Requirement Doc - JS to BR

1. Product Development Use-cases:

2. Upgrade Use-cases: &

none

File Conversion - JS to BR

This conversion effort focuses primarily on the file User want to convert e.g. **GRC module JavaScript file**, the script `MS_GRC_ASSET.js`. This main file often imports or includes logic from shared utility scripts such as `MS_GRC_COMMON.js` which in turn imports `MS_GRC_UTIL.js`. The scope includes converting the main file's logic to Business Rules **along with any referenced portions of the utility files**. Key considerations for scope include:

- **Primary Target File:** The attached/copied JS file (e.g., `MS_GRC_ASSET.js`) is fully in scope for conversion to BR format.
- **Imported Utilities:** Shared JS files like `MS_GRC_COMMON.js` and `MS_GRC_UTIL.js` are in scope **only for the parts that are actually used by the main file**. We will identify functions or code sections in these utilities that the main file calls and include those in the conversion.
- **Unused Code Exclusion:** Any functions or code in the common/util files that are **not referenced by the main module file will not be converted** at this time. This ensures we focus on necessary logic related to attached main scripts.

By limiting the scope to used code, we ensure efficiency and reduce the risk of altering behavior that isn't needed in the new environment.

Conversion Rules

All conversions from JS to Business Rules must adhere to the following rules to maintain consistency and traceability:

- **1. Convert Only Referenced Code:** *Only* JavaScript code that is actually referenced or invoked in the will be converted into Business Rule logic. For example, if `MS_GRC_ASSET.js` calls a function `calculateRisk()` from `MS_GRC_COMMON.js`, then the implementation of `calculateRisk()` is in scope for conversion. Unused functions or sections in `MS_GRC_COMMON.js`/`MS_GRC_UTIL.js` will remain unchanged.
- **2. Preserve Original JS as Comments:** Every block of JS code that gets converted into a Business Rule must remain in the source files as a commented-out section. This means after conversion, the original JS logic is still present in the file but enclosed in comments so it no longer executes. This provides a backup reference and aids in verifying that the new BR logic matches the old behavior. It also serves as a fallback during testing and a historical record of what was changed.
    - **Note**: This process of keeping the old/unconverted is followed now to have the code in safe mode for fixing any issues but has to be removed for a cleaner code. This is fine for now but once audit trail is introduced, this should be removed. Developer should be referring to the audit trail for changes made.
- **3.BR Consolidation & Export**

  Once conversion is complete, the new Business Rule code will be presented in a consolidated view, enabling developers to:

1. **Copy & Paste:** Easily copy individual rules and paste them into the Designer, assigning each to the appropriate form component (field, section, grid, header, etc.)
2. **Ensure Functional Parity:** Verify that each BR reproduces the original JS behavior—form validation, data processing, UI manipulation—while conforming to the target platform's syntax (E1 or E2). The tool will flag any syntax or logic changes required for E2/React compatibility.
3. **Maintain Execution Order:** Because E2 no longer supports chained JS files, all rules are centralized in a single sequence to preserve the original logical flow.
4. **Bulk Export Option:** Export all converted Business Rules at once, or pick and choose individual rules as needed.

- **4. Commenting Strategy for Converted Utility Code**:
  - **Comment in Main File Only**
    ** Consolidate any imported JS logic into the main module file (e.g., `MS_GRC_ASSET.js`).
    ** Immediately before the new Business Rule block, comment out the original JS snippet.
    ** Annotate with source context, for example:

    ```
    {code:javascript}
    // [Converted from MS_GRC_COMMON.js: calculateRisk()]
    // Original JS lines 23–45
    // function calculateRisk(...) { … }
    {code}
    ```
  - **Preserve Utility Files Unchanged**
    ** Do not modify or comment out the code in the utility files (`MS_GRC_COMMON.js`, `MS_GRC_UTIL.js`).
    ** This ensures other modules that still depend on those utilities continue to work without disruption.
  - **Context Annotation**
    ** In every commented block, include the main file name (e.g., `MS_GRC_ASSET.js`) and the original utility source (file and function).
- **5. Maintain Readability:** The inserted comments and BR code should be clearly marked and formatted for readability. For example, begin comment blocks with a consistent tag like `// [Converted to BR] Original code: ...` to differentiate them from other comments. This will help developers quickly understand which parts of the code were part of the conversion effort.
- 6. **Usage of BR Syntax grammar:** BR syntax grammar should be used while conversion. This will ensure that there are no syntactical errors.

By following these conversion rules, we maintain a clear link between the old JS implementation and the new Business Rule implementation, which is critical for debugging and future maintenance.

