Event Sourcing - Disruptor Pattern

> Event Sourcing
> Async Event Processing
> Producer/Consumer – Disruptor Pattern

Abstract

Layered Architecture, Microkernel Architecture, Microservices Architecture and Event-Driven (Event-Sourcing) Architecture each solve specific challenges.

This document outlines a demo application that examines Event-Driven Architecture.

The central force in an Event-Driven Architecture is Event-Sourcing. Event-Sourcing requires a temporal *Message Broker*.

The absence of blocking data structures is what allows an Event-Sourcing platform to handle a massive number of client requests w/ one thread.

A Message ßroker can be implemented by adopting ideas found in the *Disruptor Design Pattern*, which in turn leverages a data structure variously called a *Ring Buffer* or a *Circular Buffer*.

Mutable/immutable *Events* can be used in a *Producer/Consumer* fashion, but the important idea is that the core event-processing implementation employs a single thread (no locks, no synchronization).

The Disruptor concept has its roots in the LMAX trading Platform (UK) which was open-sourced by LMAX circa 2012 (it is a battle-tested idea).

Apache *log4j2* adopts the Disruptor pattern, via its *Disruptor.jar*, which enables log4j2 to offer an asynchronous logging modality.

If combined with a *Correlation ID* (*CID*), asynchronous log4j2 can act as a *Replayable Journal* for an Event-Sourcing platform.

ßackground

A Ring Buffer can be used to provide fast, single-threaded *Event Processing*.

In order to prevent race conditions (thread safety) when using a single thread of execution, memory blocks must be aligned to assure that False Sharing does not occur in CPU Cache Lines (complicated by the fact that modern computers have multiple CPUs, each containing multi-cores.

At the lowest level, this means that memory must be "padded" to assure that word-tearing never happens.

The conceptualization was implemented by LMAX*, a European commercial trading platform.

Open-sourced circa 2012, LMAX's underlying *Disruptor* technology is used by log4j2† for its async logging feature/function via a binary distribution:

```
Disruptor.jar
```

Here are the Maven coordinates for log4j2 async enablement‡:

```
<dependency>
  <groupId>com.lmax</groupId>
  <artifactId>disruptor</artifactId>
  <version>3.2.1</version>
</dependency>
```

Similarly, here are the Maven coordinates which are needed by this (Exchanger) app's *Disruptor/RingBuffer* async enablement§:

```
<dependencies>
  <dependency>
    <groupId>com.googlecode.disruptor</groupId>
    <artifactId>disruptor</artifactId>
    <version>2.10.4</version>
  </dependency>
```

Design, high-level

In each of the three interrelated apps, the `publish` (ßash script) enables the sharing of the functionality within the three interdependent modules. Sharing is accomplished via Maven `<dependency/>` elements.

The Event is presented to the Exchanger – by the action of data having been made accessible to an EventProducer callback method named `publisherCallback()`.

The `publisherCallback()` decides what kind/volume of data to insert introduce into the Exchanger's Ring Buffer.

The Exchanger polls the `EventProducer.publisherCallback()` method, using a `while()` loop.

The demo application has two instances of EventProducer, and a single instance of EventConsumer. This proves the efficacy of the Exchanger's Ring Buffer.

When Exchanger the detects that an Event has been added to its (internal) Ring Buffer, its sole (internal) `EventHandler` instance – of type `EventHandler<Event>` – responds by calling the sole method in EventConsumer:

```
take(Event)
```

This completes one event-sourcing cycle.

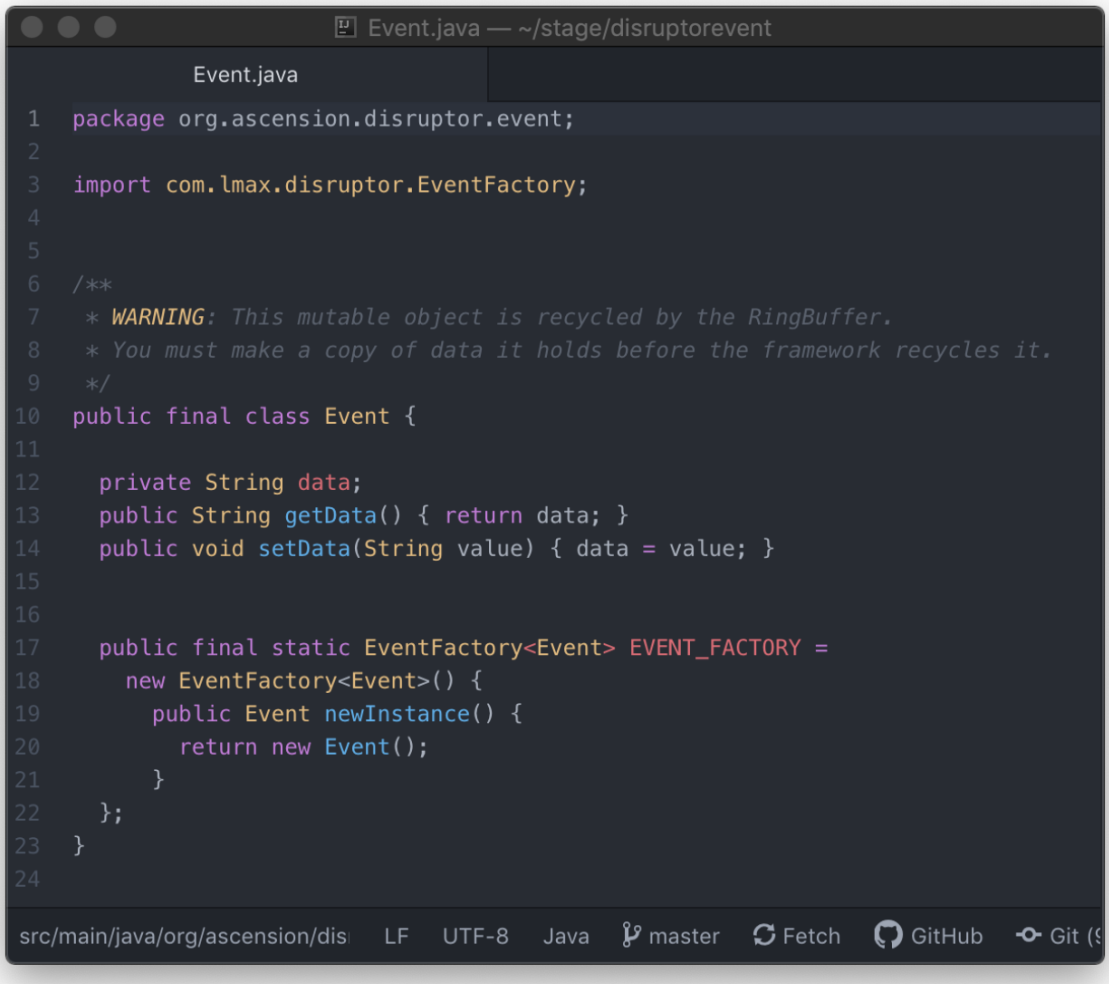
Event, detailed analysis

On lines 17-22, the `org.ascension.disruptor.event.Event.java` class wraps the canonical `EventFactory` class.

