exactpro
EXITUS ACTA PROBAT

Integrated Framework for Automation in GUI Testing

Exactpro recommendations

February 2026

Executive Summary

Many Exactpro clients design, build, and operate dynamic, highly interactive applications using frameworks such as React, supported by Node.js based build and execution environments. This document presents practical recommendations illustrating how lightweight automated checkers can reveal issues that are difficult or costly to detect through other approaches.

If you are using Playwright or similar tools as part of your test implementation, these findings may be particularly relevant. To learn more about building automated checkers as a safety net, helping to improve software quality while accelerating delivery, please contact your Exactpro representative.

Introduction

If the purpose of software testing is to help businesses detect and solve the underlying issues that make the product less valuable to its users (and other stakeholders), UI testing should not be limited to pre-scripted testing, albeit automated. Full automation of a limited number of pre-scripted checks may, in fact, introduce risk because it creates an illusion of overconfidence: overreliance on scripted procedures takes charge in the context where 'part of the interaction is happening off-script'.[1]

Holistic UI testing should involve a combination of several scripted and non-scripted (exploratory) methods – an approach that is attuned to the complexity and the nature of the application under test. It is Exactpro's recommendation that the automated test library comprise artefacts from all exploration methods in the comprehensive testing approach. The approach should treat requirements and test artefacts as structured, linked data to ensure full traceability between them.

UI Testing Challenges

- Lack of Transparent Coverage Visibility, with no clear identification of what is covered, how it corresponds to acceptance criteria, or whether gaps in test coverage exist.
- Fragility of UI Automation – It is often unclear whether the current test library monitors and detects errors when UI, data or workflows change. GUI, API, data validation and log analysis are typically tested in isolation. This fragmented approach fails to reflect how the system behaves as an integrated whole and allows cross-layer defects to escape detection.
- Uncontrolled growth of test suites, making them costly to maintain and slow to execute.
- Usage of BDD assets that are directly translated into executable UI scripts and do not scale into reliable automation.

[1] Bach, J., Bolton, M. (2025). Taking Testing Seriously. How Testing Looks to Management (chapter co-written by Keith Klain). p544

- Focus on step-by-step UI interactions while the system's state changes independently. This validates mechanics rather than behaviour and often misses defects related to system state transitions, data consistency, or side effects across the application.
- Absence of Property-Based Validation. Most UI tests typically validate individual scenarios in isolation, resulting in large numbers of similar tests and limited ability to detect systemic issues, patterns, or inconsistencies.

From Fragmented BDD Scripts to Provable System Coverage

Many organisations attempt to automate UI testing by directly translating BDD acceptance criteria into executable end-to-end GUI tests.
While this approach is attractive from a business-readability perspective, in practice it often leads to fragile, slow and hard-to-scale test automation.

This observation should not be interpreted as a limitation of BDD itself. BDD remains a valuable technique for expressing expected system behaviour in a structured and human-readable form. When BDD scenarios are treated primarily as execution scripts rather than behavioural specifications, their scalability and diagnostic value are reduced.

In this approach, BDD scenarios are preserved and reused as specification artefacts. They are mapped onto workflow models and property-based checks, ensuring traceability while avoiding step-by-step execution.

Exactpro Recommendations

Based on Exactpro's experience, we recommend following a combined test automation approach, which includes **Model-based** and **Property-based testing** approaches. As per the approach, the Exactpro team will:

- Optimise existing scenarios into templates that capture behaviour rather than UI steps.
- Use the templates to construct a workflow model enriched with sequences of inputs, actions, conditions, outputs, data flows and annotations.
- Maintain explicit traceability between requirements, BDD assets, workflow elements, and automated checks. Use annotations to map workflows to requirements and acceptance criteria, ensuring clear traceability across all test artefacts.
- Execute validation primarily through the workflows created based on the model, followed by property-based and consistency checks rather than directly scripted UI flows.
- Develop an integrated holistic testing pipeline with a scalable, automation architecture.

The combined approach allows the same behavioural intent to be captured as BDD specifications while enabling scalable, robust automation and comprehensive, traceable coverage.

