

BEARS-1101: Handle EVENT_BETTING_CONTROLS_UPDATED Kafka Message



SARATH RAMACHANDRAN

Apprenticeship Activity Report

Introduction

In this activity, I was assigned the BEARS-1101 ticket during a sprint planning session by my team lead. The business need was driven by a requirement from Base Pricing stakeholders to improve customer fairness during in-play betting by implementing **bet delays**. These delays help prevent users from exploiting timing gaps between event outcomes and system updates.

The application is a **Kafka-based Spring Boot microservice** that consumes **event-driven kafka messages**, processes them, and updates a **DynamoDB** table with the latest event data. My responsibility was to handle the `EVENT_BETTING_CONTROLS_UPDATED` Kafka message and correctly integrate the bet delay information into the existing `Event` schema stored in DynamoDB.

I worked in a cross-functional agile team of developers, QA engineers, DevOps, and business stakeholders, attending stand-ups, retrospectives, and sprint ceremonies. The work was logged in Jira and GitLab, with clear acceptance criteria, source control, and CI pipelines.

Requirement Gathering

The requirement was delivered via a Jira ticket (BEARS-1101) with acceptance criteria, a sample Kafka message payload, and a Confluence reference page. The goal was to:

- Parse the incoming message
- Extract and write `betDelayDefault` to the event level
- Optionally, if `marketBettingControlsUpdated` exists, apply the delay to that market

Tools & Sources:

- Jira (Kanban board)
- Confluence (GTP Outbound Message Spec)
- Kafka Schema

Stakeholders Involved:

- Base Pricing Product Owner
- Lead Developer
- QA
- DevOps Engineer

We worked within an Agile sprint. Requirements were tracked in Jira using Kanban-style boards, with daily stand-ups and planning sessions providing further clarification. The team follows Agile methodology with two-week sprints. Tickets progress through **To Do** → **In Progress** → **Code Review** → **Testing** → **Done stages**.

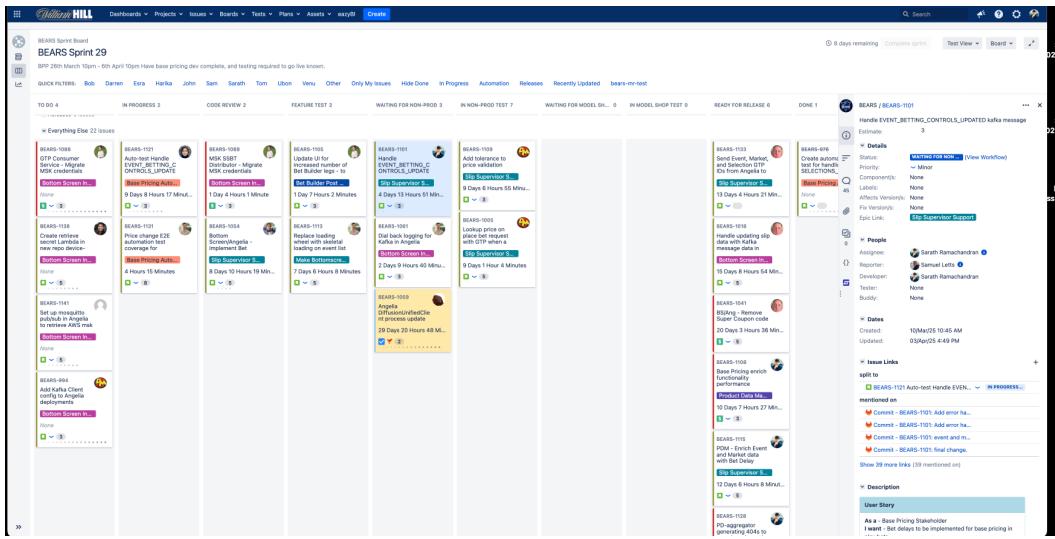


Figure 1 Sprint Board

Analysis

To understand where the new message fits:

- I reviewed the **DynamoDbService.java**, which handles existing event types.
- I explored related test files like `MessageListenerTest` and `EventPricedMergerTest`.
- I analyzed the `Event` model and verified how the `betDelay` field is structured at both event and market levels.

Key decisions:

- Reuse existing builder-based POJOs (`Event`, `Market`, etc.)
- Follow the structure of similar Kafka handlers (`EventStatusUpdatedMessage`, etc.)

Design

The plan was to:

- Extend the service class to support the new message type
- Use the builder pattern to modify nested structures safely (e.g., `Market.builder().betDelay(...)`)
- Reuse existing `mergeMarkets()` logic to ensure selections remain intact

A UML sketch was created showing the message flow from Kafka → `MessageListener` → `DynamoDbService` → DynamoDB write operation.

The design was simple: leverage existing structure and add logic to process `EVENT_BETTING_CONTROLS_UPDATED` messages. If `betDelayDefault` is > 0, add it to the

event object. If market-level controls exist, attach those delays to the appropriate markets. We reused the Event builder and merger utility where applicable.

The method `handleEventBettingControlsUpdated` was introduced in DynamoDbService, using the Enhanced DynamoDB Client API to insert or update records.

Development

All work was done in a feature branch `feature/BEARS-1101-betDelay` , rebased frequently against `develop` .

Tools and Stack:

- Java 21 (Amazon Corretto)
- Spring Boot 3.4
- IntelliJ IDEA
- DynamoDbEnhancedClient
- Mockito (for test mocking)
- JUnit 5
- Maven / GitLab CI

Key implementations:

- Created method:
`handleEventBettingControlsUpdated(EventBettingControlsUpdatedMessage)`
- Handled nested updates to `Market` objects
- Reused `EventPricedMerger.mergeMarkets()`
- Ensured `getItem` , `putItem` methods are mockable/testable

I created a new method `handleEventBettingControlsUpdated` within `DynamoDbService` . This method reads values from the Kafka message and inserts them into the DynamoDB table via an enhanced client. I used builder-style POJOs to compose the new Event and Market records. Unit tests were written to validate conditional handling of `betDelayDefault` and nested `marketBettingControlsUpdated` values.

