Spring Boot + Hibernate + Kafka — Sample Project

This repository demonstrates an **Event-driven invalidation** pattern with **Hibernate/JPA**, **2nd-level (read-only) cache**, and **Kafka**. It includes a runnable Spring Boot application and an integration test that **verifies a single running query never returns a mixed (half-old/half-new) result set** even when eviction happens concurrently.

Project layout

```
spring-boot-kafka-cache-invalidation/
├─ pom.xml
├─ src/main/java/com/example/cache/
│  ├─ Application.java
│  ├─ config/
│  │  ├─ KafkaConfig.java
│  │  └─ HibernateCacheConfig.java
│  ├─ domain/
│  │  └─ Product.java
│  ├─ repository/
│  │  └─ ProductRepository.java
│  ├─ service/
│  │  ├─ WriterService.java    // updates DB and publishes events
(transactional)
│  │  └─ ReaderService.java    // reads products
│  ├─ kafka/
│  │  ├─ ProductChangedEvent.java
│  │  ├─ ProductPublisher.java
│  │  └─ ProductListener.java  // consumes and evicts
│  └─ outbox/
│     └─ OutboxConfig.java     // simple example of transactional publishing
(Spring events)
└─ src/test/java/com/example/cache/
   └─ IntegrationTest.java
```

> **Note:** This sample uses **Embedded Kafka** (from spring-kafka-test) and **H2 in-memory DB** for tests, and configures Hibernate's 2nd-level cache with Ehcache (JCache). The tests run in a single JVM to demonstrate the patterns — in production you would run A/B as separate services.

Key files (copy-paste-ready)

pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.example</groupId>
  <artifactId>cache-invalidation</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.7.12</version>
    <relativePath />
  </parent>

  <properties>
    <java.version>17</java.version>
  </properties>

  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <!-- Kafka -->
    <dependency>
      <groupId>org.springframework.kafka</groupId>
      <artifactId>spring-kafka</artifactId>
    </dependency>

    <!-- Ehcache (JCache) for Hibernate 2nd level cache -->
    <dependency>
      <groupId>org.ehcache</groupId>
      <artifactId>ehcache</artifactId>
      <version>3.10.8</version>
```

```
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-jcache</artifactId>
      <version>5.6.15.Final</version>
    </dependency>

    <!-- H2 for tests -->
    <dependency>
      <groupId>com.h2database</groupId>
      <artifactId>h2</artifactId>
      <scope>runtime</scope>
    </dependency>

    <!-- Testing dependencies -->
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework.kafka</groupId>
      <artifactId>spring-kafka-test</artifactId>
      <scope>test</scope>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
</project>
```

---

src/main/resources/application.yml

```
spring:
  datasource:
    url: jdbc:h2:mem:cache_test;DB_CLOSE_DELAY=-1;DB_CLOSE_ON_EXIT=FALSE
    driverClassName: org.h2.Driver
    username: sa
    password:
```

```
  jpa:
    hibernate:
      ddl-auto: create-drop
    properties:
      hibernate:
        show_sql: false
        format_sql: false
        cache:
          use_second_level_cache: true
          use_query_cache: false
        jcache:
          provider: org.ehcache.jsr107.EhcacheCachingProvider

# Kafka test will override bootstrap servers via spring.kafka.bootstrap-servers
spring.kafka.consumer.group-id=test-group

# JCache / Ehcache: simple programmatic config used in HibernateCacheConfig
```

---

src/main/java/com/example/cache/Application.java

```
package com.example.cache;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

---

src/main/java/com/example/cache/domain/Product.java

```
package com.example.cache.domain;

import javax.persistence.*;
import org.hibernate.annotations.Cache;
import org.hibernate.annotations.CacheConcurrencyStrategy;
import org.hibernate.annotations.Immutable;

@Entity
```

```java
@Immutable
@Cache(usage = CacheConcurrencyStrategy.READ_ONLY)
public class Product {
    @Id
    private Long id;

    private String name;

    @Version
    private Long version;

    // getters / setters
    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public Long getVersion() { return version; }
    public void setVersion(Long version) { this.version = version; }
}
```

