Implementing Domain-Driven Design
Domain Event

2025-07-07

Overview

1. The Concept of Events & "When/Why"

1-1. Definition in One Sentence

Domain Event

A "fact" that has occurred in the business domain and is worthy of notice, preserved as an object.

- Keywords: **Happened (past tense)**, **Business Significance**

1-1. Definition in One Sentence

Domain Event

A "fact" that has occurred in the business domain and is worthy of notice, preserved as an object.

- Keywords: **Happened (past tense)**, **Business Significance**
- Can be **replayed** / **propagated**

1-2. Why Use Events?

Scenario	**Without Events...**	**With Events...**
Async Integration (Other systems need to know when a user pays)	Direct RPC call May hang if the other system is down	Publish 'PaymentConfirmed' event Queued in MQ, consumer handles it later
One Tx, one Aggregate	Updating two Aggregates synchronously Breaks consistency or requires XA	Modify A first Publish event Handler opens new Tx to modify B
Replacing Batch Jobs	Scan DB at night for "state changes"	Event is pushed to a service immediately, logic is spread out

1-3. How to Identify an Event?

When talking with a Domain Expert, these phrases are red flags:

- **"When... happens"**: *When an order ships, send the customer a mail.*

This indicates a business rule needs to react to "something happening," which is a strong candidate for an event.

1-3. How to Identify an Event?

When talking with a Domain Expert, these phrases are red flags:

- **"When... happens"**: *When an order ships, send the customer a mail.*
- **"If... then..."**: *If the balance drops below zero, freeze the account.*

This indicates a business rule needs to react to "something happening," which is a strong candidate for an event.

1-3. How to Identify an Event?

When talking with a Domain Expert, these phrases are red flags:

- **"When... happens"**: *When an order ships, send the customer a mail.*
- **"If... then..."**: *If the balance drops below zero, freeze the account.*
- **"Notify me..." / "Tell me..."**: *Notify finance if a refund exceeds $10,000.*

This indicates a business rule needs to react to "something happening," which is a strong candidate for an event.

1-4. Events CRUD Logs

- It's **not** about calling it ‘RowInsertedEvent‘ just because a row was added.
- It must be at the **business semantic level**"UserRegistered," "InventoryDeducted"to be valuable.

1-5. Minimal Java Event Interface

```java
// Every event must implement this interface
public interface DomainEvent {
    Date occurredOn();
}

// Example of a concrete event
public class UserRegistered implements DomainEvent {
    private final String userId;
    private final String email;
    private final Date occurredOn;

    public UserRegistered(String userId, String email) {
        this.userId = userId;
        this.email = email;
        this.occurredOn = new Date();
    }

    @Override
    public Date occurredOn() {
        return this.occurredOn;
    }

    // Other getters...
}
```

A Small Exercise For You

1. Think about your current system. **What business actions are named in the past tense**?
2. List them out. Don't worry about code yet, just confirm the semantics.

2. How to Name and Model Events

2-1. Event Name: Past Tense + Ubiquitous Language

1. **Source = Command**
   - In the Scrum AgilePM example, the ‘BacklogItem‘ aggregate has an action:

```
backLogItem.commitTo(aSprint);
```

   - This "action" is the **cause** of the event.

2-1. Event Name: Past Tense + Ubiquitous Language

1. **Source = Command**
   - In the Scrum AgilePM example, the ‘BacklogItem‘ aggregate has an action:

```
backLogItem.commitTo(aSprint);
```

   - This "action" is the **cause** of the event.
2. **Name = Past Tense of the Cause**
   - Because ‘commitTo()‘ has completed, the event is named ‘BacklogItemCommitted‘.

2-1. Event Name: Past Tense + Ubiquitous Language

1. **Source = Command**
   - In the Scrum AgilePM example, the ‘BacklogItem‘ aggregate has an action:

```
backLogItem.commitTo(aSprint);
```

   - This "action" is the **cause** of the event.
2. **Name = Past Tense of the Cause**
   - Because ‘commitTo()‘ has completed, the event is named ‘BacklogItemCommitted‘.

Principle

The name alone should make it clear that "this has happened" and fully align with the Bounded Context’s **Ubiquitous Language**.

2-2. Minimal Domain Interface: `DomainEvent`

The book first defines a very thin interface that all events must implement:

```
1 package com.saasovation.agilepm.domain.model;
2
3 import java.util.Date;
4
5 public interface DomainEvent {
6     Date occurredOn();
7 }
```

- Has only one responsibility: return the event's occurrence time via ‘occurredOn()‘.
- Keeping it minimal ensures events remain *lightweight* and easy to serialize.