Within the `Event` class the inline `EventFactory` constructor is assigned to a `public static final` reference (of `EventFactory`).

This construct (routine) is called indirectly within the same class that constructs the Disruptor. In this application that class is:

`org.ascension.disruptor.event.engine.Exchanger.java`



Producer, detailed analysis

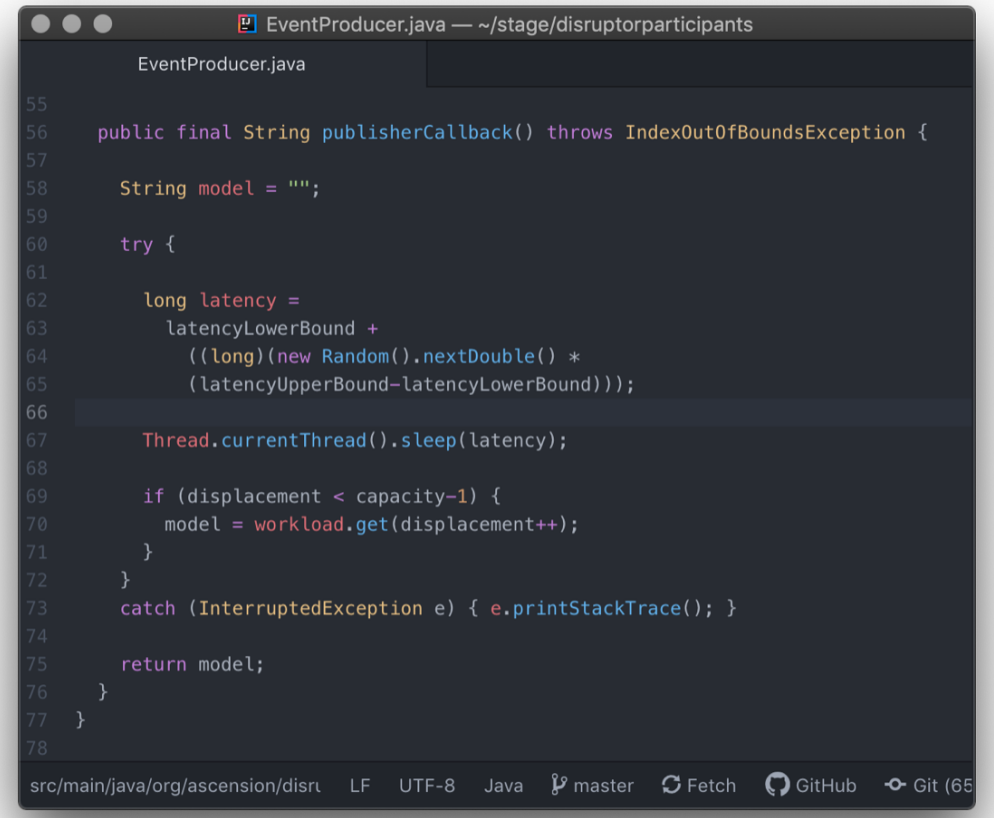
In the demo app a *Producer*, of type:

`org.ascension.disruptor.event.producer.EventProducer.java`

synthesizes a set of arbitrary `String` values into a model that is then published (in batch mode) as Events to the *RingBuffer* (the `Exchanger.java` class).

A callback method – `publisherCallback()` – introduces artificial latency into the publication. The latency simulates unpredictability (asynchronicity) into the response to the `Exchanger.java` instance's polling.

This arrangement proves the efficacy of the *RingBuffer* that is encapsulated within the Exchanger – there are two (2) *Producers* and Events from each interleave with one another||.



Consumer, detailed analysis

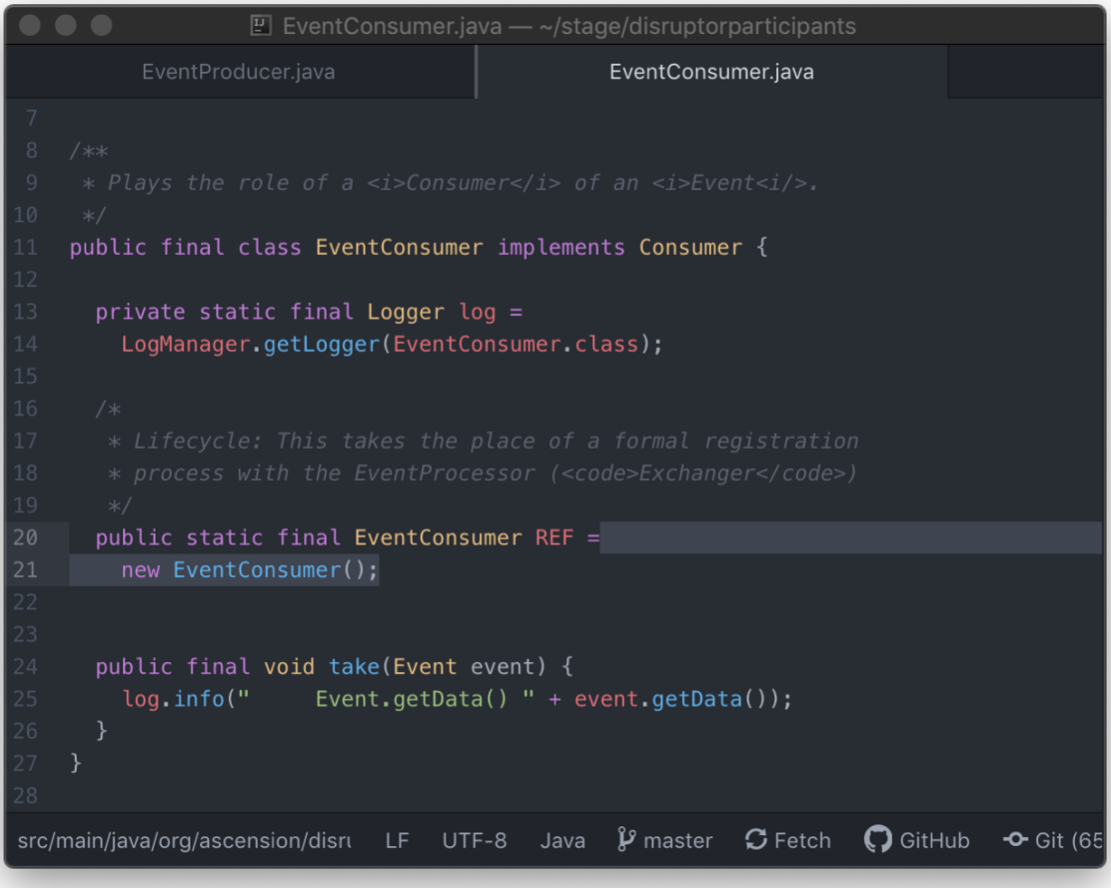
The EventConsumer simply grabs Event data, via its single (*Interface*) method, with the signature:

```
take(Event)
```