The **Model-based testing** approach consists in creating a behavioural model (a ‘digital twin’) of the system and automatically deriving test scenarios from it, ensuring systematic coverage, reduced manual effort and higher confidence in complex workflows.
The **Property-based testing** workflow is described in more detail in the remainder of the document.

Property-Based Testing (PBT)

Traditional ‘example-based’ scripts validate a limited set of predefined scenarios (‘**If I input 2+2, I expect the result to be 4**’). In contrast, **property-based testing defines invariants that** must always hold across all valid inputs and interaction sequences. Property-based automated checks exercise broad and unpredictable input spaces in order to identify violations of these invariants. The approach can be viewed as the application of a consistency oracle that compares the expected results with observable UI outputs..

This shifts the focus **from checking individual pages and widgets to consistency ‘rules’ (properties)** that apply across entire classes of UI ‘entities’ (such as ‘grids’, ‘buttons’, or ‘tabs’) and across the full system lifecycle.

Property-Based Testing – example

The setup: instead of a script explicitly stating ‘**Go to the “Trades” tab and check the grid**’, the tool programmatically scans the DOM to discover all elements identified as a ‘grid’, regardless of their location.

The verification: instead of verifying a hardcoded value like ‘**The first row must be “Ticker AAPL”**’, the tool enforces a logic property. For example: ‘**For every grid found, the row count in the UI must exactly match the row count in the database or export file**’.

The invariant examples (properties that must hold):

- Instead of a simple check to see if a specific button works, the tool monitors global invariants. For example: ‘**No matter what element is clicked or what action is performed, the browser console error count must always remain 0.**’
- A ‘**refresh**’ check that applies a ‘State Preservation’ property. The rule enforces that the application state immediately before a refresh must be identical to the state immediately after the refresh (State_Before_Refresh == State_After_Refresh).
- The ‘**Grid vs. Export**’ check – the property being tested is that UI_Data must always equal Export_Data.

Property-based Testing focuses on:

- rules that must not be violated
- large, unpredictable input spaces
- state transitions over time

By exercising these properties across varying interaction sequences and timing conditions, this approach is effective at detecting:

- rendering race conditions
- edge-value formatting issues
- state desynchronisation after long interaction sequences
- rare timing windows (fast market + user actions)
- systemic issues across UI, data, and integrations

The property-based testing methodology is highly effective for financial applications because it directly addresses market consistency, data integrity and stability.

Property-based Testing Components of the Integrated GUI Test Automation Framework

The challenges described above are addressed by an integrated automation framework that is part of Exactpro's th2 suite. The framework combines UI-level observation, protocol-level analysis, and backend reconciliation to provide holistic, system-level coverage. It consists of the following complementary components:

- **UI Interactor**
  Performs controlled UI interactions and captures UI state artefacts such as screenshots, DOM snapshots, and interaction traces. These artefacts provide a factual representation of observed system behaviour.

- **UI Checker** (UI Consistency Checker)
  Validates visual, structural, and semantic consistency of UI artefacts. The checker focuses on behavioural correctness and state representation rather than UI mechanics.

- **Protocol Property Checker** (PPC)
  Observes REST and WebSocket traffic and enforces protocol-level properties and invariants. This component detects discrepancies between client-visible behaviour, message flows and timing.

- **Backend Checker** (Backend Errors Validator and Reconciliation Engine)
  Analyses backend logs, database state, and alerts and performs reconciliation between frontend observations, protocol traffic, and persisted backend state.

Benefits of Exactpro’s Recommended Integrated GUI Test Automation Framework

- Reuses existing ‘example-based’ test assets / BDD assets as behavioural specifications and a foundation for workflow modeling, enabling rapid generation of test batches that cover broad classes of system behaviour.
- Makes Playwright or a Playwright-like UI automation usable at scale through rule-based validation and reduced maintenance overhead.
- Provides transparent, auditable reporting, where existing test assets and acceptance criteria are traceably linked to the generated test library monitoring issues across frontend, protocol and backend layers.
- Is well-suited for complex, data-heavy and regulated platforms, where consistency and traceability are critical.
- Reduces the need for manual maintenance of hundreds of tests due to the use of model-based and property-based testing (i.e. seeing the GUI under test as a number of rule-based properties).

Examples of issues detected by the Property-based testing component of the framework are provided in the **Appendix**.