Challenges included understanding builder chaining for deeply nested objects, mocking table reads and writes and resolving merge conflicts when working with teammates. I learned to use `argThat()` for verifying deep object graphs, and used reflection to inject mocks during testing.

Code changes were made to DynamoDbService.java, Event.java, Market.java, and supporting utility classes. The key method implemented was `handleEventBettingControlsUpdated` , using the Enhanced DynamoDbClient to retrieve existing records and update them with delays. Builder patterns and safe null handling were adopted throughout.

The code was committed and pushed for review via GitLab merge requests.

Ticket details and code change snippets -

BILLION HILL Dashboard Projects Issues Boards Tests Plan Assets [Search](#) [Logout](#)

BEARS

MARS-100 Handle EVENT_BETTING_CONTROLS_UPDATED kafka message

[Details](#) [Add comment](#) [Assign](#) [More](#) [Working for New Prod](#) [Edit](#)

Project: BEARs Sprint Board **Priority**: Major **Affected Services**: None

Description: **Event Type**: [BETTING_CONTROLS_UPDATED](#) **Event Date**: BEARS Sprint 20, BEARS Sprint 23 **Event Priority**: Major **KPI**: Cycle Time: 8 Days 4 Hours 42 Minutes

Details

User Story

As a Bet Pricing Stakeholder
I want to see the latest price for bet pricing to play bets
So that live betting prices are produced when customers are able to see a result and have a bet or a known outcome prior to feeds updating our prices

Ticket Description

Trading provider BWM_BETTING_CONTROLS_UPDATED kafka message

```
1 { "eventType": "EVENT_BETTING_CONTROLS_UPDATED", "eventDate": "2020-04-01T12:00:00+00:00", "eventPriority": "Major", "kpi": "Cycle Time", "affectedServices": "None", "lastUpdated": "2020-04-01T12:00:00+00:00", "version": "1.0", "versionLabel": "Initial Version", "versionNotes": "Initial Version", "versionStatus": "RELEASED", "versionType": "Major", "versionValue": "1.0", "versionLabel": "Initial Version", "versionNotes": "Initial Version", "versionStatus": "RELEASED", "versionType": "Major", "versionValue": "1.0" }
```

[See https://zentral.billionhill.com/jira/browse/MARS-100](#)

We want to handle this message, and update the lastUpdated field in the event date in Oracle DB, we will prove [marketdatacontrolupdated](#) for now

Comments

General

Given - look up dynamic DB
When - there is a [BETTING_CONTROLS_UPDATED](#) message
Then - there is a [lastUpdated](#) value in the event date and the marketdatacontrol updated as working as they are today

Scenario

Given - look up dynamic DB
When - an event is created and there is a [BETTING_CONTROLS_UPDATED](#) message with marketdatacontrol updated property
Then - there is a [lastUpdated](#) value in the event date and the marketdatacontrol updated from the lastEvent value

Attachments

[Screenshot 2020-04-01 at 12.00.00 PM \(1\).png](#) [Screenshot 2020-04-01 at 12.00.00 PM \(2\).png](#) [Drop files to attach, or browse.](#)

Issue Links

[#BEARS-102 Auto-test Handle EVENT_BETTING_CONTROLS_UPDATED kafka message](#) [#BEARS-103 Auto-test Handle EVENT_BETTING_CONTROLS_UPDATED kafka message](#)

Tested By

✓ [Cerner - BEARS-100 Auto-test handling a marketdatacontrol message for consistency](#) ✓ [Cerner - BEARS-101 Auto-test handling a marketdatacontrol message for consistency](#)
✓ [Cerner - BEARS-104 Auto-test and marking lastEvent valid](#) ✓ [Cerner - BEARS-105 True change](#)

People

Assignee: [Search...@billionhill.com](#) [Assign](#) [Unassign](#)
Reporter: [Search...@billionhill.com](#) [Assign](#) [Unassign](#)
Developer: [Search...@billionhill.com](#) [Assign](#) [Unassign](#)
Milestones: [Search...@billionhill.com](#) [Assign](#) [Unassign](#)
Watchers: [Start watching this issue](#)

Data

Created: 15Mar20 10:43 AM
Updated: 15Mar20 11:03 AM

Agile

BEARS Sprint 20 and BEARS-102
Completed Sprint: BEARS Sprint 20 and BEARS-102
[Find an issue](#)

Stack

In order to see discussions, first confirm access to your Stack account(s) in the following workspace(s): [Willam Hill](#)

Figure 2 Ticket Description

The screenshot shows a GitHub pull request interface for a project named "Retail MSK Dynamo Uploader". The pull request is titled "BEARS-1101: Handle EVENT_BETTING_CONTROLS_UPDATED kafka message" and is merged into the "develop" branch. The interface includes a sidebar with navigation links like Project, Pinned, Issues, Merge requests, Pipelines, Manage, Plan, Code, Merge requests, Repository, Branches, Commits, Tags, Repository graph, Compare revisions, Snippets, Locked files, Build, Secure, Deploy, Operate, and Help. The main area displays two code review sections:

- MessageListener.java**: A Java class with methods for handling different message types. The code includes annotations like @Override and @Data, and imports from com.williamhill.msksdk and org.springframework.stereotype.Component.
- EventPricedItemFormatter.java**: A Java class that formats event-priced items. It uses imports from com.williamhill.gtp.outboundapi, com.williamhill.msksdk, org.springframework.stereotype.Component, and java.util.stream.Collectors.