It acquires Event data on demand.

The "demand" is a call which originates with the canonical `EventHandler.onEvent(...)` method being triggered, then calling the `public EventConsumer.take()` method (see Lines 24-6).

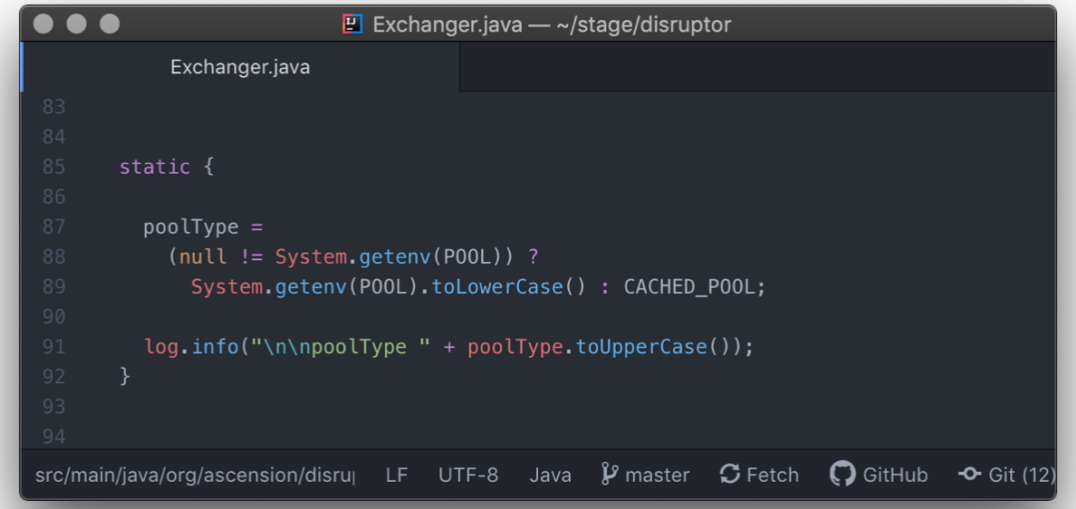
The canonical `EventHandler.onEvent(...)` method exists within the Exchanger instance and listens/responds to *RingBuffer* publication events.



Exchanger, detailed analysis

The Exchanger class encapsulates the implementation of the core *Event-Sourcing* elements.

The Exchanger class also observes the 12-Factor application credo when it allows the *Producer* thread pool model to be configured via the value of an environment variable:



When run from within a shell, the Exchanger component's thread pool model can be specified – issue;

```
export POOL=CACHED_POOL
```

While the Exchanger component defaults to a thread pool value of `CACHED_POOL`, a CLI override to `SINGLE_POOL` can be done.

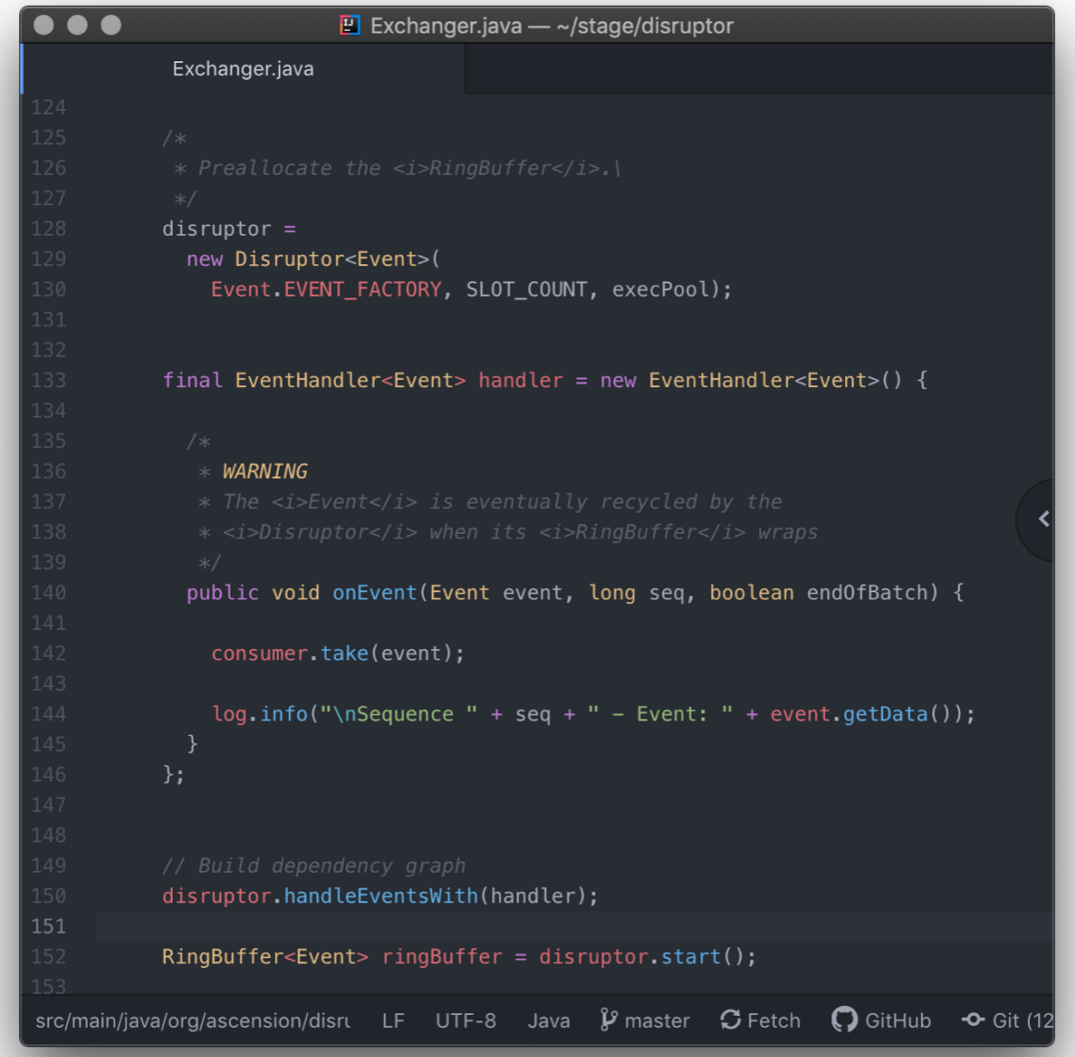
Semantics:

```
source ok C
source ok S
```

