Function: `readExcelByHeaders`

Description: The `readExcelByHeaders` function reads data from an Excel document in Appian by matching the headers in the Excel sheet with the expected headers provided by the user. The function supports exact matches, synonym matches, and fuzzy matches, making it adaptable to various header formats in the Excel file. Configuration options are provided through the `matchConfig` parameter, allowing for detailed control over the matching behavior. It returns the data as a **List of Dictionary**, which can be further processed within the Appian platform.

Parameters

- **excelDocument** `(Long)`
  - **Required**: Yes
  - **Description**: The document ID of the Excel file stored in Appian's document management system.
- **expectedHeaders** `(String[])`
  - **Required**: Yes
  - **Description**: An array of strings representing the headers expected in the Excel file. These headers will serve as keys for matching against the actual headers in the file.
- **headerSynonyms** `(Dictionary)`
  - **Required**: No
  - **Default**: `null`
  - **Description**: A dictionary that maps each expected header to a list of its synonyms. This helps in matching headers that might appear under different names in the Excel file.
- **matchConfig** `(Dictionary)`
  - **Required**: No
  - **Default**: `{}`
  - **Description**: A dictionary containing configuration options for matching headers and reading the Excel file. The following keys can be included in `matchConfig`:
    - **includeUnmatchedColumns** `(Boolean)`
      - **Required**: No
      - **Default**: `false`
      - **Description**: Determines whether columns that do not match any of the expected headers should be included in the output. Set to `true` to include these columns.

- **stopReadingAtFirstBlankRow** `(Boolean)`
  - **Required**: No
  - **Default**: `false`
  - **Description**: If `true`, the function will stop reading data as soon as it encounters the first completely blank row.
- **sheetNumber** `(Integer)`
  - **Required**: No
  - **Default**: `0`
  - **Description**: Specifies the sheet number to read from in the Excel file. Defaults to the first sheet (`0`).
- **headerPattern** `(String[])`
  - **Required**: No
  - **Default**: `null`
  - **Description**: An array of regular expression patterns used to clean or normalize the headers in the Excel file before matching. For example, `{"[^a-zA-Z0-9]", "\\s+"}` removes all non-alphanumeric characters and extra whitespace.
- **headerRow** `(Integer)`
  - **Required**: No
  - **Default**: `-1` (auto-detect)
  - **Description**: Specifies the row index that contains the headers in the Excel sheet. If not provided or set to `-1`, the function will attempt to automatically identify the header row.
- **fuzzyBoost** `(Boolean)`
  - **Required**: No
  - **Default**: `false`
  - **Description**: Enables fuzzy matching when matching headers. If `true`, the function will use a fuzzy matching algorithm to match headers that are similar but not exact matches.
- **rules** `(List of Dictionary)`
  - **Required**: No
  - **Default**: `null`
  - **Description**: A list of rules to apply when matching headers. Each rule is a dictionary containing:
    - **expectedHeader** `(String)`: The expected header to which the rule applies.
    - **ruleType** `(String)`: The type of rule to apply. Currently supported rule types include:

- **"MERGE_SYNONYMS"**: Merges all columns that match the expected header or any of its synonyms into a single field.

Returns

List of Dictionary: The function returns a TypedValue representing the data extracted from the Excel file, organized into a list of dictionaries. Each dictionary corresponds to a row in the Excel sheet, where the keys are the matched expected headers, and the values are the cell contents.

Example Usage in Appian

```
local!expectedHeaders: {
  "FirstName", "LastName", "Company", "Title", "Email", "State",
"Street"
},

local!synonyms: {
  "FirstName": {"First", "firstname", "first name", "fname"},
  "LastName": {"Last", "Surname", "lastname", "lname"},
  "Company": {"Employer", "company", "organization"},
  "Title": {"Job Title", "Position", "role"},
  "Email": {"Email Address", "email", "e-mail"},
  "State": {"Province", "state", "region"},
  "Street": {"Address", "street", "road"}
},

local!matchConfig: {
  includeUnmatchedColumns: true,
  fuzzyBoost: true,
  headerPattern: {"\[[^\]]+\]", "[^a-z0-9\\s/]", "\\s+"},
  rules: {
    {
      expectedHeader: "State",
      ruleType: "MERGE_SYNONYMS"
```

```
    },
    {
      expectedHeader: "Company",
      ruleType: "MERGE_SYNONYMS"
    },
    {
      expectedHeader: "Street",
      ruleType: "MERGE_SYNONYMS"
    }
  }
},

local!excelData: readexcelbyheaders(
  excelDocument: local!excelDocumentID,
  expectedHeaders: local!expectedHeaders,
  headerSynonyms: local!synonyms,
  matchConfig: local!matchConfig
)
```

Explanation

- **local!expectedHeaders**: Defines a list of the expected headers that should be present in the Excel file. These headers will guide the matching process.
- **local!synonyms**: Provides a mapping of each expected header to a list of potential synonyms. This helps ensure that even if headers in the Excel file are labeled differently, they can still be matched correctly.
- **local!matchConfig**: Contains configuration options for the matching process, including whether to include unmatched columns, whether to use fuzzy matching, patterns for header normalization, and rules for matching.
    - **includeUnmatchedColumns**: Set to `true` to include columns that do not match any of the expected headers in the output.
    - **fuzzyBoost**: Enables fuzzy matching to match headers that are similar but not exact matches.
    - **headerPattern**: An array of regex patterns used to clean and normalize headers before matching.
    - **rules**: Specifies rules to apply when matching headers. In this example, the "MERGE_SYNONYMS" rule is applied to "State", "Company", and "Street", merging all matching synonyms into single fields.