2-3. Event Class BacklogItemCommitted

```java
package com.saasovation.agilepm.domain.model.product;

import java.util.Date;
import com.saasovation.agilepm.domain.model.DomainEvent;

public class BacklogItemCommitted implements DomainEvent {

    private Date         occurredOn;
    private BacklogItemId backlogItemId;
    private SprintId     committedToSprintId;
    private TenantId     tenantId;

    public BacklogItemCommitted(
            TenantId aTenantId,
            BacklogItemId aBacklogItemId,
            SprintId aCommittedToSprintId) {
        super();
        this.occurredOn        = new Date();
        this.backlogItemId     = aBacklogItemId;
        this.committedToSprintId = aCommittedToSprintId;
        this.tenantId          = aTenantId;
    }

    @Override
    public Date occurredOn() {
        return this.occurredOn;
    }

    public BacklogItemId backlogItemId() {
        return this.backlogItemId;
    }
```

2-3. Event Class (Key Points)

Element	Purpose
‘occurredOn‘	Every event carries a timestamp, making it easy to sort and replay
‘backlogItemId‘	The ID of the main aggregate involved in the event
‘committedToSprintId‘	The ID of the related aggregate that needs to be notified (Sprint)
‘tenantId‘	In the book’s example, it’s a multi-tenant SaaS; any cross-context communication must include the tenant ID

2-4. Keep it Immutable

- The constructor sets all fields at once.
- Only provide **read-only** methods, **no setters**.

2-4. Keep it Immutable

- The constructor sets all fields at once.
- Only provide **read-only** methods, **no setters**.
- **Benefits**:
  - Reduces concurrency issues
  - Simplifies serialization and snapshotting

2-5. When to Add More Fields (Event Enrichment)

- If a subscriber needs additional information to operate

2-5. When to Add More Fields (Event Enrichment)

- If a subscriber needs additional information to operate
- **But first ask yourself**: Can you use the ID to query the Repository?

2-5. When to Add More Fields (Event Enrichment)

- If a subscriber needs additional information to operate
- **But first ask yourself**: Can you use the ID to query the Repository?
- **Weigh the pros and cons**:
  - If the query cost is high or the event should carry the semantics, add fields.
  - (The book discusses more "fat" event structures in the appendix on Event Sourcing.)

2-6. Actual Publishing Location

Back to the ‘BacklogItem‘ aggregate:

```
1  public void commitTo(Sprint aSprint) {
2      // ...validation, state transition...
3
4      DomainEventPublisher
5          .instance()
6          .publish(new BacklogItemCommitted(
7              this.tenantId(),
8              this.backlogItemId(),
9              this.sprintId()));
10 }
```

- **Publish immediately after the cause**.

2-6. Actual Publishing Location

Back to the ‘BacklogItem‘ aggregate:

```
1  public void commitTo(Sprint aSprint) {
2      // ...validation, state transition...
3
4      DomainEventPublisher
5          .instance()
6          .publish(new BacklogItemCommitted(
7              this.tenantId(),
8              this.backlogItemId(),
9              this.sprintId()));
10 }
```