The arguments to the `ok` script are *case-insensitive*.

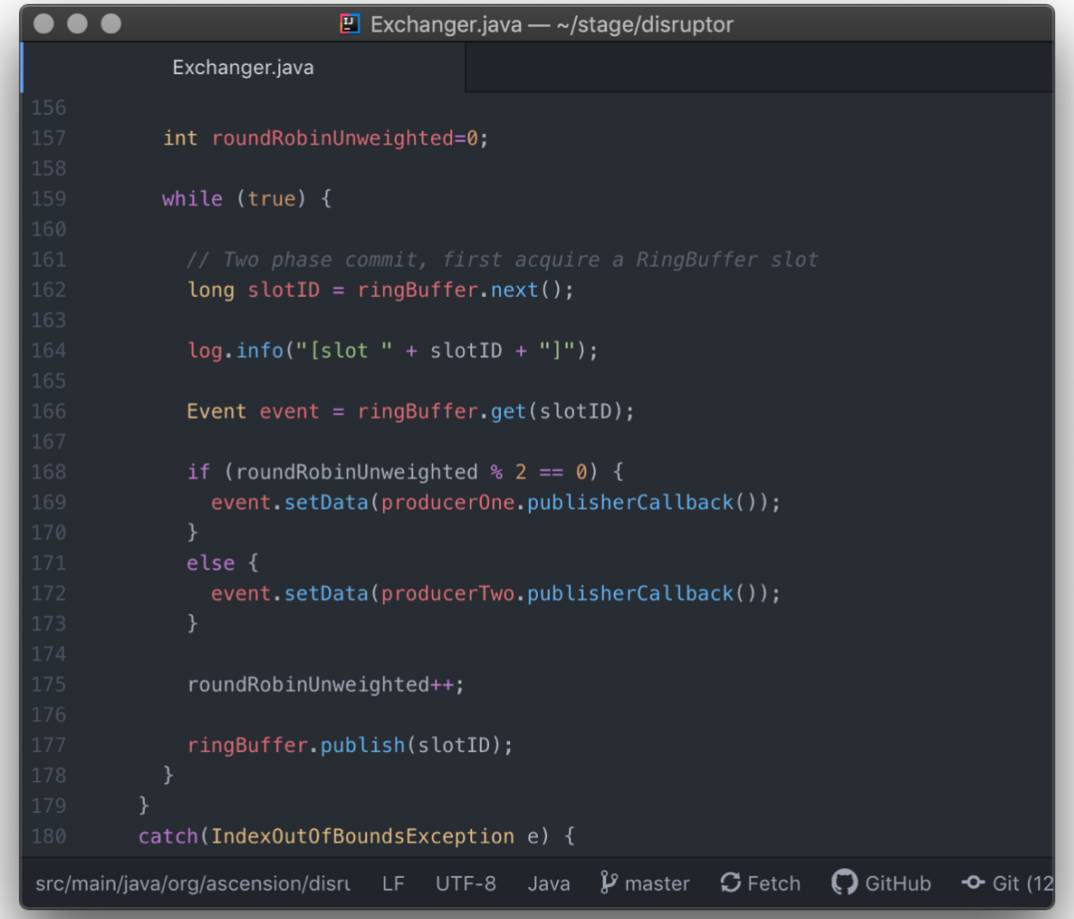
A value containing the letter C – `CACHED_POOL`
A value containing the letter S – `SINGLE_POOL`

The arguments to the `ok` script are *case-insensitive*.



The section of Exchanger code (above) registers a new instance of the (canonical) `EventHandler` (lines 133-46) with a new (canonical) `Disruptor` (lines 128-30) to create (a canonical) `RingBuffer` instance (line 152).

This operation also pre-allocates the memory for the slots in the `RingBuffer` (line 150).



In a polling loop, reference the next available *RingBuffer* slot (line 162), to access an Event, line 166.

Based on a simplistic *Round-Robin* algorithm (line 168, 175), nominate one of two *Producers* from which to get its publishable data (lines 168-73).

Publish the (*Producer's*) payload to the *RingBuffer* slot (line 177).

Application(s), wiring together of

The three components in the demo app are independent, but interdependent.

The wiring of the three components – at a binary level – is accomplished using standard Maven dependencies.

The mappings are shown immediately below.

```
<groupId>org.ascension</groupId>
<artifactId>disruptorevent</artifactId>
<packaging>jar</packaging>
<version>1.0.0-SNAPSHOT</version>
```

The Event Coordinates

```
<groupId>org.ascension</groupId>
<artifactId>disruptorparticipants</artifactId>
<packaging>jar</packaging>
<version>1.0.0-SNAPSHOT</version>
```

The Participant Coordinates

```
<groupId>org.ascension</groupId>
<artifactId>disruptorapp</artifactId>
<packaging>jar</packaging>
<version>1.0.0-SNAPSHOT</version>
```

```
  <dependency>
    <groupId>org.ascension</groupId>
    <artifactId>disruptorevent</artifactId>
    <version>1.0.0-SNAPSHOT</version>
  </dependency>

  <dependency>
    <groupId>org.ascension</groupId>
    <artifactId>disruptorparticipants</artifactId>
    <version>1.0.0-SNAPSHOT</version>
  </dependency>
```