Use Case

The `readExcelByHeaders` function is particularly useful when dealing with Excel files that may have inconsistent header names or variations. By using expected headers, synonyms, and configurable matching rules, the function can reliably extract and organize data, regardless of minor discrepancies in header labeling.

Heuristic for Header Matching Using Synonyms and Rules

The `readExcelByHeaders` function employs a robust heuristic to handle various header names in an Excel sheet. This is especially beneficial when headers in the Excel file might not exactly match the expected headers. The heuristic involves several steps:

1. **Loading Synonyms and Rules**:
    - The function utilizes the `headerSynonyms` parameter, which maps expected headers to lists of potential synonyms.
    - The `rules` specified in `matchConfig` can influence how headers are matched and processed.
2. **Exact Match**:
    - The function first attempts to find an exact match between the cleaned headers in the Excel sheet and the expected headers (including their synonyms).
    - Exact matches are given a high priority and are used directly.
3. **Synonym Match**:
    - If no exact match is found, the function checks the synonyms for each expected header.
    - Synonym matches are prioritized based on their position in the synonym list, with earlier synonyms given higher priority.
4. **Fuzzy Matching**:
    - If neither an exact match nor a synonym match is found, and if `fuzzyBoost` is enabled in `matchConfig`, the function uses a fuzzy matching algorithm to compare the headers.
    - The `FuzzyScore` algorithm assigns a score based on the similarity between the headers in the Excel sheet and the expected headers.
    - Matches with a normalized score above a certain threshold are considered.
5. **Applying Rules**:
    - After potential matches are identified, the function applies any specified rules.
    - For example, the **"MERGE_SYNONYMS"** rule merges all columns that match the expected header or any of its synonyms into a single field in the output.

6. **Conflict Resolution**:
    - The function resolves conflicts where multiple expected headers might match the same actual header.
    - Conflicts are resolved based on match scores and rules, ensuring that each actual header is matched appropriately.
7. **Data Extraction**:
    - Once headers are matched according to the expected headers and rules, the function extracts data from the corresponding columns.
    - The extracted data is returned in a structured format suitable for further processing in Appian.

Example Scenario

Given a synonym map:

```
local!synonyms: {
  "Company": {"Employer", "company", "organization", "Firm"}
}
```

And `matchConfig` rules:

```
local!matchConfig: {
  rules: {
    {
      expectedHeader: "Company",
      ruleType: "MERGE_SYNONYMS"
    }
  }
}
```

If the Excel file contains headers labeled **"Employer"** and **"Firm"**, both synonyms for **"Company"**, the function will merge the data from these columns into a single field **"Company"** in the output.

Conclusion

By configuring `matchConfig` and using header synonyms and rules, the `readExcelByHeaders` function provides flexible and robust data extraction from Excel files, even when the headers are inconsistent or vary between files. This function is essential for scenarios where data standardization is crucial, and header variations are common.

Note: Ensure that the regular expressions in `headerPattern` are correctly escaped in your implementation, as shown in the example usage.

Handling Regex Patterns in Client-Server Communication

When sending regex patterns from the client to the server, especially in Java applications, it's crucial to manage backslashes (`\`) properly to ensure the patterns function as intended.

Defining Regex Patterns on the Client Side

Use Single Backslashes (`\`): Define your regex patterns using single backslashes to prevent over-escaping during serialization and transmission.

```
local!headerPattern: {

  "\[[^\]]+\]",    /* Removes substrings like [form field] */

  "[^a-z0-9\\s/]", /* Removes any character that is not a-z, 0-9,
whitespace, or '/' */

  "\\s+"           /* Replaces multiple whitespace characters with a
single space */

}
```

- **Avoid Double Backslashes (`\\`)**: Do not use double backslashes in your regex patterns, as this can cause over-escaping when the string is serialized and transmitted.

Why Single Backslashes?

- **String Escaping in Appian Expressions:**
  - In Appian expressions, a single backslash is represented as `"\\"` in the code.
  - Using a single backslash in your pattern (e.g., `"\s+"`) ensures that when the string is transmitted to the server, it doesn't get extra backslashes added.

- **Serialization and Transmission:**
    - During serialization, characters like backslashes may be escaped again.
    - If you start with double backslashes, they may turn into quadruple backslashes in the transmitted data, altering the regex pattern.

Server-Side Compilation

- The server receives the transmitted patterns and compiles them into regex patterns without needing to adjust for over-escaping.
- This ensures that the regex operations perform as expected on the server side.

Example Scenario

- **Goal:** Remove all substrings that match `[some text]` from headers.

Client-Side Pattern Definition:

Copy code

```
"\[[^\]]+\]"  /* Correct usage with single backslashes */
```

- **What Happens During Transmission:**
    - The pattern remains as `\[[^\]]+\]` in the serialized data.
    - The backslashes are correctly preserved for regex compilation on the server.
- **Server-Side Regex Compilation:**
    - The server compiles the pattern and correctly identifies and removes substrings enclosed in square brackets.

Common Pitfalls

- **Over-Escaping Backslashes:**
    - Using double backslashes (e.g., `"\\s+"`) can lead to the pattern being received as `"\\\\s+"` on the server.
    - This pattern would incorrectly match a literal backslash followed by `s+`, not the intended whitespace characters.