Contact us

Our integrated framework for automation in GUI Testing turns scripted validation into a systematic, comprehensive and scalable testing practice.

Reach out via info@exactpro.com to book an introductory call.

Appendix

This section lists defects detected by a single automated checker (a single element of the Integrated Framework for Automation in GUI Testing). The results presented below were obtained by analysing the web front end of a React-based CRM system after a workflow simulation performed by robots that followed model state transitions.

We are therefore pleased to share these findings as a practical illustration of how automated checkers can uncover issues that are difficult or costly to detect using other testing approaches.

Resolved Issues

1. Market Prices - Decimal Rounding

Observed: Price showed 0.437 in grid before page refresh, 0.440 after page refresh
Root Cause: Database columns were defined with insufficient precision

```
-- Before: Only 2 decimal places
last_price DECIMAL(12, 2)  -- 0.437 → rounds to 0.44
```

Fix: Increased precision to 4 decimal places in V1__baseline.

```
-- After: 4 decimal places
last_price DECIMAL(14, 4)  -- 0.437 → stored as 0.4370
```

Affected columns: last_price, price_change, prev_close, trade_value, product_volume, open_price, high_price, low_price

2. Timestamps - Nanosecond vs Microsecond Precision

Observed: *updatedAt* field showed different values in grid before page refresh, after page refresh (e.g., 2024-01-15T10:30:00.123456789Z vs 2024-01-15T10:30:00.123456Z)

Root Cause: Precision mismatch between Java and PostgreSQL

- Java *Instant.now()* has **nanosecond** precision
- PostgreSQL *TIMESTAMP* has **microsecond** precision
- WebSocket broadcast sent the Java value (before Database save)
- REST API returned the database value (truncated)

```
// Before: Nanosecond precision
entity.setUpdatedAt(Instant.now());  // 2024-01-15T10:30:00.123456789Z
// After DB save: 2024-01-15T10:30:00.123456Z (truncated)
```

Fix: Created *TimeUtils.nowTruncated()* utility that truncates to microseconds (matching *PostgreSQL* precision)

```
// After: Consistent microsecond precision
entity.setUpdatedAt(nowTruncated());  // 2024-01-15T10:30:00.123456Z
```

Affected files:
All service classes that set updatedAt:

- ParticipantService.java
- CustomerService.java
- UserService.java
- ApprovalService.java
- DocumentService.java
- ProductTokenService.java
- MarketPriceService.java
- ContractService.java
- ProductStateService.java

3. Products Ownership - Missing Participant/Product Names

Observed: In the “Products Ownership” tab, WebSocket updates showed participant IDs (e.g., *123*) instead of names (e.g., "*Acme Corp*"), while REST API returned proper names after page refresh.

Root Cause: *ActivityBook.java* was creating/converting *ActivityPosition* objects without populating the *participantName* and *productTokenName* fields for WebSocket broadcasts.

```
// Before: Only IDs were set
private ActivityPosition entityToPosition(ActivityPositionEntity entity) {
   position.setParticipantId(entity.getParticipantId());
   position.setProductTokenId(entity.getProductTokenId());
   // participantName and productTokenName were null!
}
```

Fix: Updated *entityToPosition()* and *getOrCreatePosition()* to populate names using existing lookup methods:

```
// After: Names are populated
position.setParticipantName(getParticipantName(entity.getParticipantId()));
position.setProductTokenName(getProductTokenName(entity.getProductTokenId()));
```

Affected files:
- *ActivityBook.java* - entityToPosition() and getOrCreatePosition() methods
- *ActivityPosition.java* - Also updated to use nowTruncated() for timestamps

4. Contracts - WebSocket Payload Date Conversion

Observed: Contracts showed different timestamp formats before page refresh, after page refresh (e.g., 2026-02-03**T**03:20:31.**923079Z** vs 2026-02-03 03:20:33).