The Exchanger Coordinates, followed by its two (2) Dependency Coordinates

---

Application(s) Lifecycle, scripts

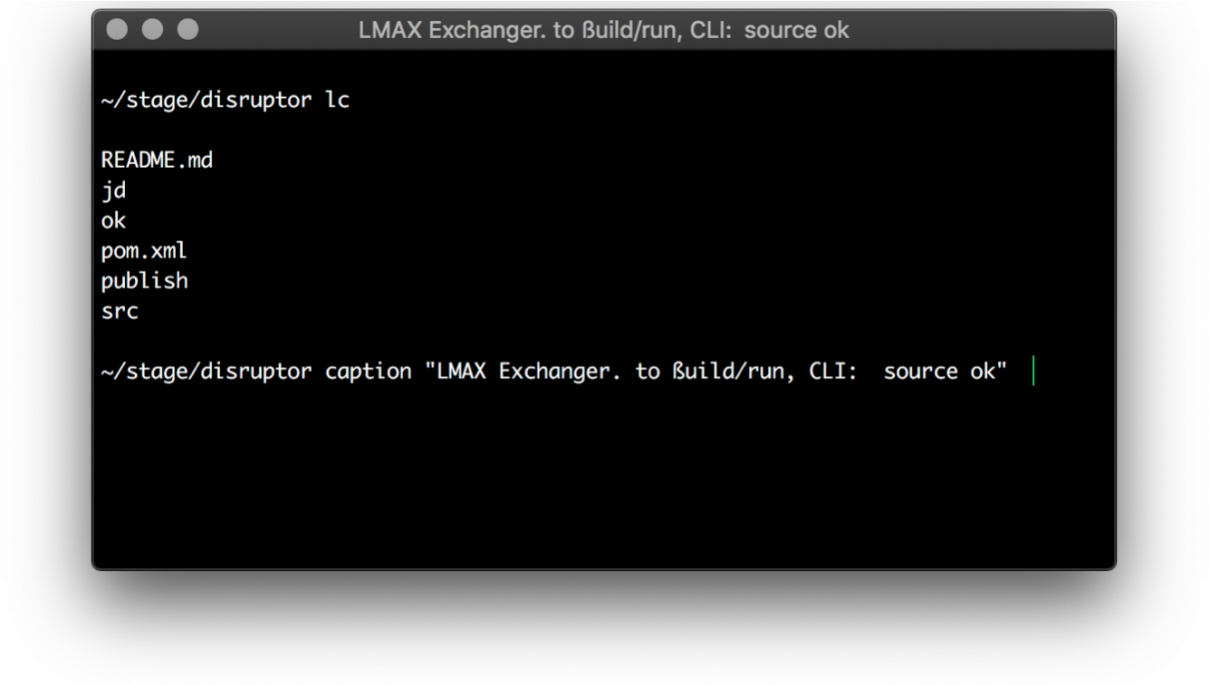
The three components/parts of the Event-Sourcing demo app all adopt the same build approach.

The Disruptor component, housing the `Exchanger.java` class, is the driver – it has a `main()` method. It has dependencies on both of the other components.

The other two application components – Participants, Event – each have an additional script that "publishes" their Maven coordinates to the local repo, making them available to any other components (such as the Disruptor/Exchanger components). This script (w/o an extension) is named: `publish`

Each of the app components' build scripts (w/o extension) are named: `ok`

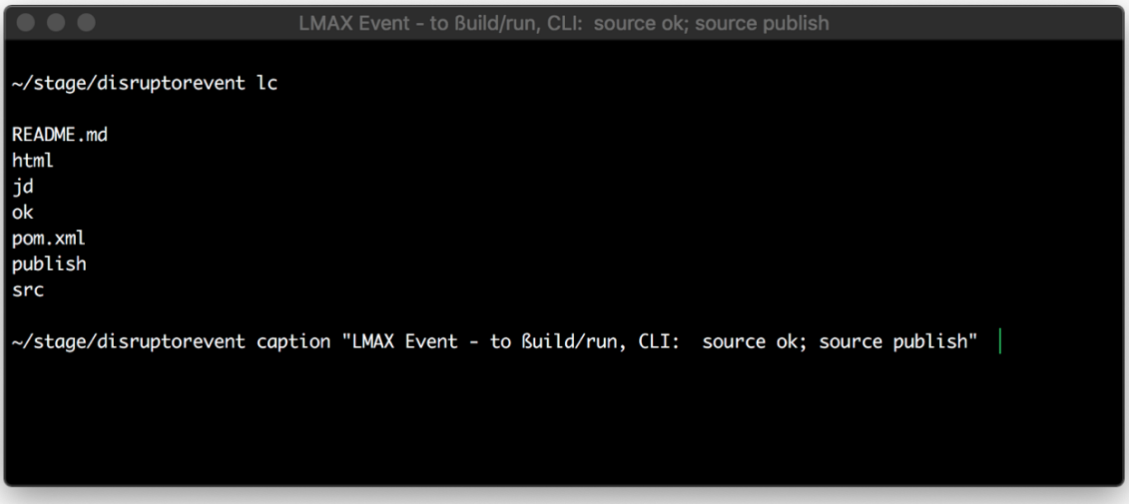
The lifecycle scripts are shown immediately below.



ßuild/run – This is the main project (1/3), Exchanger (Disruptor).

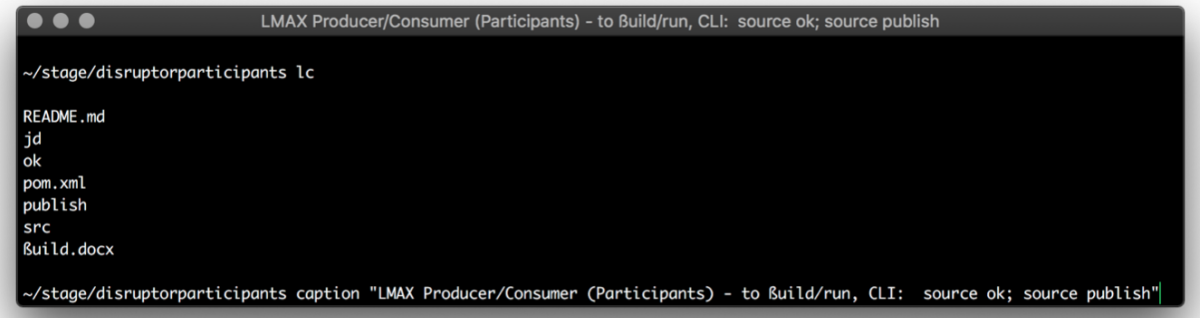
This component of the application triad directly leverages LMAX technology.