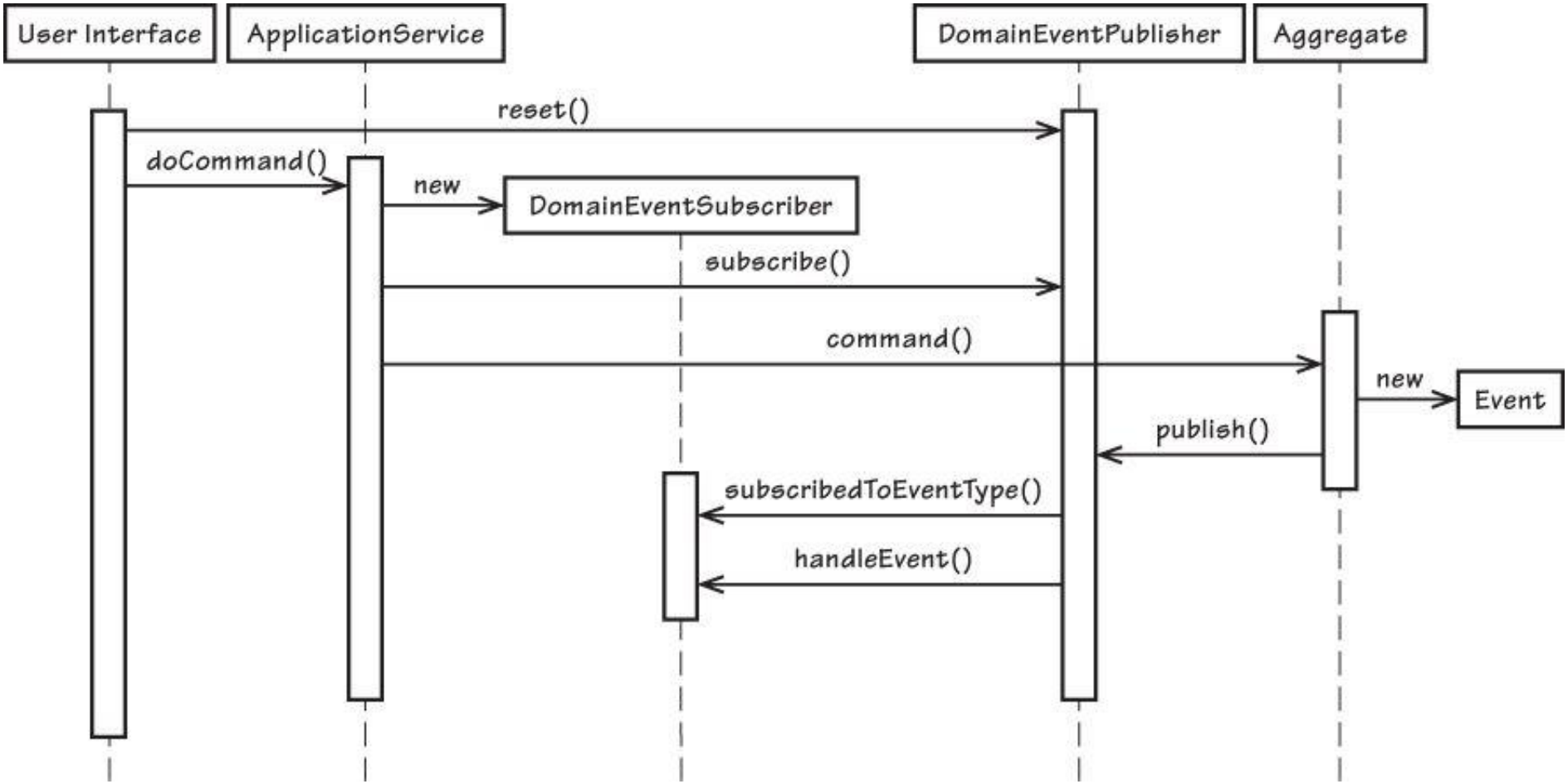
- **Publish immediately after the cause**.
- **Validate and change state first**, then publish the event, ensuring the "fact" is reliable.

Section Review

1. **Event Name**: Use past tense, directly corresponding to a command.
2. **Event Structure**: Implement ‘DomainEvent‘, keep it immutable.
3. **Required Fields**: ‘occurredOn‘ + primary aggregate ID + related aggregate ID for notification + ‘tenantId‘ (for multi-tenancy).
4. **Richness**: Keep it sufficient; add more fields if needed, but remember to maintain immutability.
5. **Publishing Time**: ‘publish()‘ immediately after the aggregate’s state has successfully changed and validation has passed.

3. Lightweight Publish-Subscribe

Event Publishing Sequence



3-1. Complete Source Code: DomainEventPublisher

```java
package com.saasovation.agilepm.domain.model;

import java.util.ArrayList;
import java.util.List;

public class DomainEventPublisher {

    @SuppressWarnings("unchecked")
    private static final ThreadLocal<List> subscribers =
            new ThreadLocal<List>();

    private static final ThreadLocal<Boolean> publishing =
            new ThreadLocal<Boolean>() {
                @Override protected Boolean initialValue() {
                    return Boolean.FALSE;
                }
            };

    public static DomainEventPublisher instance() {
        return new DomainEventPublisher();
    }
    public DomainEventPublisher() { super(); }

    @SuppressWarnings("unchecked")
    public <T> void publish(final T aDomainEvent) {
        if (publishing.get()) { // Prevent re-entrant calls
            return;
        }
        try {
            publishing.set(Boolean.TRUE);

            List<DomainEventSubscriber<T>> registeredSubscribers =
```

3-3. Typical Flow of a "Single Request"

```
1 // Enter a Web Filter (at the start of each HTTP request)
2 DomainEventPublisher.instance().reset();
3
4 // Application Service registers subscribers
5 DomainEventPublisher.instance().subscribe(subscriber);
6
7 // Execute Aggregate behavior publish event
8 backlogItem.commitTo(aSprint); // internally calls publish(...);
```

- ‘reset()‘ clears subscribers from the previous request.
- ‘publish()‘ triggers the event.

3-4. DomainEventSubscriber Interface

```
1 public interface DomainEventSubscriber<T> {
2     void handleEvent(T aDomainEvent);
3     Class<T> subscribedToEventType();
4 }
```

- ‘subscribedToEventType()‘
  - Returns the specific event class The Publisher uses this for matching.
  - Returning ‘DomainEvent.class‘ means "consume all events".
- ‘handleEvent()‘
  - The logic should be lightweight: it should **not** modify another Aggregate. For async work (MQ / Email), write to an Outbox or schedule a task.