Root Cause: WebSocket payloads were not going through the same date conversion as REST API responses.
- REST API: Generated client uses *ContractFromJSON()* which converts ISO strings to Date objects
- WebSocket: Raw *JSON.parse* kept dates as strings, then type-cast to ApiContract

```
// Before: Type cast doesn't convert dates
const updatedContract = this.mapContract(message.payload as ApiContract);
```

Fix: Use ContractFromJSON() to properly convert WebSocket payloads:

```
// After: Proper date conversion
const apiContract = ContractFromJSON(message.payload);
const updatedContract = this.mapContract(apiContract);
```

Affected files:
- ContractStore.ts - handleContractUpdate() method

Note: Time differences (e.g., 31 vs 33 seconds) may still occur if background schedulers (ProductAcquisitionScheduler, CashPaymentRobot) update Contracts between the WebSocket broadcast and browser refresh.
This is expected behavior, not a bug.

5. Contracts - Missing Customer/Participant Names in WebSocket Broadcasts

Observed: WebSocket updates for Contracts showed empty "-" values for Customer, Issuer, Originating Bank, and Liquidity Provider names, while REST API returned proper names after page refresh.

Root Cause: ContractService.java had two toModel methods:

1. *toModel(ContractEntityView)* - Used by REST API, includes all names from database view
2. *toModel(ContractEntity, String, String)* - Used by WebSocket broadcasts, only included productToken and currencyCode names

```
// Before: Used entity-based toModel without names
Contract createdContract = toModel(savedContractEntity, productTokenName, currencyCode);
broadcastContractCreated(createdContract);
```

Fix: Fetch the full view with names after saving, then use for both broadcast and response:

```
// After: Fetch view with all names
Contract createdContract =
contractRepository.findContractEntityViewByContractId(savedContractEntity.getContractId())
        .map(this::toModel)
        .orElseThrow();
broadcastContractCreated(createdContract);
```

Affected files:

- ContractService.java - createContract0() and amendContract0() methods

6. Market Prices - New Market Prices Not Broadcast via WebSocket

Observed: After page refresh, new market prices (e.g., AKIC) appeared with default/zero values (lastPrice=1, everything else 0). These were absent from the "before refresh" data, causing row count and value mismatches.

Root Cause: *ProductTokenService.createMarketPriceForProductToken()* saved new market prices to the database but did NOT broadcast them via WebSocket. When the Python test script created new product tokens (Step 1), the associated default market prices were invisible to the frontend until a page refresh.

```
// Before: Market price saved but not broadcast
marketPriceRepository.save(marketPrice);
LOG.debug("Created market price for product token {}", productTokenId);
```

Fix: After saving the market price entity, build a MarketPrice model and broadcast it as a 'created' event. Also updated the frontend MarketPriceStore to handle 'created' events and use *MarketPriceFromJSON()* for proper date conversion.

```
// After: Market price saved and broadcast
MarketPriceEntity savedEntity = marketPriceRepository.save(marketPriceEntity);
broadcastMarketPriceCreated(savedEntity, productTokenName, currency.getCurrencyCode());
```

```
// Frontend: Handle both 'created' and 'updated' events with proper conversion
if ((message.event === 'created' || message.event === 'updated') && message.payload) {
  const price = MarketPriceFromJSON(message.payload);
  this.updatePrice(price);
}
```

Affected files:

- ProductTokenService.java - createMarketPriceForProductToken() method
- WebSocketMessage.java - Added marketPriceCreated() factory method
- MarketPriceStore.ts - Added 'created' event handling + MarketPriceFromJSON() conversion

7. All Tabs - Inconsistent Timestamp Display and Conversion

Observed: Timestamps showed different formats before and after refresh across multiple tabs (Users, Approvals, Participants, Quotes, etc.).

Examples:

- WebSocket: "2026-02-03T09:29:28.569276Z" (raw ISO string, not converted to Date)
- After refresh: "Tue Feb 03 2026 09:29:28 GMT+0000 (Greenwich Mean Time)" (Date.toString())

Root Cause: Three compounding issues:

1. **8 stores used** as **Type casting** instead of *FromJSON() for WebSocket payloads. This left date fields as raw ISO strings instead of Date objects. Affected: CustomerStore, UserStore, ApprovalStore, ParticipantStore, QuoteStore, QuoteHistoryStore, TransactionStore, ActivityPositionStore.
2. **All 15 table files used cellRenderer** for timestamp columns instead of valueFormatter. AG Grid uses valueFormatter for both grid display and CSV export, but cellRenderer only affects display. This caused CSV exports to use the raw value (Date.toString() or ISO string) instead of the formatted output.
3. **Inconsistent format function:** The shared toTimestampString() used toLocaleString(), which produces locale-dependent output (e.g., "2/3/2026, 9:29:28 AM"). Two stores (ContractStore, TransactionStore) also pre-formatted dates differently in their mapping functions.

