from a publisher, but wants a loosely-

coupled relationship, developers can use the event service.

The address for an event node is application-defined. By convention, the classname of the API can be used as the node name.

3 Service Discovery & Paths

Service discovery within Baratine is handled internally. Baratines internal provide a centralized service for maintaining configuration information, naming, and providing group services.

Developers can code paths to these services, and Baratine will handle their discovery transparently to the developer. This means that services are never hard-coded and instead dynamically maintained by Baratine's internals.

Paths to Baratine methods is handled by providing named URLs:

```
@Get("/hello")
public void doHello(String name,
RequestWeb request) {
  request.ok("hello ");
}
```

If no path name is provided, method names are used as the default route.

4 Clients

Outside clients can communicate with Baratine services through either normal HTTP or Websocket connections.

This allows for clients to be written in any language that can consume JSON.

5 Testing

Baratine services are designed to be isolated from each other, which means they fit naturally into independent unit tests. When possible, Baratine services should be designed as leaf services with few dependencies, which makes them easier to test. In this way, Baratine services can be tested as an ordered sequences, checking their output within each interval.

The JUnit tests can focus on this sequential logic for the service because Baratine's ensures in-order message queues. If your application has a buggy or unusual timing sequence, the JUnit tests can replicate it precisely.

6 Performance

In testing, Baratine's continuation/async performance is consistently better than async/future, especially under load.

Under light load, async performance is around 9M method calls/sec/inst while future performance is 4M calls/sec/inst, which is reasonably close. As the load increases future performance drops dramatically to 0.5M calls/sec/inst. Async performance drops as well but stays over 2.2M calls/sec/inst.

The extra cost of the blocking/future calls is primarily wake/unpark cost, which is very expensive. The blocking/future calls have a fixed blocking thread per client, while the continuation/async calls release their thread to the thread pool.

For the future call, "clients" is the number of threads, while for the continuation call, "clients" is the number of in-flight messages with fewer active, pooled threads.

	solo	2 clients	3 clients	4 clients
Pipe	39M	n/a	n/a	n/a
Send	18M	17M	15M	14M
Query Batch =1	9.2M	3.0M	2.2M	2.3M
Query Batch =4	7.4M	3.4M	4.1M	3.8M
Sync Query	4.1M	1.9M	0.65M	0.48M

Pipe has the best solo performance because it avoids blocking with its credit flow control, and because it's optimized for a single producer and consumer.

While sending from a service has lower performance than pipe, it supports multiple producers. When its inbox queue is filled, it will force the producers to block, which incurs some overhead, but more importantly freezes the producers, preventing them from processing more messages while the queue is full.

Continuation/async calls have about half the performance of a solo send, because a call is two send messages: call and reply. With multiple clients, continuation performance drops, because a reply must wake the caller's service if it's asleep.

Batched continuation calls improve performance under load, nearly double the unbatched performance. Continuation calls can batch requests because the calling thread sends its next request while the previous call is in flight.

While sync/future calls are nearly as fast under light load, they quickly drop in performance under heavy load. Unlike continuation calls, they can't batch requests because the calling thread is blocked until the call completes.

7 Conclusion

Baratine offers significant improvements over many of the issues that plague traditional architecture and prevent agile development. By employing an underlying programming model based upon single-threaded connections, Baratine's model marries operational asynchronous data with asynchronous request processing.

These operational services combine together to build loosely-coupled reactive services that make only agreements upon APIs in an isolated thread context. Thus applications that mimic CQRS, Event-sourcing, Lamba-Architecture, or many of the other models that fit into reactive programming can be implemented in Baratine.

Baratine's approach to server side development and commitment to POJO-style abstraction provides a bridge between fully asynchronous applications and legacy applications still in production. Developers ca either code proxy-based asynchronous APIs into their current legacy systems or rewrite portions of their current API as standalone operational services in Baratine. With Baratine's internals under the hood, areas such as scalability, performance, and concurrency relegated to an afterthought as developers can focus on coding clean asynchronous web services.