Conversion Workflow

The conversion process will involve an automated or semi-automated workflow to ensure all steps are executed correctly and consistently. Below is the end-to-end workflow from a developer initiating the conversion to obtaining the final converted code:

1. **Initiation – Upload/Paste JS File:** A developer begins by providing the system with the JavaScript file (for example, by uploading `MS_GRC_ASSET.js` or pasting its content into the conversion tool). This file serves as the entry point for the conversion process. Developer will/need to upload all the imported files.
2. **Dependency Resolution – Scan & Import:** The system scans the content of the JS file to identify any references to other scripts or libraries (e.g., calls to functions defined in `MS_GRC_COMMON.js` or `MS_GRC_UTIL.js`). It will automatically

resolve these **nested imports** by loading the referenced files and analyzing their content. This step may involve parsing import statements or known include patterns to gather all dependent code. At the end of this step, the tool has a complete view of the main file plus any relevant sections of utility files that are needed.

3. **Analysis – Generate Conversion Plan:** Using the information gathered, the system generates a **Conversion Plan**. This plan outlines, for each involved file, what changes will be made. Key elements of the plan include:

    - **Referenced Lines/Sections:** A list of specific functions or line ranges in the utility files (`COMMON.js` , `UTIL.js` , etc.) that are invoked by the main file and therefore need conversion.

    - **BR Mappings:** For each identified JS function or logic block, the plan will describe the corresponding Business Rule implementation or reference a predefined BR template/pattern. For example, it might say "Line 50-80 in MS_GRC_COMMON.js (function validateAsset()) will be converted to a BR and placed in MS_GRC_ASSET.js under section X."

    - **Planned Changes:** A summary of modifications, such as "JS to BR conversion details, JSs from the dependent module which are getting converted to BR, etc..." Essentially, the plan acts as a diff-like overview of what the execution will do.

    - **Review Interface:** The plan is typically presented in a human-readable format (potentially a table or list in the tool's UI) so that the developer or analyst can easily review which parts of the code will change.

4. **Review & Approval – Modify or Execute:** The user reviews the Conversion Plan. At this stage, they have the option to **Modify** the plan or **execute** it:

    - If something looks incorrect or needs adjustment (for instance, if an imported file had multiple versions or a function name ambiguity), the user can modify the plan. This could involve deselecting certain conversions, add a missing reference, or edit the mapping (e.g., choosing a different existing Business Rule if one already exists).

    - Once satisfied, the user proceeds to execute the plan. This confirmation triggers the system to carry out the conversion steps on the actual code.

5. **Execution – Automated Conversion:** Upon execution, the system performs the code transformation as outlined:

    - **Show the BR Code:** The main file (`MS_GRC_ASSET.js` in our example) is updated. For each identified JS logic section, a corresponding Business Rule code block is shown. This could be done by adding functions or script segments that implement the same logic under the Business Rule paradigm.

    - **Comment Out JS Blocks:** The original JS code that has now been superseded by the new BR code is commented out. This includes code within the main file (if any original logic there is replaced) as well as the code in the utility files. The comments clearly mark the old code and may include a note about the conversion (for traceability). The structure might look like:

      ```
      javascript
      ```

      `// [Converted to BR] Start of original validateAsset() // ...original JS code... // [Converted to BR] End of original validateAsset()`

      immediately followed by the new Business Rule code implementing `validateAsset()` logic.

    - **Maintain File Integrity:** The system ensures that file formatting and integrity remain intact. Other unrelated code in the files is left unmodified. Only the targeted sections are commented and new sections inserted.

    - The generated BR should be validated Syntactically, compliable. Note: Runtime validation if can be done during this this will help in reducing the effort drastically.

6. **Output Generation – Summary & Download:** After conversion, the system provides the results in multiple forms:

   - **Tabular Summary (JS vs BR):** A table or report is generated that lists each converted section side-by-side with its new implementation. One column will show the original JS snippet (or a reference to its name/line numbers), and the adjacent column will show the Business Rule code or a description of the new logic. This makes it easy for stakeholders to verify that every piece of functionality has a corresponding rule in the new format. It also serves as documentation for future reference.

   - **Inline Commented Code:** The updated source files (the main file and any affected utility files) are available with all changes applied. In these files, one can see the inline comments where the old code resides, and the new BR code inserted. This is useful for developers who want to review or test the changes in context.

   - **Downloadable Files:** The tool will allow the user to download the modified files (e.g., the new `MS_GRC_ASSET.js` with BR code, and updated `MS_GRC_COMMON.js` / `MS_GRC_UTIL.js`). These can be packaged (for example, in a ZIP archive) for easy handoff. The developer can then deploy these files to the GRC application or further integrate them as needed.