Fix:

1. Changed toTimestampString() from toLocaleString() to produce consistent UTC format "YYYY-MM-DD HH:mm:ss":

```
// Before: locale-dependent
return value.toLocaleString();
```

```
// After: consistent UTC, seconds precision
return date.toISOString().replace('T', ' ').split('.')[0];
```

2. Added timestampValueFormatter and switched all 15 table files from cellRenderer: timestampCellRenderer to valueFormatter: timestampValueFormatter.

3. Fixed all 8 stores to use *FromJSON() for WebSocket payloads:

```
// Before: dates stay as strings
const newUser = message.payload as UserProfile;
// After: dates properly converted to Date objects
const newUser = UserProfileFromJSON(message.payload);
```

4. Fixed pre-formatting in mappers:

```
// TransactionStore - before: ugly Date.toString() output
updatedAt: tx.updatedAt ? String(tx.updatedAt) : '',
// After: proper ISO string for valueFormatter to parse
updatedAt: tx.updatedAt instanceof Date ? tx.updatedAt.toISOString() : ...
```

Affected files:

- cellRenderers.tsx - toTimestampString(), added timestampValueFormatter
- 8 store files - Added *FromJSON() imports and usage in WebSocket handlers
- 15 table files - Changed cellRenderer → valueFormatter for timestamp columns
- ContractStore.ts, TransactionStore.ts - Fixed date pre-formatting in mappers

8. Activity Transactions - Missing Participant/Product Names in Service entityToTransaction()

Observed: Activity transactions could show participant IDs instead of names if served through ActivityBook.getTransactionsByParticipant().

Root Cause: *ActivityBook.java* had two code paths for creating ActivityTransaction objects:

1. recordTransaction() - Used for WebSocket broadcasts, properly enriched with participant/product names via getParticipantName() and getProductTokenName()
2. entityToTransaction() - Used by getTransactionsByParticipant(), was missing name enrichment

Meanwhile, ActivityBookController.java had its own entityToTransaction() that correctly included name enrichment (used by the REST GET endpoint).

```
// Before: Missing name fields
private ActivityTransaction entityToTransaction(ActivityTransactionEntity entity) {
  tx.setFromParticipantId(entity.getFromParticipantId());
```

```
    // fromParticipantName, toParticipantName, productTokenName were NOT set
}
```

Fix: Updated ActivityBook.entityToTransaction() to match the controller's implementation, adding name enrichment:

```
// After: Names populated
tx.setProductTokenName(getProductTokenName(entity.getProductTokenId()));
tx.setFromParticipantName(getParticipantName(entity.getFromParticipantId()));
tx.setToParticipantName(getParticipantName(entity.getToParticipantId()));
```

Affected files:

- ActivityBook.java - entityToTransaction() method

9. Customers - Duplicates After Refresh, Missing Rows in Real-time

Observed: After page refresh, the Customers grid showed duplicate rows (same customer appearing multiple times). Real-time WebSocket updates showed fewer customers than refresh.

Root Cause: The REST endpoint used a LEFT JOIN between customer and contract tables, returning one row per customer-contract pair. A customer with 3 active (non-DONE) contracts produced 3 rows. Meanwhile, WebSocket broadcasts sent plain Customer objects (one per customer, no contractId).

```
-- Before: LEFT JOIN produced duplicates
SELECT c.*, s.contract_id
FROM customer c
LEFT JOIN contract s ON c.customer_id = s.customer_id AND s.contract_status NOT IN
('DONE')
```

The contractId field wasn't even displayed in the grid columns, so the duplicates appeared as identical rows. The AG Grid getRowId used a composite key ${customerId}-${contractId} to accommodate the duplicates, masking the underlying issue.