Each code review section shows the commit history, including the author, date, and file changes. The interface also includes a search bar, a file tree on the left, and various status indicators at the top right.

Figure 3 Code changes - MessageListener

The screenshot shows a GitHub pull request titled "Merged: BEARS-1101: Handle EVENT_BETTING_CONTROLS_UPDATED kafka message feature/BEARS-1101-new into develop". The code changes are located in the file `dynamoDbService.java`. The changes are as follows:

```

81 +     }
82 +
83 +     public void handleEventBettingControlsUpdated(EventBettingControlsUpdatedMessage message) {
84 +         try {
85 +             DynamoDbTable<Event> table = dynamoDbEnhancedClient.table(tableName, TableSchema.fromBean(Event.class));
86 +             Event dynamoBEventBetDelayItem = table.getItem(r -> r.key(k -> k.partitionValue(message.getEventId())));
87 +             Event eventSet0DelayItem = formatDataUpdate(message, dynamoBEventSet0DelayItem);
88 +             table.putItem(eventSet0DelayItem);
89 +             logger.info("{} message sent to DynamoDB for event {}", message.obtainMessageType(), message.getEventId());
90 +         } catch (Exception e) {
91 +             logger.error("Error processing {} message: {}", message.obtainMessageType(), e.getMessage());
92 +             throw new RuntimeException("Failed to handle EVENT_BETTING CONTROLS message", e);
93 +         }
94 +     }
95 +
96 +
97 + }
98 +
99 + private Event FormatDataUpdate(EventBettingControlsUpdatedMessage message, Event existingDynamoDBData) {
100 +     Map<String, Market> markets = message.getMarketBettingControlsUpdated();
101 +     Stream<Event> stream() {
102 +         collect(
103 +             Collectors.toMap(marketBettingControlsUpdated -> marketBettingControlsUpdated.getMarketId(),
104 +                 marketBettingControlsUpdated ->
105 +                     Market.builder().betDelay(marketBettingControlsUpdated.getBetDelay()).build()
106 +             ));
107 +         Event newEvent = Event.builder()
108 +             .id(message.getEventId())
109 +             .betDelay(message.getBetDelayDefault())
110 +             .markets(markets)
111 +             .build();
112 +         if (existingDynamoDBData != null) {
113 +             return eventPriceMerger.mergeMarkets(newEvent, existingDynamoDBData);
114 +         }
115 +         return newEvent;
116 +     }
117 + }

```

Figure 4 Code changes - DynamoDbService

The screenshot shows a GitHub pull request titled "Merged: BEARS-1101: Handle EVENT_BETTING_CONTROLS_UPDATED kafka message feature/BEARS-1101-new into develop". The code changes are located in the file `dynamoDbService.java`. The changes are as follows:

```

81 +     }
82 +
83 +     public void handleEventBettingControlsUpdated(EventBettingControlsUpdatedMessage message) {
84 +         try {
85 +             DynamoDbTable<Event> table = dynamoDbEnhancedClient.table(tableName, TableSchema.fromBean(Event.class));
86 +             Event dynamoBEventBetDelayItem = table.getItem(r -> r.key(k -> k.partitionValue(message.getEventId())));
87 +             Event eventSet0DelayItem = formatDataUpdate(message, dynamoBEventSet0DelayItem);
88 +             table.putItem(eventSet0DelayItem);
89 +             logger.info("{} message sent to DynamoDB for event {}", message.obtainMessageType(), message.getEventId());
90 +         } catch (Exception e) {
91 +             logger.error("Error processing {} message: {}", message.obtainMessageType(), e.getMessage());
92 +             throw new RuntimeException("Failed to handle EVENT_BETTING CONTROLS message", e);
93 +         }
94 +     }
95 +
96 +
97 + }
98 +
99 + private Event FormatDataUpdate(EventBettingControlsUpdatedMessage message, Event existingDynamoDBData) {
100 +     Map<String, Market> markets = message.getMarketBettingControlsUpdated();
101 +     Stream<Event> stream() {
102 +         collect(
103 +             Collectors.toMap(marketBettingControlsUpdated -> marketBettingControlsUpdated.getMarketId(),
104 +                 marketBettingControlsUpdated ->
105 +                     Market.builder().betDelay(marketBettingControlsUpdated.getBetDelay()).build()
106 +             ));
107 +         Event newEvent = Event.builder()
108 +             .id(message.getEventId())
109 +             .betDelay(message.getBetDelayDefault())
110 +             .markets(markets)
111 +             .build();
112 +         if (existingDynamoDBData != null) {
113 +             return eventPriceMerger.mergeMarkets(newEvent, existingDynamoDBData);
114 +         }
115 +         return newEvent;
116 +     }
117 + }