   - **Logging and Versioning:** (If applicable) The system may log the changes or version the files, so that there is a record of the conversion. This helps in case the team needs to roll back or compare changes later.

7. **Verification – Testing the Converted Code:** For Phase 1 - developers will copy the BR and place it in instance’s specific FORM’s field/etc. Publish the Form and test.

Context-aware Conversion

1. Context-aware Conversion → Editor-Based Conversion (Human Input → BR)

2. Users can:

   1. **Fetch:** Load the selected form’s metadata from the instance to dynamically list all fields, sections, grids, containers, and headers.

   2. **Choose:** Any form or component from the list to open the Rule Editor, then author or convert logic into a Business Rule.

   3. **Select:** Module → Objects → Forms → [Form Name]
      Define or edit a form-level Business Rule.

   4. **Select:** Module → Objects → Forms → [Form Name] → [Field/Section/Grid/Container/Form Header/etc.]
      Target a specific UI component for component-level Business Rules.

Business Rule Editor Use-cases

- Editor usecase for Developer community- JS - Team Apps - Confluence

- **Auto complete**: Intelligent suggestions for rule names, function calls, field identifiers, and parameters as developer type.

- **Syntax Validation**: Real-time BR grammar checks based on BR Syntax grammar to catch errors before saving or compiling Business Rules.

- **Compilation**: One-click compile to verify that your rules can be parsed and deployed in the target environment.

- **Unit Testing**: scaffold and execute isolated tests against individual rules, with clear pass/fail reports and execution logs.

- **Syntax Highlighting**: Colorize keywords, parameters, comments, and literals to improve readability and reduce typos.

Upgrade Context – E2 Upgrade

This JS-to-BR conversion is a critical part of the broader E2 upgrades. In previous versions of the GRC platform (pre-E2), it was common to use **JavaScript chaining** to implement form logic and validations. This meant splitting logic across multiple files that execute in sequence. For example, the Risk module might have used three separate scripts to manage form behavior:

- `MS_GRX_RISK_PRE.js` – executed before the main form logic (pre-processing or initial validation setup),
- `MS_GRX_RISK.js` – the primary script with core logic, and
- `MS_GRX_RISK_POST.js` – executed after the main logic (post-processing or cleanup).

Under the E2 platform, this approach is no longer supported. The new standard is that all necessary form and business logic for a module must be consolidated, typically into a single script or handled by Business Rules within one context. **JS chaining is deprecated**, meaning the platform will not automatically load or execute multiple JS files in sequence. Instead, the expectation is to have one comprehensive Business Rule script for the module (for example, consolidating all risk logic into `MS_GRX_RISK.js`).

Given this context, our conversion has the following considerations and benefits:

- **Consolidation of Logic:** The conversion merges what used to be spread across `*_PRE.js`, `*.js`, and `*_POST.js` into one place. By converting to Business Rules in the main file, we inherently achieve this consolidation. All the checks and behaviors that were in pre/post scripts will be integrated into the main script as part of the Business Rule code. This aligns the custom module with E2's requirement of having a single source of truth for logic.
- **Compliance with E2 Architecture:** Business Rules are the recommended way to implement server-side logic in the E2 platform. By moving our code to BR format, we ensure compatibility with any underlying changes in how the GRC application loads and executes code. This reduces the risk of functionality breaking after the migration.
- **Preparation for Future Upgrades:** When we upgrade to E2 and beyond, the platform's upgrade engine will compare our customized files with new baseline files (from the vendor). Since we have significantly modified the main script files (by adding BR logic and commenting old code), **future upgrades will require careful, diff-based merging**.

Multi-row Sample

There might be some scenarios where BR creation will be falling under Multirow level and Form Level.

Scenarios like, we need at least one row in the multorow. for the given condition.
this validation or BR can be in both MultiRow and Form Level. these logic has to read from the js and intelligently it has to converted into BR.

{
"constraints": [
{
"errorLevel": "error",
"errorCode": "RULE_ERR_ts57x64bpftv",
"message": "The Incident Classifications tab requires two rows: one for Potential Maximum Consequence, one for Actual Consequence record.",
"expression": "(F.UPLOAD_TYPE.value != '' && F.UPLOAD_TYPE.value != null) ? (F.PMC.countActiveRows()+'' > '0' ? true : false): true",
"label": "size"
}
]
}