The other two application components are: Event, Participants (Producer/Consumer).



ßuild/run – This is the Event project (2/3).

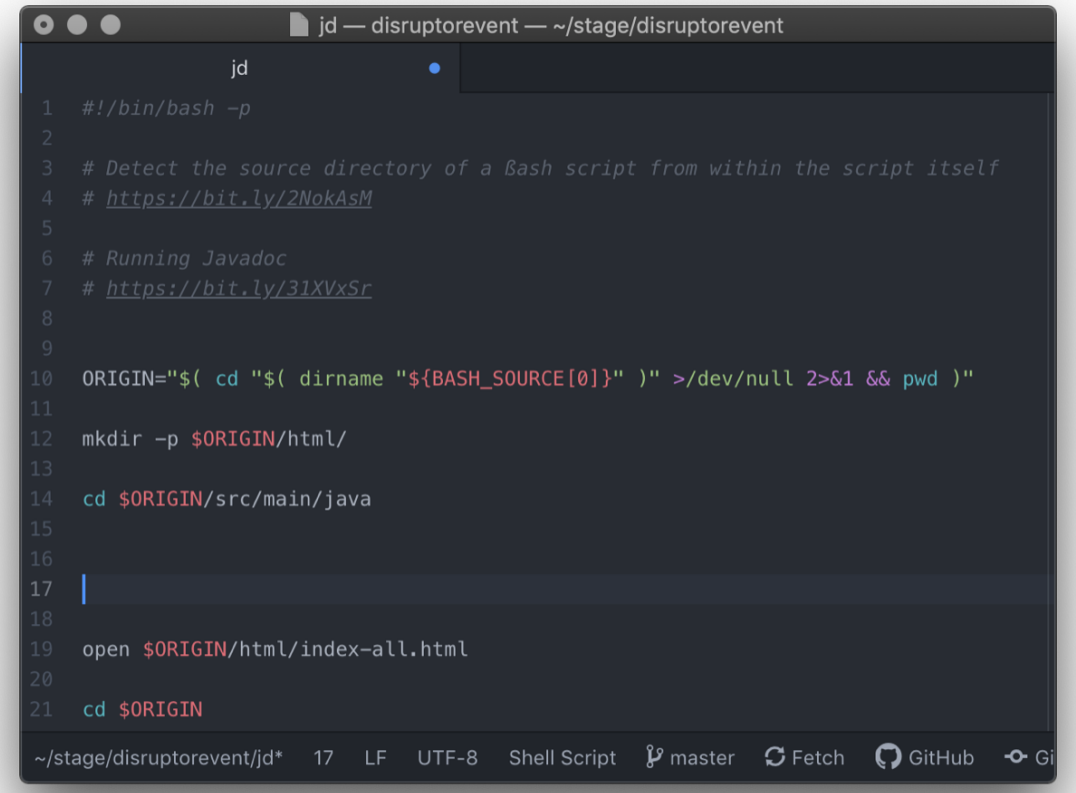
This part of the application triad is an intricate part of the LMAX technology; this event is an LMAX-aware event.



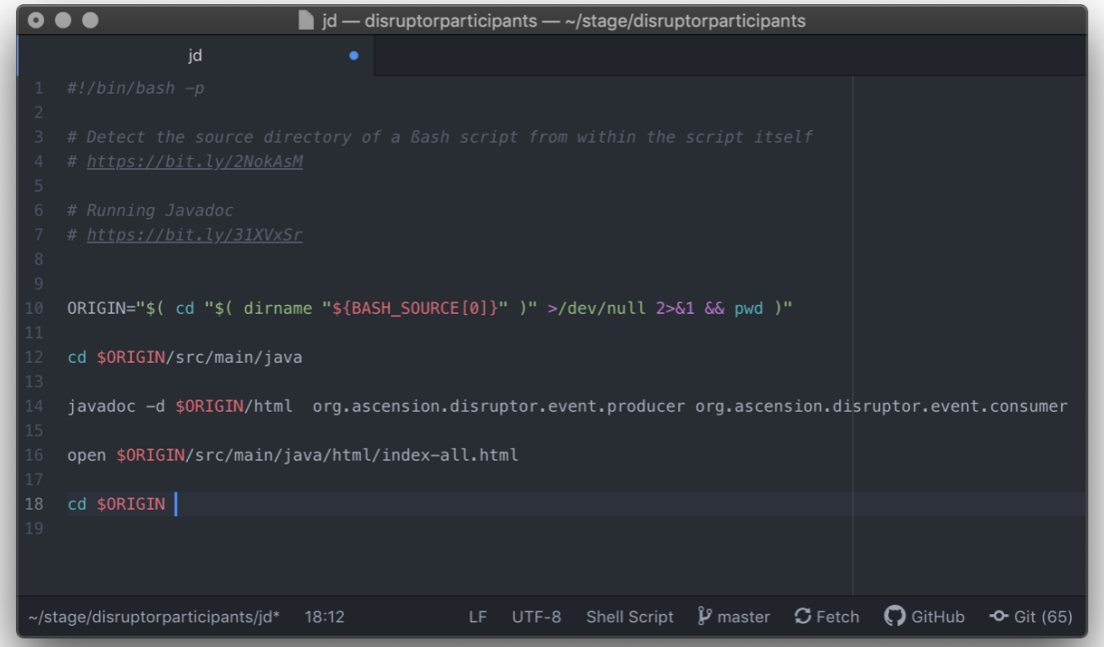
ßuild/run – This is the (Producer/Consumer) Disruptor-centric part of the triad (3/3).

This part of the application triad dispatches/retrieves LMAX Events.