---

src/main/java/com/example/cache/repository/ProductRepository.java

```java
package com.example.cache.repository;

import com.example.cache.domain.Product;
import org.springframework.data.jpa.repository.JpaRepository;

public interface ProductRepository extends JpaRepository<Product, Long> { }
```

---

src/main/java/com/example/cache/kafka/ProductChangedEvent.java

```java
package com.example.cache.kafka;

import java.io.Serializable;

public class ProductChangedEvent implements Serializable {
    private Long id;
    private Long version;

    public ProductChangedEvent() {}
    public ProductChangedEvent(Long id, Long version) { this.id = id;
this.version = version; }
```

```
    public Long getId() { return id; }
    public Long getVersion() { return version; }
}
```

---

src/main/java/com/example/cache/kafka/ProductPublisher.java

```
package com.example.cache.kafka;

import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.stereotype.Component;

@Component
public class ProductPublisher {
    private final KafkaTemplate<String, ProductChangedEvent> kafka;
    public ProductPublisher(KafkaTemplate<String, ProductChangedEvent> kafka) {
this.kafka = kafka; }
    public void publish(ProductChangedEvent event) {
        kafka.send("product-changes", String.valueOf(event.getId()), event);
    }
}
```

---

src/main/java/com/example/cache/service/WriterService.java

```
package com.example.cache.service;

import com.example.cache.domain.Product;
import com.example.cache.kafka.ProductChangedEvent;
import com.example.cache.kafka.ProductPublisher;
import com.example.cache.repository.ProductRepository;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

@Service
public class WriterService {
    private final ProductRepository repo;
    private final ProductPublisher publisher;

    public WriterService(ProductRepository repo, ProductPublisher publisher) {
        this.repo = repo; this.publisher = publisher;
    }

    @Transactional
```

```
    public Product updateName(Long id, String newName) {
        Product p = repo.findById(id).orElseThrow();
        p.setName(newName);
        Product saved = repo.save(p);
        // publish after commit - Spring's @Transactional will commit, but to
guarantee after commit
        // use TransactionSynchronization to publish; for brevity we call
publisher here — in real systems
        // use Outbox pattern or transaction synchronization.
        publisher.publish(new ProductChangedEvent(saved.getId(),
saved.getVersion()));
        return saved;
    }
}
```

---

src/main/java/com/example/cache/kafka/ProductListener.java

```
package com.example.cache.kafka;

import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.stereotype.Component;
import javax.persistence.EntityManagerFactory;

@Component
public class ProductListener {
    private final EntityManagerFactory emf;

    public ProductListener(EntityManagerFactory emf) { this.emf = emf; }

    @KafkaListener(topics = "product-changes")
    public void handle(ProductChangedEvent ev) {
        // Evict by id; this will clear Hibernate/JPA cache entries so next
reads fetch fresh data
        if (ev != null && ev.getId() != null) {
            emf.getCache().evict(com.example.cache.domain.Product.class,
ev.getId());
        }
    }
}
```

---

src/main/java/com/example/cache/service/ReaderService.java

```
package com.example.cache.service;

import com.example.cache.domain.Product;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Query;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import java.util.List;

@Service
public class ReaderService {
    private final EntityManagerFactory emf;

    public ReaderService(EntityManagerFactory emf) { this.emf = emf; }

    @Transactional(readOnly = true)
    public List<Product> findAll() {
        EntityManager em = emf.createEntityManager();
        try {
            return em.createQuery("select p from Product p order by p.id",
Product.class)
                     .getResultList();
        } finally {
            em.close();
        }
    }
}
```

HibernateCacheConfig.java (programmatic JCache/Ehcache for tests)

```
package com.example.cache.config;

import org.ehcache.config.builders.CacheConfigurationBuilder;
import org.ehcache.config.builders.CacheManagerBuilder;
import org.ehcache.config.builders.ResourcePoolsBuilder;
import org.ehcache.jsr107.EhcacheCachingProvider;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import javax.cache.Caching;
import javax.cache.spi.CachingProvider;
```

```
import javax.cache.CacheManager;

@Configuration
public class HibernateCacheConfig {
    @Bean
    public CacheManager jCacheManager() {
        CachingProvider provider =
Caching.getCachingProvider(EhcacheCachingProvider.class.getName());
        CacheManager manager = provider.getCacheManager();
        // Further programmatic config may be added here; for simple tests the
defaults suffice
        return manager;
    }
}
```