--

{
"constraints": [
{
"errorLevel": "error",
"errorCode": "RULE_ERR_s12tsjqcumw3",
"message": "Duplicate Classifications type found, please correct the data.",
"expression": " F.PMC.noDuplicates('CLASSIFICATION_TYPE_ID')",
"label": "orbduplicateValidation"
},
{
"errorLevel": "error",
"errorCode": "RULE_ERR_qe1kmlz8usz7",
"message": "The Incident Classifications tab requires two rows: one for Potential Maximum Consequence, one for Actual Consequence record.",
"expression": "(F.UPLOAD_TYPE.value != '' && F.UPLOAD_TYPE.value != null) ? (F.PMC.countActiveRows()+'' == F.PMC_COUNT.value ? true : false): true",
"label": "pmcVal"
},
{
"errorLevel": "error",
"errorCode": "RULE_ERR_1h7x10kaqfqd5",
"message": "The Potential Maximum Consequence Overall Classification cannot be N/A, kindly update the classification accordingly.",
"expression": "(F.OVERALL_CLASSIFICATION.fetchAllValues({emp -> emp.CLASSIFICATION_TYPE_ID == '2'})[0] != '' && F.OVERALL_CLASSIFICATION.fetchAllValues({emp -> emp.CLASSIFICATION_TYPE_ID == '2'})[0] != '') ? (F.OVERALL_CLASSIFICATION.fetchAllValues({emp -> emp.CLASSIFICATION_TYPE_ID == '2'})[0] != '0' ? true : false): true",
"label": "pmcVal1"
},
{
"errorLevel": "error",
"errorCode": "RULE_ERR_vbeo1uizn8s3",
"message": "The Actual Consequence Overall Classification cannot be greater than Potential Maximum Consequence Overall Classification. This is also applicable to each consequence category individually. Kindly update the classification

```
accordingly.",
"expression": "(F.OVERALL_CLASSIFICATION.fetchAllValues({emp -> emp.CLASSIFICATION_TYPE_ID == '1'})[0] != ''
&& F.OVERALL_CLASSIFICATION.fetchAllValues({emp -> emp.CLASSIFICATION_TYPE_ID == '1'})[0] != '' &&
F.OVERALL_CLASSIFICATION.fetchAllValues({emp -> emp.CLASSIFICATION_TYPE_ID == '2'})[0] != '' &&
F.OVERALL_CLASSIFICATION.fetchAllValues({emp -> emp.CLASSIFICATION_TYPE_ID == '2'})[0] != '') ? (
F.OVERALL_CLASSIFICATION.fetchAllValues({emp -> emp.CLASSIFICATION_TYPE_ID == '1'})[0] <=
F.OVERALL_CLASSIFICATION.fetchAllValues({emp -> emp.CLASSIFICATION_TYPE_ID == '2'})[0] &&
(F.HEALTH_SAFETY.fetchAllValues({emp -> emp.CLASSIFICATION_TYPE_ID == '1'})[0] <=
F.HEALTH_SAFETY.fetchAllValues({emp -> emp.CLASSIFICATION_TYPE_ID == '2'})[0] &&
F.ENVIRONMENT.fetchAllValues({emp -> emp.CLASSIFICATION_TYPE_ID == '1'})[0] <=
F.ENVIRONMENT.fetchAllValues({emp -> emp.CLASSIFICATION_TYPE_ID == '2'})[0] &&
F.SOCIAL_PERFORMANCE.fetchAllValues({emp -> emp.CLASSIFICATION_TYPE_ID == '1'})[0] <=
F.SOCIAL_PERFORMANCE.fetchAllValues({emp -> emp.CLASSIFICATION_TYPE_ID == '2'})[0] &&
F.PMC_SECURITY.fetchAllValues({emp -> emp.CLASSIFICATION_TYPE_ID == '1'})[0] <=
F.PMC_SECURITY.fetchAllValues({emp -> emp.CLASSIFICATION_TYPE_ID == '2'})[0] &&
F.IMAGE_AND_REPUTATION.fetchAllValues({emp -> emp.CLASSIFICATION_TYPE_ID == '1'})[0] <=
F.IMAGE_AND_REPUTATION.fetchAllValues({emp -> emp.CLASSIFICATION_TYPE_ID == '2'})[0]? true : false)):  true",
"label": "pmcVal2"
}
]
}
```

References

1. Reference to Diff Tool for JS to BR: https://metricstream.atlassian.net/wiki/spaces/AR/pages/57790553/Upgrade+Tooling+-+Steps+to+follow+in+the+projects#:~:text=JS%20to%20BR%20Code%20Assistant
2. Impact of Business Rules & Form JS - GS-Europe - Confluence
3. Business Rules - Business Rules - Confluence
4. Business Rules User Guide - Technical Publications - Confluence
5. Platform Developer Guide - Business Rules - Technical Publications - Confluence
6. Business Rules - User Guide - Technical Publications Team - Confluence
7. Business Rules - Guidelines - Technical Publications - Confluence
8. Limitations - Business Rules - Confluence