Domain-Specific Query Pattern

This guide demonstrates how to wrap PeeGeeQ's `EventQuery` API with domain-specific query methods that use business language and maintain async/non-blocking behavior.

Overview

The domain-specific query pattern provides:

1. **Domain Language** - Methods named after business concepts instead of technical query operations
2. **Type Safety** - Strongly typed return values specific to your domain
3. **Async/Non-Blocking** - Returns `CompletableFuture` to avoid blocking the Vert.x event loop
4. **Encapsulation** - Hides query complexity behind simple method calls
5. **Testability** - Easy to mock for unit testing

The Pattern

❌ Anti-Pattern: Blocking the Event Loop

```
// DON'T DO THIS - blocks the event loop!
public List<BiTemporalEvent<TransactionEvent>> queryTransactionsByAccount(String accountId) {
    return eventStore.query(EventQuery.forAggregate(accountId)).join();  // ❌ BLOCKS!
}
```

✅ Correct Pattern: Async/Non-Blocking

```
// DO THIS - maintains async chain
public CompletableFuture<List<BiTemporalEvent<TransactionEvent>>> queryTransactionsByAccount(String accountId) {
    return eventStore.query(EventQuery.forAggregate(accountId));  // ✅ Non-blocking
}
```

Example: TransactionService

Basic Query Methods

```
@Service
public class TransactionService {

    private final EventStore<TransactionEvent> eventStore;

    /**
     * Domain-specific query: Get all transactions for a specific account.
     * Wraps EventQuery.forAggregate() with domain language.
     */
    public CompletableFuture<List<BiTemporalEvent<TransactionEvent>>>
```

```
            queryTransactionsByAccount(String accountId) {
        return eventStore.query(EventQuery.forAggregate(accountId));
    }

    /**
     * Domain-specific query: Get transactions by account and type.
     * Demonstrates the new EventQuery.forAggregateAndType() convenience method.
     */
    public CompletableFuture<List<BiTemporalEvent<TransactionEvent>>>
            queryTransactionsByAccountAndType(String accountId, String eventType) {
        return eventStore.query(EventQuery.forAggregateAndType(accountId, eventType));
    }
}
```

Convenience Methods

Build higher-level methods on top of the basic queries:

```
/**
 * Get only recorded (non-corrected) transactions for an account.
 */
public CompletableFuture<List<BiTemporalEvent<TransactionEvent>>>
        queryRecordedTransactions(String accountId) {
    return queryTransactionsByAccountAndType(accountId, "TransactionRecorded");
}

/**
 * Get only corrected transactions for an account.
 */
public CompletableFuture<List<BiTemporalEvent<TransactionEvent>>>
        queryCorrectedTransactions(String accountId) {
    return queryTransactionsByAccountAndType(accountId, "TransactionCorrected");
}
```

Composing Async Operations

Use `.thenApply()` and `.thenCompose()` to chain operations:

```
/**
 * Calculate account balance at a specific point in time.
 */
public CompletableFuture<BigDecimal> getAccountBalance(String accountId, Instant asOf) {
    return queryTransactionsByAccount(accountId)
        .thenApply(transactions -> {
            return transactions.stream()
                .filter(event -> !event.getValidTime().isAfter(asOf))
                .map(event -> {
                    TransactionEvent txn = event.getPayload();
                    return txn.getType() == TransactionType.CREDIT
                        ? txn.getAmount()
                        : txn.getAmount().negate();
                })
                .reduce(BigDecimal.ZERO, BigDecimal::add);
        });
}
```

EventQuery Convenience Methods

PeeGeeQ provides several static factory methods for common query patterns:

```
// Query all events
EventQuery.all()

// Query by event type
EventQuery.forEventType("OrderCreated")

// Query by aggregate ID
EventQuery.forAggregate("order-123")

// Query by aggregate ID AND event type (NEW!)
EventQuery.forAggregateAndType("order-123", "OrderCreated")

// Query as of a specific valid time
EventQuery.asOfValidTime(Instant.now())

// Query as of a specific transaction time
EventQuery.asOfTransactionTime(Instant.now())
```

Usage in Tests

When testing, call `.get()` on the `CompletableFuture` to wait for results:

```
@Test
void testDomainSpecificQueryMethods() throws Exception {
    // Record some transactions
    transactionService.recordTransaction(request1).get();
    transactionService.recordTransaction(request2).get();

    // Query using domain-specific methods
    List<BiTemporalEvent<TransactionEvent>> allTransactions =
        transactionService.queryTransactionsByAccount("ACC-001").get();

    assertEquals(2, allTransactions.size());

    List<BiTemporalEvent<TransactionEvent>> recordedOnly =
        transactionService.queryRecordedTransactions("ACC-001").get();

    assertEquals(2, recordedOnly.size());
}
```

Benefits

1. Domain Language

Instead of:

```
eventStore.query(EventQuery.forAggregateAndType("ACC-001", "TransactionRecorded"))
```

You write:

```
transactionService.queryRecordedTransactions("ACC-001")
```

2. Encapsulation

The service layer hides:

- Event type strings ("TransactionRecorded", "TransactionCorrected")
- Query builder complexity
- Async handling details

3. Type Safety

```
// Strongly typed - returns TransactionEvent, not generic Object
CompletableFuture<List<BiTemporalEvent<TransactionEvent>>> transactions =
    transactionService.queryRecordedTransactions("ACC-001");
```

4. Testability

Easy to mock in unit tests:

```
@Mock
private TransactionService transactionService;

when(transactionService.queryRecordedTransactions("ACC-001"))
    .thenReturn(CompletableFuture.completedFuture(mockTransactions));
```

Important: Aggregate ID

To use `EventQuery.forAggregate()` or `EventQuery.forAggregateAndType()`, you **must** set the `aggregateId` when appending events:

```
// ✅ CORRECT - includes aggregateId
eventStore.append("TransactionRecorded", event, validTime,
                Map.of(), null, accountId);  // accountId is the aggregate

// ❌ WRONG - aggregateId is null, queries won't work
eventStore.append("TransactionRecorded", event, validTime);
```

See Also

- `TransactionService.java` - Full implementation example
- `SpringBootBitemporalApplicationTest.java` - Test examples
- `EventQuery.java` - API reference for query building

Summary

The domain-specific query pattern:

1. Returns `CompletableFuture` (never blocks with `.join()`)
2. Uses business language for method names
3. Encapsulates query complexity
4. Maintains type safety
5. Enables easy testing and mocking