3-5. Aggregate Publishing: Revisited

```
public void commitTo(Sprint aSprint) {
    // ...validation and state change...
    DomainEventPublisher.instance()
        .publish(new BacklogItemCommitted(
            this.tenantId(),
            this.backlogItemId(),
            this.sprintId()));
}
```

- First, ensure business invariants pass and state is updated then publish the event.
- Publishing immediately triggers all subscribers on the same thread; the transaction is only committed if successful.

Summary

1. **ThreadLocal + Synchronous Call**: Zero external dependencies, simple to test.
2. **reset subscribe publish**: These three steps align with a single HTTP / gRPC request.
3. **‘publishing‘ flag** ensures:
   - Re-entrant ‘publish()‘ calls are not allowed.
   - ‘subscribe()‘ can only be called when not publishing, avoiding concurrent list modification.
4. **Subscriber Logic Limit**: Only perform reads, logging, or writes to an Outbox. **Do not directly modify other Aggregates**.

4. Outbox/Event Store & Transactional Consistency

Three Basic Strategies

- **Shared Persistence Store**: Model and messages are written in the same local transaction.
  - \+ Pros: Simple, strong consistency.
  - \- Cons: Schema pollution.

Three Basic Strategies

- **Shared Persistence Store**: Model and messages are written in the same local transaction.
  - \+ Pros: Simple, strong consistency.
  - \- Cons: Schema pollution.
- **XA Two-Phase Commit**: The model's DB and the messaging store participate in the same global transaction.
  - \+ Pros: Technical separation.
  - \- Cons: Poor performance, limited support.

Three Basic Strategies

- **Shared Persistence Store**: Model and messages are written in the same local transaction.
  - \+ Pros: Simple, strong consistency.
  - \- Cons: Schema pollution.
- **XA Two-Phase Commit**: The model's DB and the messaging store participate in the same global transaction.
  - \+ Pros: Technical separation.
  - \- Cons: Poor performance, limited support.
- **Outbox / Event Store (Book's Example)**: Events are first written to a local DB 'events' table, then delivered by a separate forwarder.
  - \+ Pros: Local transaction guarantees consistency, supports replay.
  - \- Cons: Requires a custom forwarder, downstream consumers need to handle deduplication.

4-1. Why Choose Outbox / Event Store?

1. As a message queue, publishing to various integration systems
2. Providing pull-based notifications via REST
3. Complete history, not just for auditing, but also for replaying
4. Business intelligence, trend analysis, forecasting, etc.
5. Foundation for Event Sourcing
6. Corrections/Patches: Marking events as invalid

4-2. Global Subscription and Storing in Event Store

Use AOP to intercept all Application Services and register a subscriber that "consumes all events":

```
@Aspect
public class IdentityAccessEventProcessor {
    @Before(
      "execution(* com.saasovation...*.*(..))")
    public void listen() {
        DomainEventPublisher
            .instance()
            .subscribe(new DomainEventSubscriber<DomainEvent>() {
                @Override
                public void handleEvent(DomainEvent aDomainEvent) {
                    store(aDomainEvent);
                }
                @Override
                public Class<DomainEvent> subscribedToEventType() {
                    return DomainEvent.class; // all events
                }
            });
    }

    private void store(DomainEvent aDomainEvent) {
        EventStore.instance().append(aDomainEvent);
```