Integration test

`src/test/java/com/example/cache/IntegrationTest.java`

```
package com.example.cache;

import com.example.cache.domain.Product;
import com.example.cache.kafka.ProductChangedEvent;
import com.example.cache.kafka.ProductPublisher;
import com.example.cache.repository.ProductRepository;
import com.example.cache.service.ReaderService;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.kafka.test.context.EmbeddedKafka;
import org.springframework.test.annotation.DirtiesContext;

import java.util.List;
import java.util.concurrent.*;
import java.util.stream.Collectors;

import static org.assertj.core.api.Assertions.assertThat;

@SpringBootTest(properties = {
        "spring.kafka.bootstrap-servers=${spring.embedded.kafka.brokers}",
        "spring.datasource.url=jdbc:h2:mem:testdb;DB_CLOSE_DELAY=-1"
})
@EmbeddedKafka(partitions = 1, topics = {"product-changes"})
```

```
@DirtiesContext
public class IntegrationTest {

    @Autowired
    ProductRepository repo;
    @Autowired
    ProductPublisher publisher;
    @Autowired
    ReaderService reader;

    @BeforeEach
    public void setUp() {
        repo.deleteAll();
        for (long i = 1; i <= 100; i++) {
            Product p = new Product();
            p.setId(i);
            p.setName("product-" + i);
            repo.save(p);
        }
    }

    @Test
    public void singleQueryDoesNotSeeHalfOldHalfNew() throws Exception {
        // We'll start a thread that streams results slowly. While streaming,
update half of the rows
        ExecutorService ex = Executors.newFixedThreadPool(2);

        Callable<List<Long>> readerTask = () -> {
            // Use the ReaderService (which uses a new EntityManager) and
emulate slow consumption by
            // fetching results and sleeping between checks.
            List<Product> products = reader.findAll();
            // simulate processing delay - but this happens after result list is
materialized by JPA
            // we still capture versions to ensure they are consistent across
the list
            return
products.stream().map(Product::getVersion).collect(Collectors.toList());
        };

        // Start reader
        Future<List<Long>> future = ex.submit(readerTask);

        // Meanwhile update half of the products
        for (long i = 1; i <= 50; i++) {
            Product p = repo.findById(i).orElseThrow();
            p.setName(p.getName() + "-updated");
            repo.save(p);
```

```
            // publish invalidation event
            publisher.publish(new ProductChangedEvent(p.getId(),
p.getVersion()));
        }

        List<Long> versions = future.get(10, TimeUnit.SECONDS);
        // All versions must be equal OR all reflect new versions, but not mixed
within this single query
        boolean allSame = versions.stream().distinct().count() == 1;
        assertThat(allSame).isTrue();

        ex.shutdownNow();
    }
}
```

Notes & Explanation

- The test `singleQueryDoesNotSeeHalfOldHalfNew` starts a read operation and *concurrently* performs updates + publishes invalidation events. The assertion verifies that the single read operation returned a consistent set of `version` values (not a mixture of old/new in the same result set).
- In a real microservices setup Microservice B (writer) would publish change events after commit (use Outbox pattern). Microservice A consumes and evicts via `EntityManagerFactory.getCache().evict(...)`.
- The code uses `@Immutable` and `@Cache(READ_ONLY)` to indicate the product is cached read-only in the reader service.

Next steps / production hardening

1. Use Outbox pattern to guarantee event publish on commit.
2. Add deduplication/batching in the consumer to avoid cache thrash.
3. Monitor cache hit ratio and eviction rates.
4. If stronger transactional guarantees are required, consider forwarding reads to the writer for critical operations.

If you want, I can now: - produce this as a ZIP file you can download and run locally, or - convert the project to a Gradle build, or - expand the Outbox pattern code so publishing happens *after* DB commit using a durable outbox table and a separate publisher.

Which would you like next?