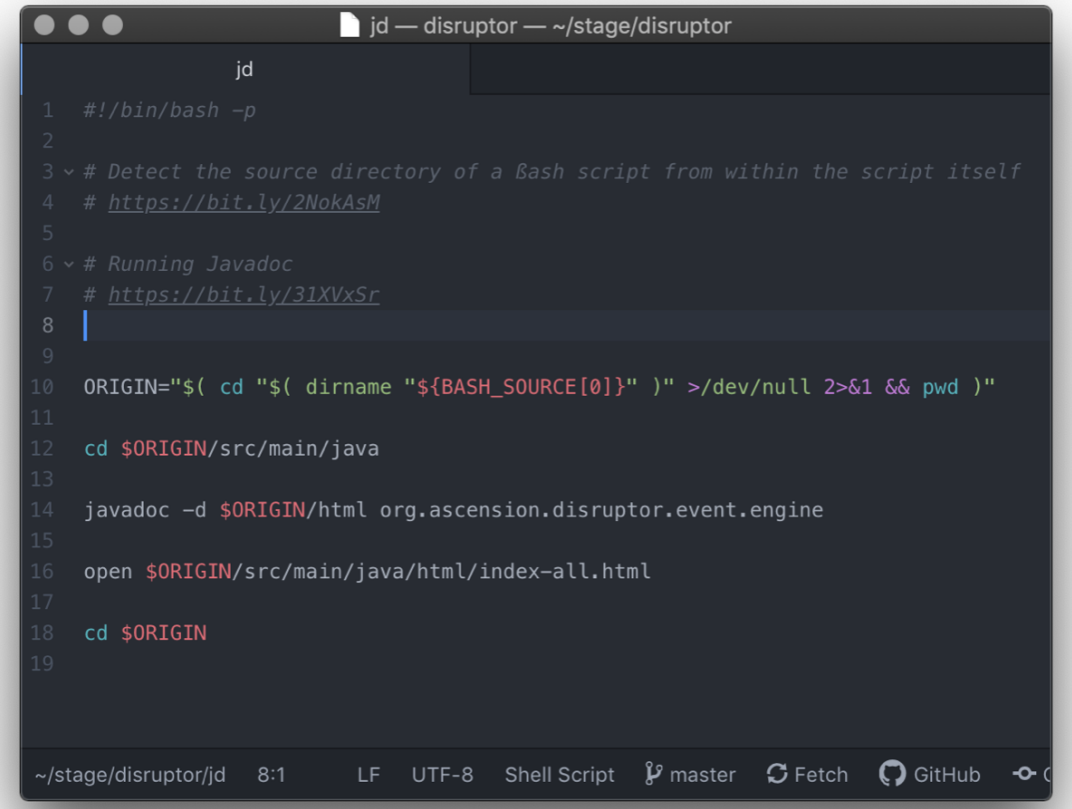
Javadoc, CLI



Javadoc – Event component



Javadoc – Participants component



Javadoc – Exchanger component

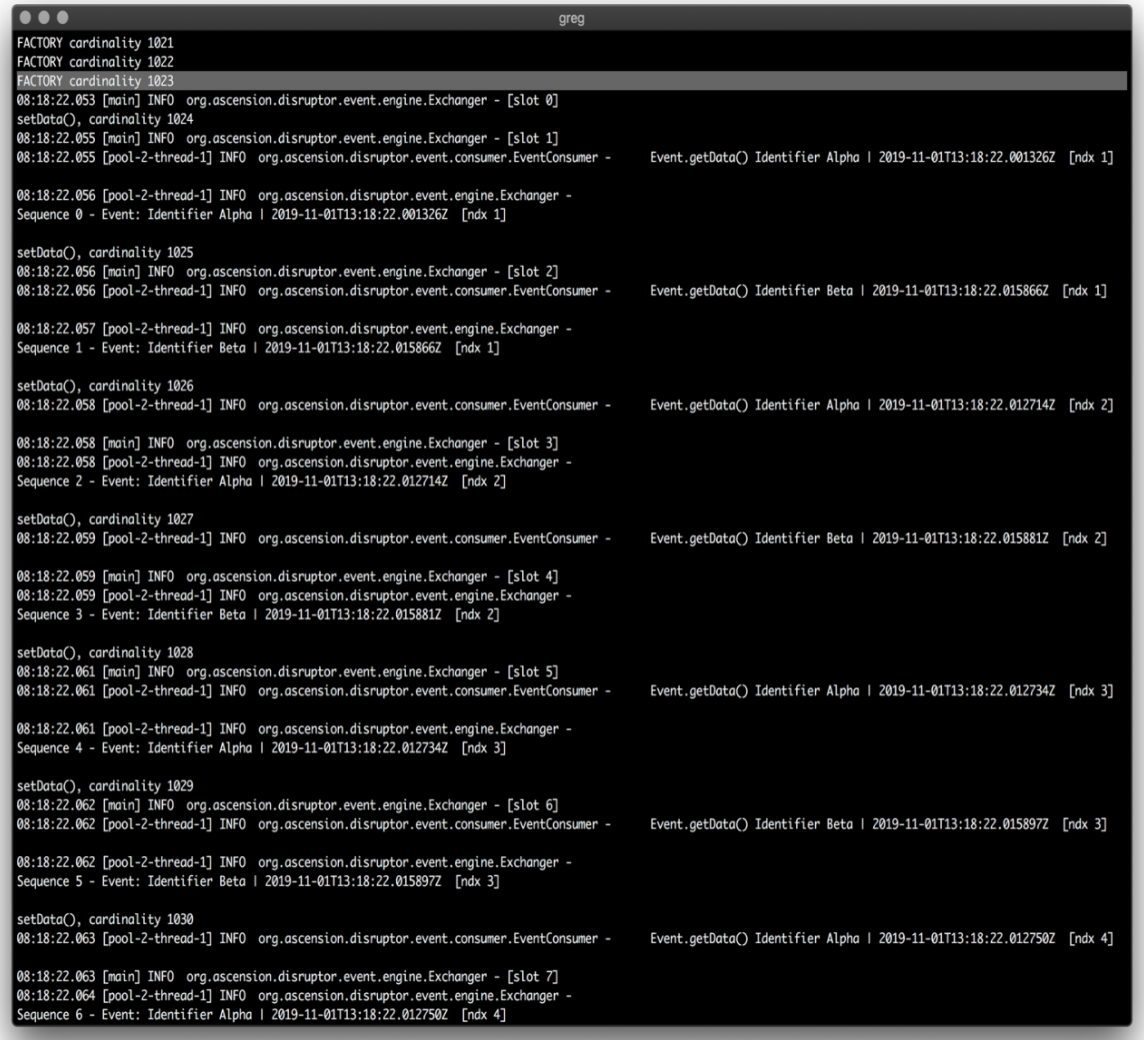
Application, running

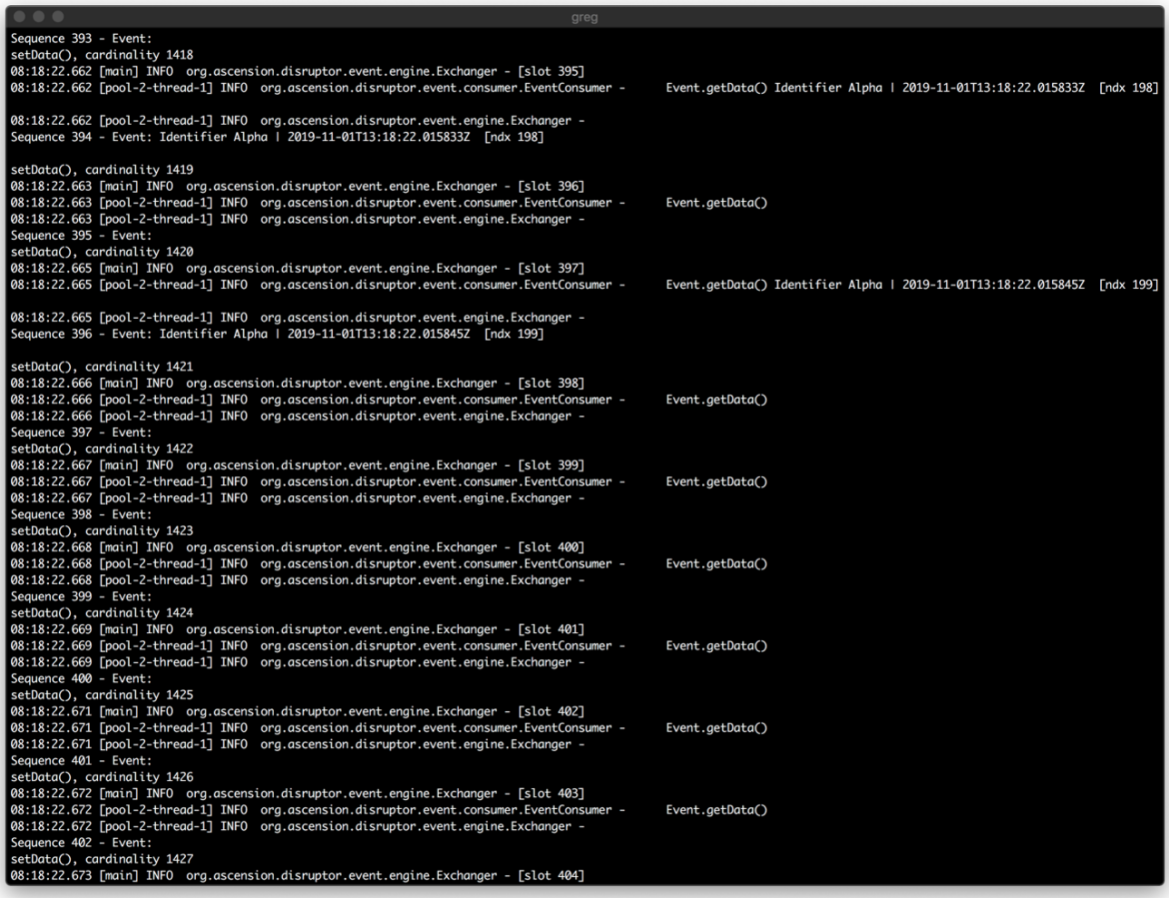
The Disruptor first creates the specific number of slots [Exchanger].

As events stream into the Exchanger, its internal instance of:

```
com.lmax.disruptor.EventHandler
```

… will acquire data from the slots which are populated w/ data from the Exchanger's `com.lmax.disruptor.RingBuffer` performing polling (of a *Producer*'s callback method) for `org.ascension.disruptor.event.Event`s:





As Events stop coming into the RingBuffer, its slots go empty (but the polling continues to fill slots, now with empty values). The memory for the slots is allocated at program inception, so it is only CPU cycles that are "wasted".

CTL-Z or CTL-C terminates the sole thread-of-execution, but there is a panic() method that is unused, but that can gracefully cleanup application components and release memory.

Appendix

* The LMAX platform is able to process market activity on the scale of 100K messages/events per second. LMAX's internal latency is 80µs, yielding a nominal trade latency of 3ms. (user experience or wall time).

† Async logging, log4j2
https://logging.apache.org/log4j/2.x/manual/async.html#UnderTheHood

‡ Async logging is a related topic; however, it is not used in the demo app. If utilized by the demo app, async logging would need a Correlation ID (CID) to maintain a semblance of continuity.

The demo app could create such a CID by importing Java's UUID:

```
import java.util.UUID;
```

https://mvnrepository.com/artifact/com.lmax/disruptor/3.2.1

https://docs.oracle.com/javase/8/docs/api/java/util/UUID.html#method.summary

§ Provides an adaptation of the Disruptor Pattern to broker an Event-driven Producer/Consumer app.