4-3. EventStore.append(...) Source Code

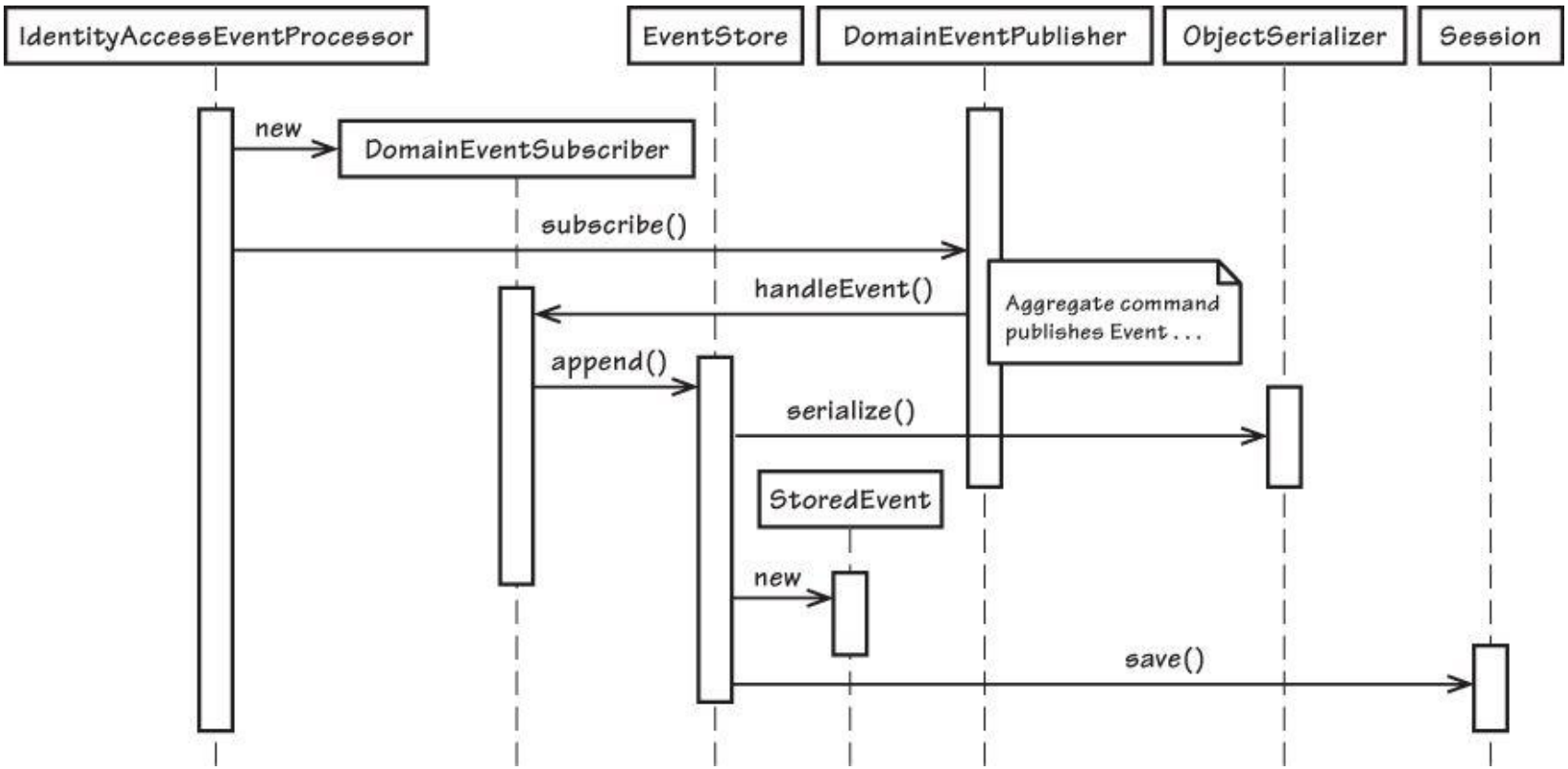
```
1  package com.saasovation.identityaccess.application.eventStore;
2
3  public class EventStore {
4      // Singleton access omitted...
5
6      public void append(DomainEvent aDomainEvent) {
7          String eventSerialization =
8              EventStore.objectSerializer().serialize(aDomainEvent);
9
10         StoredEvent storedEvent =
11             new StoredEvent(
12                 aDomainEvent.getClass().getName(),
13                 aDomainEvent.occurredOn(),
14                 eventSerialization);
15
16         this.session().save(storedEvent);
17         this.setStoredEvent(storedEvent);
18     }
19 }
```

‘append()‘ first serializes the ‘DomainEvent‘, wraps it in a ‘StoredEvent‘, and then writes it to the database via an ORM.

4-4. StoredEvent Class

```java
package com.saasovation.identityaccess.application.eventStore;

public class StoredEvent {
    private long  eventId;   // Auto-incrementing PK
    private String typeName; // Full class name of the event
    private Date  occurredOn; // Event timestamp
    private String eventBody; // JSON-serialized event content

    public StoredEvent(
        String aTypeName,
        Date  anOccurredOn,
        String anEventBody) {

        this.eventBody  = anEventBody;
        this.occurredOn = anOccurredOn;
        this.typeName   = aTypeName;
    }

    // getters / no setters immutable
}
```

Append to Event Store Sequence



4-5. Table DDL

The book uses MySQL as an example for the Event Store's table structure:

```
CREATE TABLE ‘tbl_stored_event‘ (
    ‘event_id‘   int(11)     NOT NULL AUTO_INCREMENT,
    ‘event_body‘ varchar(65000) NOT NULL,
    ‘occurred_on‘ datetime    NOT NULL,
    ‘type_name‘  varchar(100) NOT NULL,
    PRIMARY KEY (‘event_id‘)
) ENGINE=InnoDB;
```