```

Figure 5 Code changes - DynamoDbService

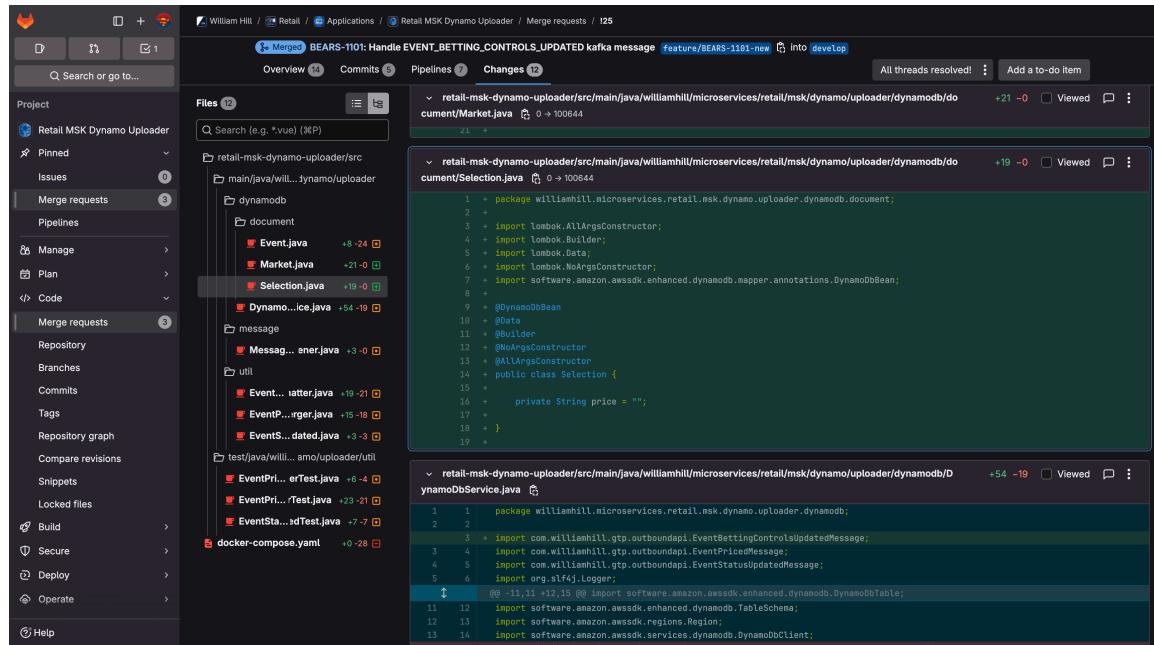


Figure 6 Code changes – Selection

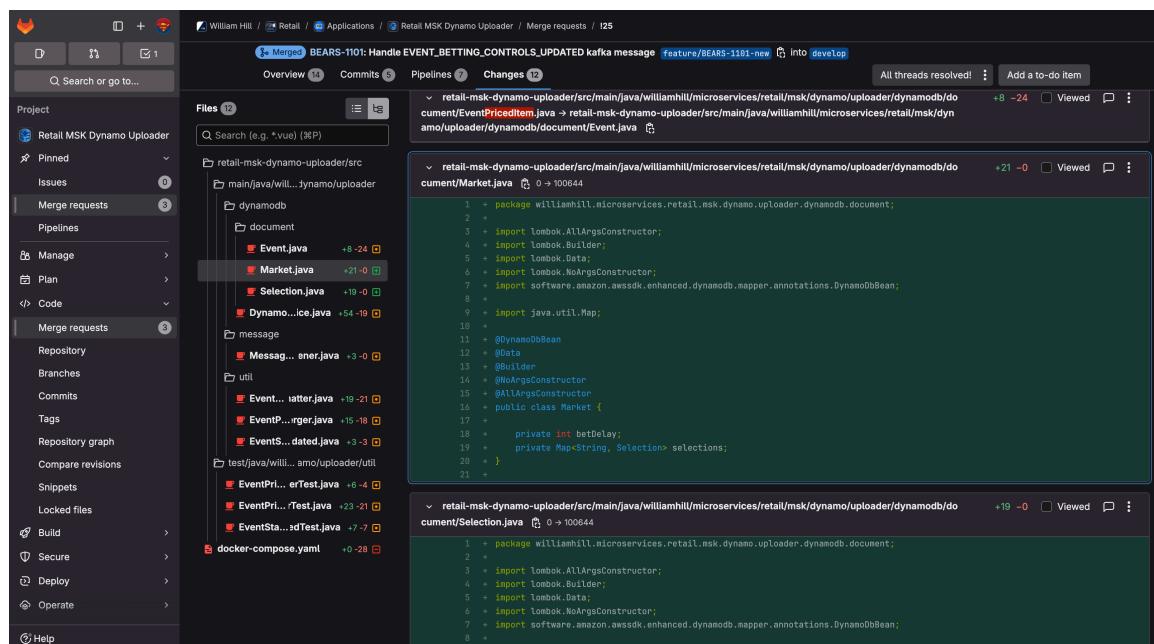


Figure 7 Code changes - Market

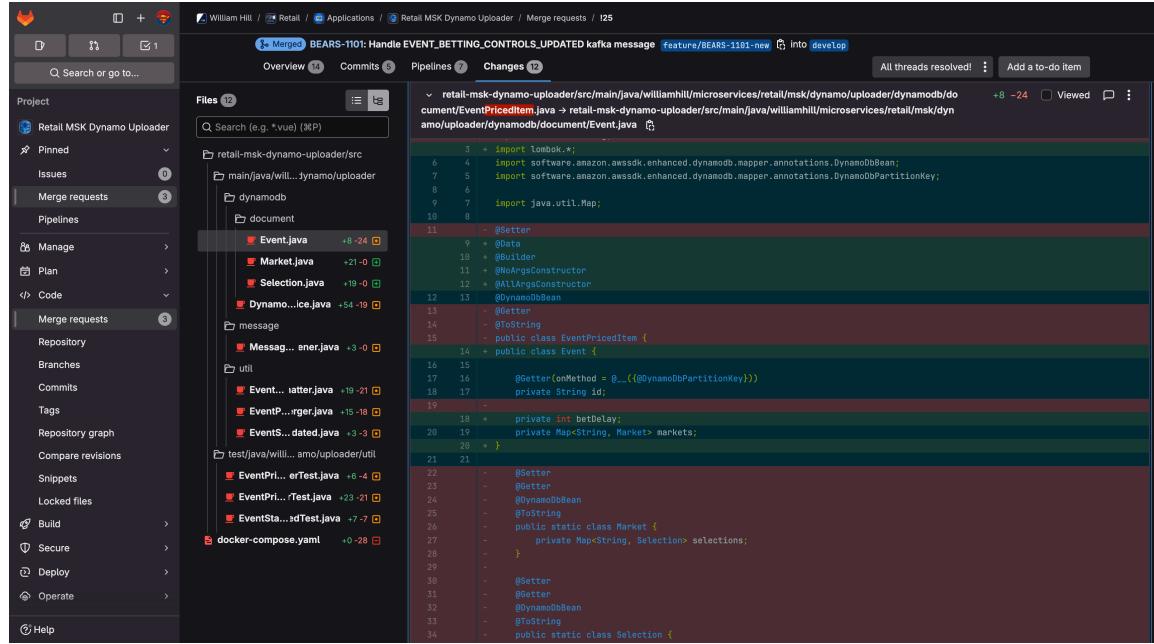


Figure 8 Code changes - Event

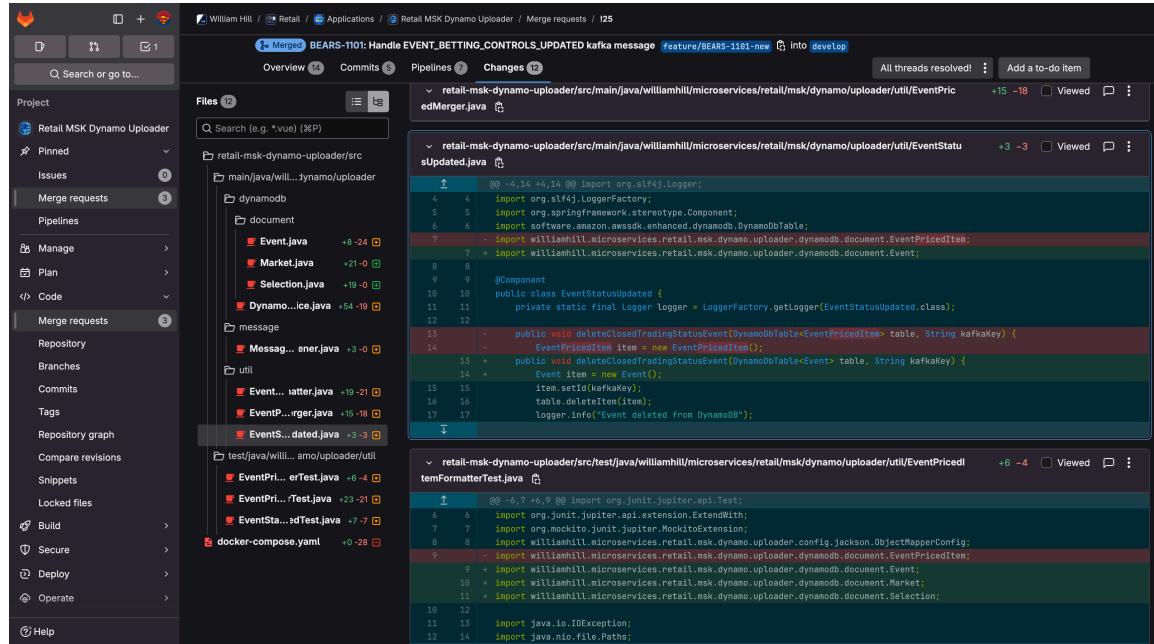


Figure 9 Code changes - EventStatusUpdated

The screenshot shows a code review interface for a Java project. The left sidebar contains a navigation menu with options like Project, Merge requests, Pipelines, and Code. The main area displays a diff between two branches. The code changes are shown in a Java file named `EventPricedMerger.java`. The diff highlights added lines (green) and deleted lines (red). The code itself is related to handling events and managing markets.

```

28 -     for (Map.Entry<String, EventPricedItem.Market> entry : newMarkets.entrySet()) {
29 -         String marketId = entry.getKey();
30 -         EventPricedItem.Market newMarket = entry.getValue();
31 -         Map<String, Market> newMarkets = newItem.getMarkets();
32 -         Map<String, Market> existingMarkets = existingItem.getMarkets();
33 -         newMarkets.forEach((marketId, newMarket) -> {
34 -             if (existingMarkets.containsKey(marketId)) {
35 -                 logger.info("Merging selections for existing market: {}", marketId);
36 -                 mergeSelections(existingMarkets.get(marketId), newMarket);
37 -                 existingMarkets.get(marketId).setSetDelay(newMarket.getSetDelay());
38 -             } else {