|| The interleaving behavior is "baked into" the polling logic/strategy of the Exchanger. This strategy, as code in the demo app, is *Round-Robin* and is determined by modulo arithmetic; the divisor is based upon the number of *Producers*.

¶ The padding provision – innate to the Disruptor – prevents "corruption" of the contents of cache lines, which occurs when multiple (client) threads contend-for/share a part of a host machine's CPUs cache lines.

To compound matters, there are typically at least four (4) cores per CPU, and there may be multiple CPUs per machine.

When two or more variables share part of a cache line, and values of one of more of the other variables is invalidated, the entire cache line's contents must be invalidated too; this affects the cache lines in all CPUs, all of which must then be invalidated.

This antipattern is known as False Sharing.

The value of this meta-variable is architecture-dependent – so, it is important to choose a value the padding that is a multiple of the largest payload (Event) that the app (will allow, by convention) to be processed.

(*continues*)

(*continued*)

NB

While constructing an instance of the `com.lmax.disruptor.dsl.Disruptor.java` class, note that the implementation of the `ExecutorService` interface is deprecated in favor of an instance of a `ThreadFactory` instance; this is because a `ThreadFactory` reports when it's unable to construct a *Producer* `Thread`.

References

https://dzone.com/articles/ring-buffer-a-data-structure-behind-disruptor

https://trishagee.github.io/post/dissecting_the_disruptor_demystifying_memory_barriers/

http://trishagee.github.io/post/dissecting_the_disruptor_writing_to_the_ring_buffer/

http://mechanitis.blogspot.com/2011/06/dissecting-disruptor-how-do-i-read-from.html

https://lmax-exchange.github.io/disruptor/docs/com/lmax/disruptor/EventHandler.html#method.summary