Fix: Removed the LEFT JOIN. The REST endpoint now queries the customer table directly via Micronaut Data's findAll(pageable) and findByBankId(bankId, pageable), returning one row per customer — matching what WebSocket sends.

```
// Before: View query with LEFT JOIN
customerRepository.findAllCustomerEntityView(pageable).map(this::toModel)
```

```
// After: Direct table query
customerRepository.findAll(pageable).map(this::toModel)
```

Frontend getRowId simplified from composite key to plain customerId.

Affected files:

- CustomerRepository.java - Removed LEFT JOIN query; added findByBankId() method
- CustomerService.java - Switched to findAll()/findByBankId(); removed toModel(CustomerEntityView) overload
- CustomerTable.tsx - Simplified getRowId to use only customerId

10. Product Ownership - More Rows After Refresh Than Real-time

Observed: The Product Ownership grid showed more rows after refresh than received via real-time WebSocket updates.

Root Cause: The REST endpoint queried the participant_position table, but **no Java code ever wrote to that table** — it was a dead table. WebSocket events came from ActivityBook which operates on the activity_position table. These are completely separate tables tracking different data.

The frontend store subscribed to the activity-positions WebSocket topic and refetched via REST on each event. But since REST read from the wrong table, the data never reflected actual activity changes. Additionally, the old query used LEFT JOIN from participant to participant_position, producing rows with NULL product tokens for participants with no positions.

```
-- Before: Read from dead participant_position table with LEFT JOIN
SELECT p.participant_name, p.participant_type, a.product_token_name, sum(pp.product_qty)
FROM participant p
LEFT JOIN participant_position pp ON p.participant_id = pp.participant_id
LEFT JOIN product_token a ON pp.product_token_id = a.product_token_id
GROUP BY p.participant_name, p.participant_type, a.product_token_name
```

Fix: Rewrote the query to read from activity_position — the table that ActivityBook actually writes to and broadcasts WebSocket events for. Changed from LEFT JOIN to INNER JOIN, filtered out zero balances, and aggregated with GROUP BY + SUM() to collapse multiple accounts per participant+product into a single row (the activity_position table has one row per (participant_id, account, product_token_id) combination).

```
-- After: Read from the live activity_position table, aggregated across accounts
SELECT p.participant_name, p.participant_type, a.product_token_name, sum(lp.balance)
FROM activity_position lp
INNER JOIN participant p ON lp.participant_id = p.participant_id
INNER JOIN product_token a ON lp.product_token_id = a.product_token_id
WHERE lp.balance > 0
GROUP BY p.participant_name, p.participant_type, a.product_token_name
```

Affected files:

- ParticipantPositionRepository.java - Rewrote SQL query to use activity_position table with aggregation

11. Quote History - Participant IDs Instead of Names in Real-time

Observed: Quote History grid showed raw participant IDs (e.g., 1, 2) in the Taker/Maker columns for real-time WebSocket updates. After refresh, proper human-readable names appeared (e.g., "Bank of Example").

Root Cause: Neither QuoteBookService.broadcastHistoryUpdate() nor QuoteBookQuoteController.entityToHistoryEntry() populated takerParticipantName and makerParticipantName on the Quote model — only the IDs were set. The frontend compensated with a client-side participantNamesById lookup map, but this was fragile and didn't always resolve correctly.

Meanwhile, the active quotes REST endpoint (GET /quote-book/quotes) had a separate enrichQuotesWithParticipantNames() method that resolved names — but this enrichment was never applied to quote history entries or WebSocket broadcasts.

```
// Before: Only IDs set, no names
Quote quote = new Quote();
quote.setTakerParticipantId(entity.getTakerParticipantId());
// takerParticipantName never set!
quote.setMakerParticipantId(entity.getMakerParticipantId());
// makerParticipantName never set!
```

Fix: Added enrichParticipantNames() method to QuoteBookService that resolves participant names from the database. Applied it to:

1. broadcastHistoryUpdate() — WebSocket broadcasts for quote history
2. entityToQuote() — shared method used by both REST (entityToHistoryEntry) and WebSocket
3. broadcastQuoteCreated() / broadcastQuoteUpdated() — WebSocket broadcasts for active quotes

```
// After: Names resolved before broadcast/response
private void enrichParticipantNames(Quote quote) {
  if (quote.getTakerParticipantId() != null) {
    Long takerId = Long.parseLong(quote.getTakerParticipantId());
    participantRepository.findById(takerId)
        .ifPresent(p -> quote.setTakerParticipantName(p.getParticipantName()));
  }
```

```
    // same for maker
}
```