39 -                 logger.info("Adding new market: {}", marketId);
40 -                 existingMarkets.put(marketId, newMarket);
41 -             }
42 -         });
43 -         existingItem.setMarkets(existingMarkets);
44 -         existingItem.setSetDelay(newItem.getSetDelay());
45 -         logger.info("Markets merged successfully for event: {}", existingItem.getId());
46 -         return existingItem;
47 -     }
48 -     private void mergeSelections(EventPricedItem.Market existingMarket, EventPricedItem.Market newMarket) {
49 -         if (newMarket.getSelections() == null) {
50 -             logger.warn("No new selections to merge.");
51 -             return;
52 -         }
53 -         @@@ -56,13 +57,10 @@
54 -         public class EventPricedMerger {
55 -             ...
56 -         }
57 -     }
58 -     Map<String, EventPricedItem.Selection> existingSelections = existingMarket.getSelections();
59 -     Map<String, EventPricedItem.Selection> newSelections = newMarket.getSelections();
60 -     for (Map.Entry<String, EventPricedItem.Selection> entry : newSelections.entrySet()) {
61 -         ...
62 -     }
63 - }

```

Figure 10 Code changes - EventPricedMerger

The screenshot shows a git log interface for a Java project. The left sidebar contains a navigation menu with options like Project, Merge requests, Pipelines, and Code. The main area displays a list of commits. Each commit is shown with its author, timestamp, subject, and a list of changes. The commits are related to handling events and managing markets.