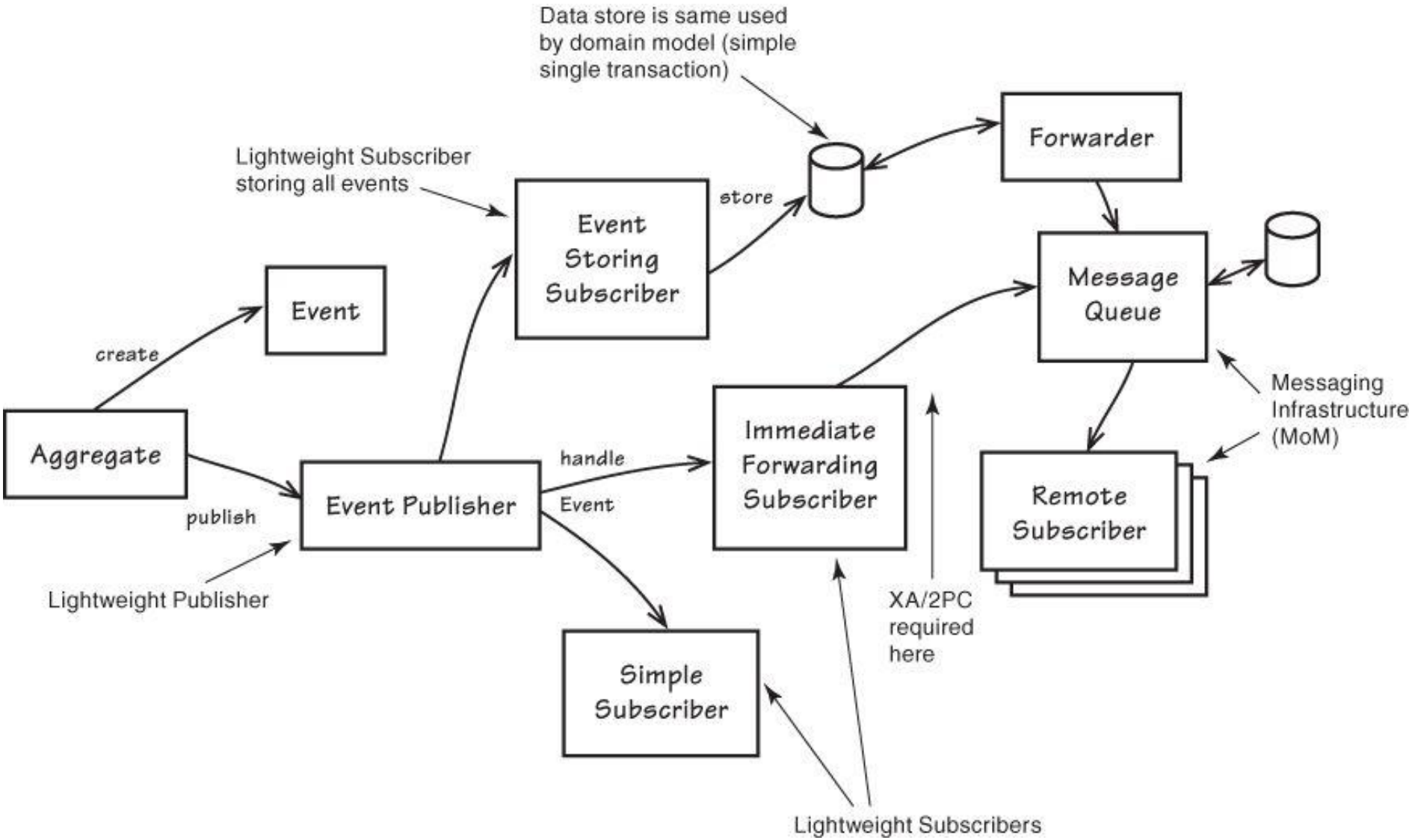
This `tbl_stored_event` table is the implementation of the Outbox pattern.

Section 4 Summary

- **Three Consistency Strategies**: Shared DB, XA, and Outbox/Event Store; the book's example uses the Outbox pattern.
- **AOP Global Subscription**: 'IdentityAccessEventProcessor' intercepts all AppServices, subscribes to all events, and calls 'append()'.
- **Immutable StoredEvent**: Encapsulates serialized data, a primary key, and a timestamp.
- **Outbox Table DDL**: `tbl_stored_event` serves as the persistence area. A Forwarder then pushes its content to MQ/REST, ensuring consistency between the model and events.

5. Forwarding Stored Events Across Bounded Contexts

Architectural Pipeline



5-1. Architectural Style Overview

- **Push via Middleware**:
  - Use messaging middleware like RabbitMQ or ActiveMQ to **push** stored events to subscribers.
- **Pull via REST**:
  - Expose a notification log URI, allowing clients to periodically **pull** new events from the service.