Affected files:

- QuoteBookService.java — Added ParticipantRepository dependency, enrichParticipantNames() helper, entityToQuote() shared method; enrichment applied to all broadcast methods
- QuoteBookQuoteController.java — entityToHistoryEntry() now delegates to quoteBookService.entityToQuote()

12. Contract Transactions - Status=PROCESSING in Real-time, COMPLETED After Refresh

Observed: The Transactions grid showed PROCESSING status for newly created transactions via WebSocket real-time updates. After refresh, the same transactions showed COMPLETED.

Root Cause: createWorkflowTransaction() saved the transaction with PROCESSING status and **immediately broadcast** it via WebSocket. Then the caller (executeXXX() method) updated the status to COMPLETED and saved to the database — **but never broadcast the update**.

Sequence:

1. createWorkflowTransaction() → save(status=PROCESSING) → broadcast(PROCESSING) ← WebSocket sees this
2. executeXXX() → set status=COMPLETED → update in DB ← no broadcast!
3. REST refresh → query DB → returns COMPLETED ← REST sees this

```
// Before: broadcast in createWorkflowTransaction() with PROCESSING
ContractTransactionEntity savedTx = contractTxRepository.save(tx);  // status=PROCESSING
broadcastContractTransactionCreated(toTransactionModel(savedTx));          //      broadcast PROCESSING
return savedTx
// Then in executeXXX():
tx.setContractTransactionStatus(ContractTransactionStatus.COMPLETED);
contractTxRepository.update(tx);  // no broadcast after this!
```

Fix: Removed the broadcast from createWorkflowTransaction() and added a finalizeTransaction() helper that both updates the DB and broadcasts. All 32 contractTxRepository.update(tx) calls in the execute methods were replaced with finalizeTransaction(tx), ensuring the transaction is always broadcast with its final status (COMPLETED or FAILED).

```
// After: broadcast only when final status is set
private void finalizeTransaction(ContractTransactionEntity tx) {
  contractTxRepository.update(tx);
  broadcastContractTransactionCreated(toTransactionModel(tx));
}

// In createWorkflowTransaction(): no broadcast (just save with PROCESSING)
// In executeXXX(): finalizeTransaction(tx) broadcasts COMPLETED/FAILED
```

Affected files:

- ContractService.java — Removed broadcast from createWorkflowTransaction(); added finalizeTransaction() helper; replaced all 32 contractTxRepository.update(tx) calls with finalizeTransaction(tx)

13. Quotes - Cancelled Quotes Not Removed in Real-time

Observed: After *CANCEL_ALL_QUOTES*, quotes RFQ-21 and RFQ-22 remain in the grid (22 rows). After refresh, only 20 rows — cancelled quotes are gone.

Root cause: *QuoteBookService.cancelAllQuotes()* set quote state to CANCELLED and recorded history, but did NOT broadcast quoteDeleted events via WebSocket. The frontend QuoteStore never received notification to remove the cancelled quotes.

Fix: Added broadcastQuoteDeleted(quote.getQuoteId()) call inside the cancelAllQuotes() loop, after persisting and recording history.

Files changed:

- crmtest-server/.../service/QuoteBookService.java — added broadcast in cancelAllQuotes()

14. Contract Transactions - Missing From/To Participant Names in Real-time

Observed: All transactions show empty **From** and **To** columns in real-time. After page refresh, participant names appear (e.g., "Abu Dhabi Islamic Bank" → "Sharjah Islamic Bank").

Root cause: *ContractService.toTransactionModel(ContractTransactionEntity)* (used for WebSocket broadcasts) only set sourceParticipantId/targetParticipantId without resolving names. The REST endpoint used *toTxModel(ContractTransactionView)* which got names from a JOIN database.

Fix: Added ParticipantRepository.findById() lookups in toTransactionModel() to resolve sourceParticipantName and targetParticipantName before broadcasting.