- Tom Hewitt: Resolved 1 week ago
 - 4 replies Last reply by Tom Hewitt 1 week ago
- Darren McSwiggan: Resolved 1 week ago by Sarath Ramachandran
 - 5 replies Last reply by Darren McSwiggan 1 week ago
- Darren McSwiggan: Resolved 1 week ago by Sarath Ramachandran
 - 2 replies Last reply by Venu Anna 1 week ago
- Sarath Ramachandran: added 1 commit 1 week ago
 - 37873434 - Revert "Merge branch 'feature/test-1090' into 'develop'"
 - Compare with previous version
- Darren McSwiggan: Resolved 1 week ago by Sarath Ramachandran
 - 1 reply Last reply by Darren McSwiggan 1 week ago
- Sarath Ramachandran: added 1 commit 1 week ago
 - 65c4f3fe - BEARS-1101: Add error handling in handleEventBettingControlsUpdated for consistency.
 - Compare with previous version
- Sarath Ramachandran: added 1 commit 1 week ago
 - ed5771d1 - BEARS-1101: Revert package.json.
 - Compare with previous version
- Sarath Ramachandran: added 1 commit 1 week ago
 - c1e7d4e9 - BEARS-1101: Remove unused docker-compose file as per MR comments
 - Compare with previous version

Figure 11 Git logs

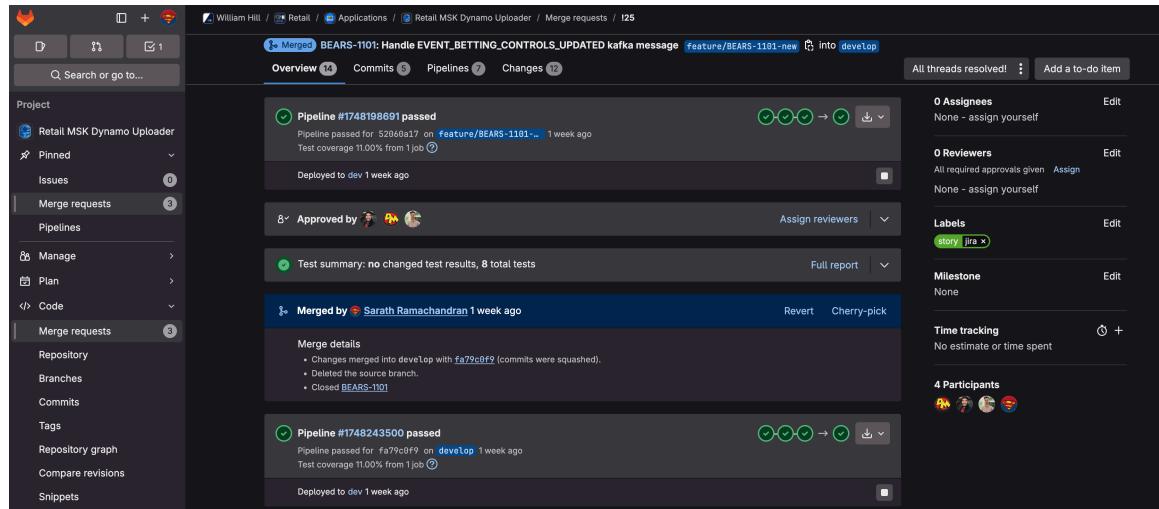


Figure 12 Git Overview

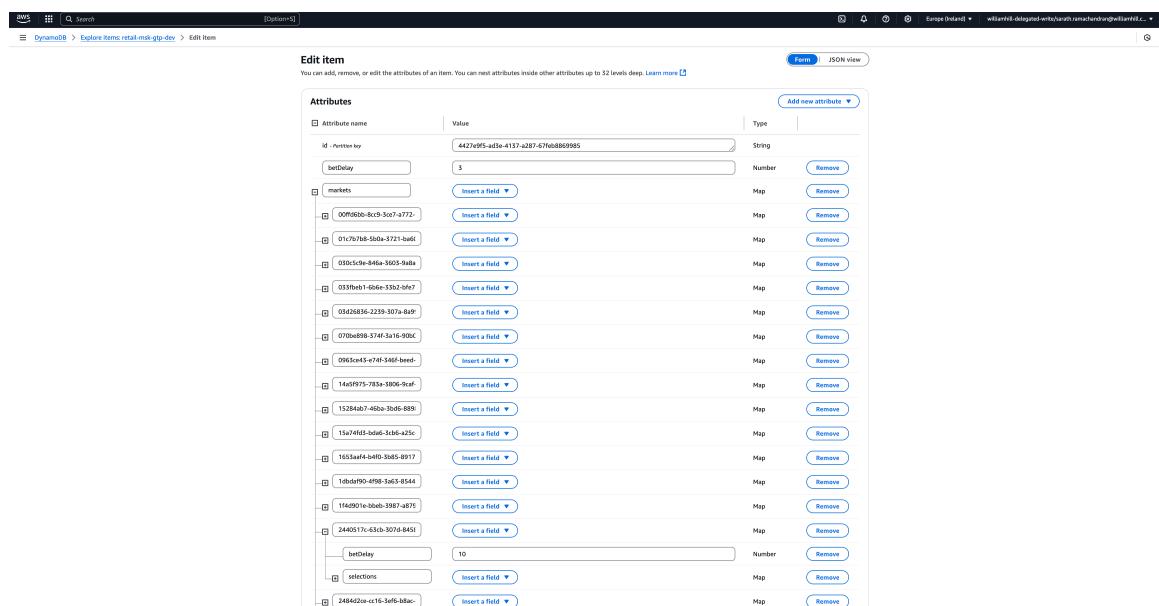


Figure 13 Updated Dynamo DB table

Debugging

Debugging & Issue Resolution During development, I encountered several issues:

- Lombok builder methods not working due to missing annotation processing
- IntelliJ misconfiguration leading to unresolved symbols
- Maven dependency resolution failures (1.0.0 artifact not found)

To resolve them:

- I enabled annotation processing in IntelliJ
- Reloaded Maven projects from disk and cleared caches
- Used mvn -U clean install and updated dependency versions

Debugging was done by adding log statements, checking stack traces, comparing working test files, and using IntelliJ's debugger to step through object creation logic.