5-2. NotificationService.publishNotifications()

This is the **core push process**:

```java
// src/.../NotificationService.java
@Transactional
public void publishNotifications() {
    // 1. Find the tracker for the last published message
    PublishedMessageTracker publishedMessageTracker =
        this.publishedMessageTracker();

    // 2. Fetch all new, unpublished StoredEvents from the Event Store
    List<Notification> notifications =
        this.listUnpublishedNotifications(
            publishedMessageTracker.mostRecentPublishedMessageId());

    // 3. Create a MessageProducer (e.g., for a RabbitMQ exchange)
    MessageProducer messageProducer = this.messageProducer();

    try {
        // 4. Push one by one
        for (Notification notification : notifications) {
            this.publish(notification, messageProducer);
        }
        // 5. Update the tracker with the ID of the last published StoredEvent
        this.trackMostRecentPublishedMessage(
            publishedMessageTracker, notifications);
    } finally {
        messageProducer.close();
    }
}
```

5-3. Fetching Unpublished Events

```
1  // in NotificationService
2  protected List<Notification> listUnpublishedNotifications(
3          long aMostRecentPublishedMessageId) {
4
5      EventStore eventStore = EventStore.instance();
6
7      List<StoredEvent> storedEvents =
8        eventStore.allStoredEventsSince(aMostRecentPublishedMessageId);
9
10     return this.notificationsFrom(storedEvents);
11 }
```

Directly calls ‘EventStore.allStoredEventsSince(...)‘ to get a sorted list of ‘StoredEvent‘s.

5-4. Synchronously Pushing a Single Notification

```
1  // in NotificationService
2  protected void publish(
3          Notification aNotification,
4          MessageProducer aMessageProducer) {
5
6      MessageParameters messageParameters =
7        MessageParameters.durableTextParameters(
8            aNotification.type(),
9            Long.toString(aNotification.notificationId()),
10           aNotification.occurredOn());
11
12     String notification = objectSerializer().serialize(aNotification);
13
14     aMessageProducer.send(notification, messageParameters);
15 }
```

Header used for deduplication and quick filtering, ‘notificationId‘ supports deduplication.

5-5. Establishing/Reconnecting a RabbitMQ Producer

```
1 // in NotificationService
2 private MessageProducer messageProducer() {
3     Exchange exchange = Exchange.fanOutInstance(
4         ConnectionSettings.instance(),
5         EXCHANGE_NAME,
6         true);
7
8     return MessageProducer.instance(exchange);
9 }
```

Rebuilds the connection each time to avoid long connection errors.

5-6. Periodically Triggering the Push

The book uses **JMX TimerMBean** to periodically trigger:

```
1  // Register MBean listener somewhere
2  mbeanServer.addNotificationListener(
3      timer.getObjectName(),
4      new NotificationListener() {
5          @Override
6          public void handleNotification(
7                  Notification aTimerNotification,
8                  Object aHandback) {
9
10             ApplicationServiceRegistry
11                 .notificationService()
12                 .publishNotifications();
13         }
14     },
15     null,
16     null);
```

5-7. RESTful Pull Style (Brief)

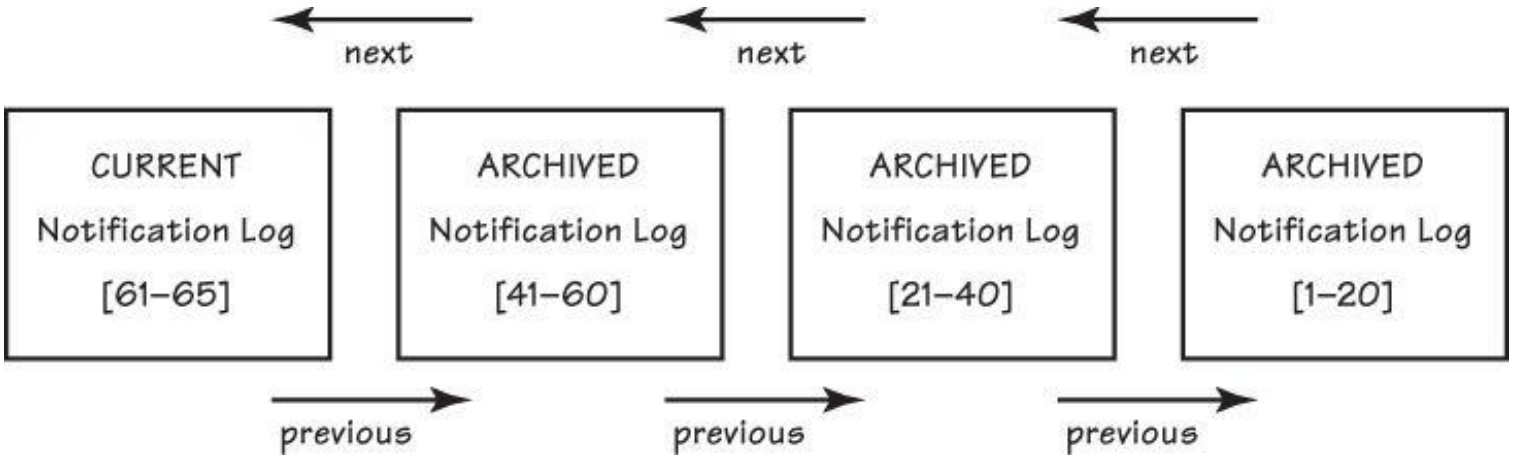
```java
@Path("/notifications")
public class NotificationResource {
    @GET @Produces(...)
    public Response getCurrentNotificationLog() {
        NotificationLog log =
          notificationService.currentNotificationLog();
        // Convert to REST DTO and return JSON
    }
    @GET @Path("{id}") ...
    public Response getNotificationLog(...) { ... }
}
```

Clients can periodically poll ‘/notifications‘ to pull the latest events.