Files changed:

- crmtest-server/.../service/ContractService.java — enriched toTransactionModel() with participant name resolution

15. Participant Positions (Activity) - Missing Zero-Balance Rows in Real-time

Observed: 105 rows in real-time vs 110 after page refresh. 5 MAIN account positions with balance=0 were missing from real-time data.

Root cause: ActivityPositionStore.handleRealtimeUpdate() matched positions by (participantId, productTokenId), which is NOT unique when the same participant holds the same product in multiple accounts (MAIN and ACC-xxx). During contract processing, a participant's MAIN account briefly holds tokens before transferring to the contract account. The MAIN position update would overwrite the wrong row (the ACC-xxx row found first by findIndex), causing the MAIN position to never appear as a separate row.

Fix: Changed the findIndex key from (participantId, productTokenId) to activityPositionId, which is unique per position row.

Files changed:

- crmtest-spa/.../activityPositions/store/ActivityPositionStore.ts — use activityPositionId for matching

16. Quote History - Participant Names Show "-" Due to Frontend Lookup Race

Observed: Taker/Maker columns show "-" in before-refresh CSV, actual names after refresh

Root Cause: QuoteHistoryTable.tsx used participantNamesById[String(takerId)] lookup map exclusively. The lookup map is populated by fetchParticipants() which runs async without setting isLoading, creating a race where CSV export can happen before the map is populated.

The backend already enriches quote.takerParticipantName / quote.makerParticipantName via enrichParticipantNames() in both REST and WebSocket paths, but the table ignored these fields.

Fix: Changed valueGetter to prefer quote.takerParticipantName / quote.makerParticipantName from the backend-enriched data, falling back to the lookup map only when the name field is absent.

Files Changed: crmtest-spa/src/features/quoteHistory/components/QuoteHistoryTable.tsx

17. Product Management - Timestamps Always Show Current Time Instead of Database Value

Observed: All 90 rows differ only in timestamp (e.g., 21:26:22 vs 21:26:35)

Root Cause: ProductStateService.toModel() set updatedAt to nowTruncated().atZone(ZoneOffset.UTC) — the current time at response-build time — instead of the database entity's actual updated_at. The ProductSummaryView record and its SQL query didn't include the updated_at column at all.

Fix: Added updatedAt field to ProductSummaryView record, added c.updated_at as updated_at to the SQL query, changed toModel() to use entity.updatedAt() instead of nowTruncated().

Files Changed: ProductSummaryView.java, ProductStateRepository.java, ProductStateService.java

Investigation Checklist

When a new discrepancy is found:

1. **Identify the field(s)** with different values
2. **Check database schema** - Is precision sufficient? (DECIMAL(x, y))
3. **Check timestamp precision** - Is nowTruncated() being used instead of Instant.now()?
4. **Check entity-to-model conversion** - Are all fields populated (especially names/labels from related entities)?
5. **Check WebSocket broadcast** - Is it sending the saved entity or the input?
6. **Check API response** - Is it transforming/formatting data differently?
7. **Check frontend store** - Is the WebSocket handler updating state correctly?
8. **Check WebSocket payload conversion** - Is *FromJSON() being used to convert dates properly?
9. **Check CSV export** - Is valueFormatter vs raw value causing display differences?

10. **Check side-effect creation** - When creating an entity triggers creation of related entities (e.g., product token → market price), are the related entities also broadcast via WebSocket?
11. **Check for LEFT JOINs in REST queries** - Do they produce duplicate rows (one per joined record)? Is the joined data even displayed in the grid?
12. **Check REST and WebSocket read the same table** - Are they backed by the same database table, or different tables that drift apart?

Related Files

- Test script: crmtest-spa/tests/download-csv-test.spec.ts
- Database migrations: crmtest-server/src/main/resources/db/migration/
- WebSocket broadcaster: crmtest-server/src/main/java/.../websocket/WebSocketBroadcaster.java
- Activity book service: crmtest-server/src/main/java/.../service/ActivityBook.java
- Frontend stores: crmtest-spa/src/features/*/store/
- Cell renderers/formatters: crmtest-spa/src/components/common/cellRenderers.tsx
- Time utilities: crmtest-server/src/main/java/.../util/TimeUtils.java