Testing

Unit tests were written in DynamoDbServiceTest.java using JUnit 5 and Mockito. Three scenarios were validated:

- Skip update if betDelay is zero
- Insert new event with default and market-level delay
- Update existing event with delay values

These tests ensured no writes occurred when `betDelayDefault` was 0 and verified correct behavior with full payloads. Unit tests validate message transformation, merging logic, and DynamoDB writes. We used Maven to build and test locally, and tests ran successfully in the CI pipeline.

Unit testing allowed quick validation and confidence in the integration logic. I mocked Kafka message classes and nested builder objects, and asserted the final objects passed to DynamoDB using `argThat()` with deep matching logic.

The screenshot shows a GitHub pull request page for a Java project. The pull request is titled "BEARS-1101: Handle EVENT_BETTING_CONTROLS_UPDATED kafka message". It has 182 changes and is ready to merge. The code is located in the file `retail-msk-dynamo-uploader/src/test/java/williamhill/microservices/retail/msk/dynamo/uploader/dynamodb/DynamoDbServiceTest.java`. The changes are concentrated in the `testJava/williamhill/dynamodb` section, specifically in the `DynamoDbServiceTest.java` file. The code is annotated with JUnit tests and Mockito mock objects for `Event`, `Market`, and `Selection` classes.

```
    public void shouldHandleEventWithMultiplePriceData() {
        var eventPricedMessage = mock(EventPricedMessage.class);
        when(eventPricedMessage.getEventId()).thenReturn("eventId");
        when(eventPricedMessage.obtainMessageType()).thenReturn("EVENT_PRICED");

        var selectionPriced = mock(SelectionPriced.class);
        when(selectionPriced.getSelectionId()).thenReturn("selectionId");
        when(selectionPriced.getPrices()).thenReturn(List.of(Price.builder().value("10/1").build()));

        var marketPriced = mock(MarketPriced.class);
        when(marketPriced.getMarketId()).thenReturn("marketId");
        when(marketPriced.getSelectionsPrices()).thenReturn(List.of(selectionPriced));

        Selection selection = Selection.builder()
            .price("10/1")
            .build();

        Event formattedEvent = Event.builder()
            .id("eventId")
            .betDelay(4)
            .markets(Map.of("marketId", Market.builder().betDelay(5).selections(Map.of("selectionId", selection)).build()))
            .build();

        when(eventPricedMessage.mergeMarkets(any(), any())).thenReturn(formattedEvent);
        when(eventPricedMessageFormatter.formatEventPriceMessage(any(EventPricedMessage.class))).thenReturn(formattedEvent);
        when(mockTable.getItem(any(GetItemEnhancedRequest.class))).thenReturn(existingEvent);
        service.handleEventPriceMessage(eventPricedMessage);
        verify(mockTable).putItem(formattedEvent);
    }
}
```