5-8. Summary

- **Push**: ‘NotificationService‘ batch-pushes StoredEvents to RabbitMQ, using a Tracker to ensure exactly-once logic.
- **Pull**: Provide current & historical NotificationLogs via a JAX-RS Resource.
- **Latency Tolerance**: The event stream is eventually consistent and can tolerate delays.

Notification Log Pagination



6. Common Pitfalls & Testing Strategies

Common Pitfalls

- **Event handler modifies a second Aggregate**: Violates the "one transaction, one Aggregate" rule.
- **Failure to reset subscribers leads to leftovers**: The ‘ThreadLocal‘ subscriber list is not ‘reset()‘ at the start of a request.
- **Nested publishes are ignored**: The ‘publishing‘ flag skips subsequent ‘publish()‘ calls within a handler.
- **Lack of idempotency and deduplication**: A unique ID is needed for deduplication in cross-system communication.

Testing Strategy (1) - Publisher/Subscriber Unit Test

```
1  public class DomainEventPublisherTest {
2    @Before
3    public void setUp() {
4      DomainEventPublisher.instance().reset();
5    }
6
7    @Test
8    public void whenPublish_thenSubscriberReceivesEvent() {
9      AtomicReference<MyDomainEvent> received = new AtomicReference<>();
10     DomainEventPublisher.instance().subscribe(
11       new DomainEventSubscriber<MyDomainEvent>() {
12         public void handleEvent(MyDomainEvent e) { received.set(e); }
13         public Class<MyDomainEvent> subscribedToEventType() {
14           return MyDomainEvent.class;
15         }
16       }
17     );
18
19     MyDomainEvent evt = new MyDomainEvent(...);
20     DomainEventPublisher.instance().publish(evt);
21     assertSame(evt, received.get());
22   }
23 }
```

Testing Strategy (2) - Other Strategies

1. **Event Store (Outbox) Test**: Verify that serialization, ordering, and persistence are correct.
2. **Modeling Tests**: Test domain object behavior and invariants.
3. **Mocked Notification Forwarder Test**: Mock the ‘MessageProducer‘ to verify that events are ‘send()‘ and the tracker is updated.
4. **Idempotent Consumer Test**: Simulate sending the same event multiple times and assert that the handling logic is executed only once.

Conclusion

The above are common pitfalls and corresponding testing strategies for implementing Domain Events. They help you adhere to the single-Aggregate rule while ensuring the reliability and testability of your event mechanism.

Appendix: Deeper Dive

A. Event Identity

1. **When is it needed?**: For deduplication across systems, comparison, or when accessed as an Aggregate.
2. **Identity from event attributes**: Event name + Aggregate ID(s) + ‘occurredOn‘ timestamp is usually sufficient.
3. **Formally generating a unique ID**: If attributes are insufficient, a UUID can be assigned.
4. **Provided by message middleware**: Many MQs provide a unique message ID in the header.
5. **equals()/hashCode()**: Only need to be implemented if the event is stored as an Aggregate.

B. Modeling Events as Aggregates

When an event is created directly from a client request, it can be modeled as an Aggregate.

- **Immutable**: All state is injected during construction.
- **Unique Identity**: A specially generated ID.
- **Independent Storage**: First saved to a Repository, then published.

```
1  public class ClientTriggeredEvent implements DomainEvent {
2      private final EventId     eventId;
3      private final Date        occurredOn;
4      private final ImportantData data;
5      // Constructor fills all fields at once immutable
6      // getters...
7  }
8
9  public class EventCreationService {
10     public void createAndPublish(ImportantData d) {
11         ClientTriggeredEvent e =
12           new ClientTriggeredEvent(new EventId(), new Date(), d);
13         this.eventRepository.save(e); // Never delete
14         this.messagePublisher.publish(e); // Outbox or XA
15     }
16 }
```

C. Whiteboard Time Exercise

- **List existing but uncaptured events**:
  - User login failure, automatic order cancellation on timeout...
- **Consider model improvements**:
  - Can turning them into explicit Domain Events replace complex batch queries?
- **Focus on dependencies**:
  - Pay attention to scenarios that cross Aggregates and require eventual consistency.
- **Keep event behavior side-effect free**:
  - Only perform side-effect-free queries and data encapsulation.