Change request log 1

1. Concept Location

Step	Description	Rationale
1	We ran the system.	-
2	We interacted with the system: after logging in, we entered the home screen.	To get familiar with some of the features of the system and identify the screens or graphical elements we had to change.
3	We examined the point list table and tried to find a way to access its details.	To determine how the data is displayed and where modifications are needed.
4	After finding a button to the right of the table, we clicked it, which opened the Point Details page. We searched for "Point Details Table" to locate the section requiring changes.	Because we identified a button on the screen labeled "Point Details."
5	We opened the Network tab in the browser's developer tools to identify which API was being called to fetch the details table data.	To trace the backend request responsible for populating the table.
6	We searched for the keyword 'Annotation' in Visual Studio Code using Cmd + Shift + F, as it seemed uncommon and could help locate the UI code.	This led us to the table's code inside dataPointDetails.jsp.
7	We identified the table ID as historyTableData, which was found inside a function named getHistoryTableData.	-
8	We performed a global search for getHistoryTableData in the codebase.	This search led us to the DataPointDetailsDwr.java file.
9	We marked the class DataPointDetailsDwr as "located".	We confirmed that this class had to be modified.

Time spent (in minutes): 65

Classes and methods inspected:

- /mango/mangoSource/war/WEB-INF/jsp/dataPointDetails.jsp
  - <table> below </script> tag
- /mango/mangoSource/src/com/serotonin/mango/web/dwr/DataPointDetailsDwr.java
  - function getHistoryTableData()

2. Impact Analysis

Step	Description	Rationale
1	We made a list of methods called by DataPointDetailsDwr.getHistoryTableData(int limit).	To track the classes that could be impacted by the change.
2	We inspected PointValueFacade.getLatestPointValues(int limit).	This method retrieves historical data points, and any change in its logic might affect data retrieval.
3	We examined RenderedPointValueTime.setValue(String value) and RenderedPointValueTime.setAnnotation(String annotation).	These methods format and store values and annotations that are later displayed in the UI.
4	We analyzed Functions.getHtmlText(DataPointVO pointVO, PointValueTime pvt).	This function determines how values are displayed in the UI and may need changes if the formatting requirements are updated.
5	We checked DwrResponseI18n.addData("history", renderedData).	This ensures that the modified data structure is correctly sent back to the frontend.
6	We examined dataPointDetails.jsp for the <table id="historyTableData"> element.	The UI needs to reflect the processed data correctly, so this file must be updated if the structure changes.

Time spent (in minutes): 30

Classes and Methods Inspected :

- **DataPointDetailsDwr.java**
  - DwrResponseI18n getHistoryTableData(int limit)
- **PointValueFacade.java**
  - List<PointValueTime> getLatestPointValues(int limit)
- **RenderedPointValueTime.java**
  - void setValue(String value)
  - void setTime(long time)
  - void setAnnotation(String annotation)
- **Functions.java**
  - static String getHtmlText(DataPointVO pointVO, PointValueTime pvt)
  - static long getTime(PointValueTime pvt)
- **dataPointDetails.jsp**
  - <table id="historyTableData">

3. Prefactoring (optional)

Step	Description	Rationale
1	We extracted the logic for truncating numeric values into a separate method called truncateValue(String value).	Using the **"Extract Method"** refactoring improves code readability and reusability. This ensures that truncation logic is maintained separately, making it easier to modify or reuse elsewhere.
2	We modified RenderedPointValueTime.setValue() to use truncateValue(Functions.getHtmlText(pointVO, pvt)).	This ensures that numeric values displayed in the UI are consistently truncated to two decimal places.
3	We ran unit tests for getHistoryTableData(int limit) to verify that the truncation logic works correctly.	Ensures that the new method does not introduce unintended side effects.
4	We manually tested the UI by navigating to the history table screen and verifying the rendered values.	Confirms that the front-end displays truncated values as expected.
5	We committed our changes to Git with a descriptive commit message.	Preserves our progress and allows easy rollback if needed.

Time spent (in minutes): 15

Classes and Methods Modified:

- DataPointDetailsDwr.java
  - Modified: DwrResponseI18n getHistoryTableData(int limit)
  - Added: private String truncateValue(String value)

4. Actualization

Step	Description	Rationale
1	We modified the method getHistoryTableData(int limit) in DataPointDetailsDwr.java to use the newly introduced truncateValue(String value) function.	Ensures that numerical values are truncated to two decimal places before being displayed in the UI.
2	We added the method truncateValue(String value) in DataPointDetailsDwr.java.	Extracting the truncation logic into a separate method improves maintainability and prevents redundant code.
3	We created and ran unit tests to validate truncateValue(String value).	Ensures the correctness of the truncation logic and prevents regressions.
4	We performed functional testing by navigating to the Point Details Table in the UI to verify that values are properly truncated.	Confirms that the change behaves as expected in the actual system.
5	We committed our changes with a clear commit message describing the fix.	Preserves version history and allows rollback if needed.

Time spent (in minutes): 60

Classes and Methods Inspected:

- DataPointDetailsDwr.java
  - getHistoryTableData(int limit)
  - PointValueFacade interactions
  - RenderedPointValueTime interactions

Classes and Methods Changed:

- **DataPointDetailsDwr.java**
  - **Modified:** getHistoryTableData(int limit)
  - **Added:** private String truncateValue(String value)

5. Postfactoring (optional)

Since our changes were minimal and focused on fixing a bug rather than restructuring the code, a major postfactoring step is not strictly necessary.

6. Validation

Step	Description	Rationale
1	**Test case defined:** Tested the regular data flow by calling getHistoryTableData(limit). **Inputs:** A valid integer limit. **Expected Output:** The history table data should be retrieved, and values should be displayed with proper formatting.	This ensures the regular expected behavior is maintained. The test passed.
2	**Test case defined:** Tested handling of large numerical values. **Inputs:** A large number in the history table data (e.g., values with more than 10 decimal places). **Expected Output:** Values should be truncated to two decimal places.	This verifies that our truncateValue function works correctly for large numbers. The test passed.
3	**Test case defined:** Tested non-numeric values in the history table. **Inputs:** String-based data instead of numbers. **Expected Output:** The values should remain unchanged, ensuring that non-numeric values are not altered.	This ensures that truncateValue does not modify non-numeric entries. The test passed.
4	**Test case defined:** Checked API response structure in the network tab after the fix. **Inputs:** Various limit values and edge cases. **Expected Output:** The API should return correctly formatted history data without errors.	This confirms the correctness of the API response and verifies that our changes do not break existing functionality. The test passed.

Time spent (in minutes): 60

7. Summary of the change request

Phase	Time (minutes)	No. of classes inspected	No. of classes changed	No. of methods inspected	No. of methods changes
Concept location	65	2	0	1	0
Impact Analysis	30	2	0	5	0
Prefactoring	15	1	1	1	1
Actualization	60	1	1	2	2
Postfactoring	-	-	-	-	-
Verification	60	1	0	4	0
Total	230	2	1	5	2

8. Conclusion

Concept location was straightforward using developer tools and keyword searches. Impact analysis required inspecting interconnected methods to prevent side effects.

We encapsulated truncation logic for maintainability, but testing was challenging—unit tests validated logic, while UI changes required extensive manual testing. Ensuring non-numeric values remained unaffected needed extra validation. Verifying API responses while maintaining functionality was a major challenge, requiring network inspection and debugging. Adapting to coding standards and collaborating across teams added complexity, but a structured approach ensured efficient implementation.