Figure 14 Code changes - DynamoDbServiceTest

Figure 15 Code changes - DynamoDbServiceTest

```

28 +     public class DynamoDbServiceTest {
29 +         ...
30 +         private DynoamodService service;
31 +         private DynoamodConnectionClient enhancedClient;
32 +         private Map<String, String> mockTable;
33 +         private EventPriceMerger eventPriceMerger;
34 +         private EventPriceItemFormatter eventPriceItemFormatter;
35 +         private EventStatusUpdated eventStatusUpdated;
36 +         private EventPriceDateFormatter eventPriceDateFormatter;
37 +         private EventPriceTableFormatter eventPriceTableFormatter;
38 +         private @Mock DynamoDbTable<String> mockTable;
39 +         private void setup() {
40 +             enhancedClient = mock(DynamoDbConnectionClient.class);
41 +             eventPriceMerger = mock(EventPriceMerger.class);
42 +             eventPriceItemFormatter = mock(EventPriceItemFormatter.class);
43 +             eventStatusUpdated = mock(EventStatusUpdated.class);
44 +             eventPriceTableFormatter = mock(EventPriceTableFormatter.class);
45 +         }
46 +         ...
47 +         @Test
48 +         void shouldHandlePutEventBetBettingControlsUpdated() {
49 +             var message = mock(EventBettingControlsUpdatedMessage.class);
50 +             when(message.getEventId()).thenReturn("event-1");
51 +             when(message.getEventName()).thenReturn("event-name");
52 +             when(message.getEventTime()).thenReturn("2023-01-01T00:00:00Z");
53 +             when(message.getEventStatus()).thenReturn("PENDING");
54 +             when(message.getEventPriceTable()).thenReturn(Collections.emptyList());
55 +             when(service.handleEventBettingControlsUpdated(message)).thenReturn(null);
56 +             when(eventPriceMerger.mergeEvents(any(Event.class))).thenReturn(formattedEvent);
57 +             when(mockTable.getitem(any(GetItemEnhancedRequest.class))).thenReturn(null);
58 +             when(eventPriceTableFormatter.getTables(any(Event.class))).thenReturn(formattedTable);
59 +             when(mockTable.putitem(any(PutItemEnhancedRequest.class))).thenReturn(null);
60 +             when(eventPriceMerger.mergeTables(any(Event.class))).thenReturn(formattedTable);
61 +             when(mockTable.getitem(any(GetItemEnhancedRequest.class))).thenReturn(null);
62 +             when(eventPriceTableFormatter.getTables(any(Event.class))).thenReturn(formattedTable);
63 +             when(mockTable.putitem(any(PutItemEnhancedRequest.class))).thenReturn(null);
64 +             when(eventPriceTableFormatter.getTables(any(Event.class))).thenReturn(formattedTable);
65 +             when(mockTable.getitem(any(GetItemEnhancedRequest.class))).thenReturn(null);
66 +             when(eventPriceTableFormatter.getTables(any(Event.class))).thenReturn(formattedTable);
67 +             when(mockTable.getitem(any(GetItemEnhancedRequest.class))).thenReturn(null);
68 +             when(eventPriceTableFormatter.getTables(any(Event.class))).thenReturn(formattedTable);
69 +             when(mockTable.getitem(any(GetItemEnhancedRequest.class))).thenReturn(null);
70 +             when(eventPriceTableFormatter.getTables(any(Event.class))).thenReturn(formattedTable);
71 +             when(mockTable.getitem(any(GetItemEnhancedRequest.class))).thenReturn(null);
72 +             when(eventPriceTableFormatter.getTables(any(Event.class))).thenReturn(formattedTable);
73 +             when(mockTable.getitem(any(GetItemEnhancedRequest.class))).thenReturn(null);
74 +             when(eventPriceTableFormatter.getTables(any(Event.class))).thenReturn(formattedTable);
75 +             when(mockTable.getitem(any(GetItemEnhancedRequest.class))).thenReturn(null);
76 +             when(eventPriceTableFormatter.getTables(any(Event.class))).thenReturn(formattedTable);
77 +             when(mockTable.getitem(any(GetItemEnhancedRequest.class))).thenReturn(null);
78 +             when(eventPriceTableFormatter.getTables(any(Event.class))).thenReturn(formattedTable);
79 +             when(mockTable.getitem(any(GetItemEnhancedRequest.class))).thenReturn(null);
80 +             when(eventPriceTableFormatter.getTables(any(Event.class))).thenReturn(formattedTable);
81 +             when(mockTable.getitem(any(GetItemEnhancedRequest.class))).thenReturn(null);
82 +             when(eventPriceTableFormatter.getTables(any(Event.class))).thenReturn(formattedTable);
83 +         }
84 +     }

```

Figure 16 Code changes - DynamoDbServiceTest

Communication

Throughout this task, I maintained regular communication with my team to ensure clarity and alignment. Key interactions included:

- **Daily Stand-ups:** I provided updates on the ticket's progress during morning team calls, discussing blockers and next steps.
- **Peer Reviews & Feedback:** I collaborated with a teammate who had also worked on the same feature branch previously. I reviewed their code and incorporated some of their logic for consistency and maintainability.
- **Slack Conversations:** I shared implementation doubts, test coverage ideas, and mocking strategies with the team via Slack. When I faced issues with ambiguous builder methods or mocking nested classes, I reached out for guidance.
- **Merge Request Discussions:** I submitted my code via a GitLab Merge Request, where I responded to reviewer comments and made necessary improvements. This helped refine both logic and test coverage.
- **Pair Debugging:** For test failures and runtime issues, I sat with my senior developer for pair debugging, which helped resolve builder-related edge cases effectively.

This open and collaborative approach helped ensure that the feature aligned with team expectations and coding standards and helped me upskill through real-time guidance.

Result

The final implementation was merged into the develop branch after successful review.

DynamoDB now stores bet delays for in-play events as expected. Unit tests cover all functional scenarios.

I used Git to create a feature branch, commit changes, and resolve conflicts with teammate commits. Each commit message was clear and referenced the ticket. Screenshots of Git log and ticket board are attached. Version control was maintained via GitLab, with branches rebased and updated as needed. The Maven build used JaCoCo and other plugins for code coverage and compilation.

The feature was visible in the sprint demo, and acceptance criteria from the ticket were marked as completed in JIRA.

The screenshot shows the JIRA ticket interface for the ticket 'Handle EVENT_BETTING_CONTROLS_UPDATED kafka message'. The ticket is in the 'Done' status. The 'Details' section shows the ticket type as 'Story', priority as 'Normal', and resolution as 'None'. The 'Ticket Description' section contains the ticket's purpose and acceptance criteria. The 'Acceptance Criteria' section lists several items, many of which are marked as 'Completed'. The 'Attachments' section shows a screenshot of the 'Board' view in JIRA, where the ticket has moved from 'In Progress' to 'Done'. The 'Board' view shows the ticket's progress through various stages: In Progress, Feature Test, Ready for Review, and Done. The ticket is currently in the 'Done' stage.

Figure 17 Ticket status after successful completion

Reflection

This activity significantly enhanced my understanding of real-world backend development and improved my confidence across full SDLC stages. I learned how to:

- Extend event-driven services with new Kafka message types
- Mock nested and builder-based POJOs using ` Mockito`
- Write precise assertions using ` argThat()` and lambda matchers
- Understand and modify code working with DynamoDbEnhancedClient
- Troubleshoot complex issues including IntelliJ settings, Lombok builder limitations, and Maven errors
- Perform Git rebase, conflict resolution, and align my work with ` develop`

I also became more confident in:

- Contributing to shared unit test suites
- Writing production-safe, readable code with encapsulation and builder chaining
- Communicating progress effectively during stand-ups and merge requests

This hands-on work has been one of the most complete, satisfying features I've delivered independently. Overall, the feature delivery was smooth and aligned with Agile expectations. This task deepened my knowledge of our microservice stack, encouraged best practices in testing and mocking, and increased my ownership confidence from ticket analysis to test delivery.