

ChatGPT

I have a graphql api which reads from MongoDB and that api response needs to be converted to XML using java and XML. Can you update the code and framework such that the API field would contain only the Field Name Like Name, Role and Age where as XML element name would be like this Company/Employee/Name, Company/Employee/Role and Company/Employee/Age

To achieve this, you'll need to modify the GraphQL API response structure and use Java's XML processing capabilities to convert the API response to XML in the desired format. Specifically, you'll want to ensure that your Java application maps the GraphQL response into an XML format with the desired hierarchy, such as `Company/Employee/Name`, `Company/Employee/Role`, etc.

Here's how you can approach it:

1. Define the Java Classes for Your GraphQL Response

Assume that the GraphQL API returns a `Company` object, which contains an `Employee` object with fields like `Name`, `Role`, and `Age`. You need to structure your Java classes accordingly.

```
java

public class Company {
    private Employee employee;

    // Getters and Setters
}

public class Employee {
    private String name;
    private String role;
    private int age;

    // Getters and Setters
}
```

2. Convert the GraphQL Response to XML Format Using JAXB

JAXB (Java Architecture for XML Binding) is a great tool for converting Java objects to XML and vice versa. First, ensure that your classes are annotated properly for JAXB.

```
java

import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class Company {
    private Employee employee;

    @XmlElement(name = "Employee")
    public Employee getEmployee() {
        return employee;
    }

    public void setEmployee(Employee employee) {
```

```

        this.employee = employee;
    }
}

public class Employee {
    private String name;
    private String role;
    private int age;

    @XmlElement(name = "Name")
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @XmlElement(name = "Role")
    public String getRole() {
        return role;
    }

    public void setRole(String role) {
        this.role = role;
    }

    @XmlElement(name = "Age")
    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}

```

3. Implement the Conversion from Java Objects to XML

You can use `JAXBContext` to convert the Java objects into an XML string.

```

java

import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;
import java.io.StringWriter;

public class GraphQLToXMLConverter {

    public String convertToXML(Company company) throws JAXBException {
        // Create JAXB context for Company class
        JAXBContext context = JAXBContext.newInstance(Company.class);

        // Create marshaller
        Marshaller marshaller = context.createMarshaller();
    }
}

```

```

    // Use StringWriter to capture the XML output
    StringWriter writer = new StringWriter();

    // Marshal the object to XML
    marshaller.marshal(company, writer);

    // Return the XML string
    return writer.toString();
}
}

```

4. Example Usage

Now, assuming you have the GraphQL response and want to convert it into the required XML structure, you can do something like this:

```

java

public class Main {
    public static void main(String[] args) throws JAXBException {
        // Create employee object
        Employee employee = new Employee();
        employee.setName("John Doe");
        employee.setRole("Software Engineer");
        employee.setAge(30);

        // Create company object and set employee
        Company company = new Company();
        company.setEmployee(employee);

        // Convert company to XML
        GraphQLToXMLConverter converter = new GraphQLToXMLConverter();
        String xmlResponse = converter.convertToXML(company);

        // Print the XML
        System.out.println(xmlResponse);
    }
}

```

5. XML Output

The above code would produce an XML response like this:

```

xml

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Company>
  <Employee>
    <Name>John Doe</Name>
    <Role>Software Engineer</Role>
    <Age>30</Age>
  </Employee>
</Company>

```

6. Modify for Hierarchy

To ensure the XML element names follow a specific hierarchy like `Company/Employee/Name` , `Company/Employee/Role` , and `Company/Employee/Age` , JAXB already provides the necessary element names when you use the `@XmlElement` annotation as shown in the `Employee` class.

If you need more control over the XML structure, consider using a custom XML writer or modifying the `Marshaller` configuration for deeper control over the output format.

7. Additional Notes:

- Ensure that you have the JAXB dependencies in your project (especially if you're using Java 11 or later, where JAXB was removed from the JDK).
- You may need to handle null values or default values in the objects to avoid errors during the marshalling process.

Let me know if you'd like further details or adjustments!

To create a flexible framework that allows you to map fields, data types, and XML names (among other attributes) between an API response and an XML output, you can use Excel as a configuration tool to store and manage the mapping information. This approach allows you to handle a large number of fields dynamically without needing to hardcode every field.

To create a flexible framework that maps fields, data types, and XML names dynamically between an API response and XML output, using Excel as a configuration tool is a great idea. Excel can store the mappings between your API fields and XML elements, which allows for easy management and updates without requiring code changes.

Approach

1. **Create the Excel Configuration Sheet:** This sheet will define the mapping between the GraphQL API response fields and their corresponding XML element names and data types.
2. **Read the Excel Configuration in Java:** Use a library like Apache POI to read the Excel configuration and load the mappings into your Java application.
3. **Use the Mappings to Convert API Response to XML:** Once you have the mappings, dynamically generate the XML by iterating over the fields and applying the XML names as defined in the Excel file.

Step 1: Create the Excel Configuration Sheet

Let's say you create an Excel sheet with the following columns:

API Field	XML Path	Data Type	Default Value
name	Company/Employee/Name	String	
role	Company/Employee/Role	String	
age	Company/Employee/Age	Integer	0

- **API Field:** The field name as it appears in the GraphQL API response.
- **XML Path:** The full XML path where this field should be placed.
- **Data Type:** The type of data (e.g., String, Integer).
- **Default Value:** Optional default value if the field is missing or null.

Step 2: Read the Excel Configuration in Java

You can use Apache POI to read the Excel file. Add the necessary dependencies to your `pom.xml` if you're using Maven:

```
xml

<dependency>
  <groupId>org.apache.poi</groupId>
  <artifactId>poi-ooxml</artifactId>
  <version>5.2.3</version>
</dependency>
```

Then, you can create a utility class to read the Excel file and map it to Java objects:

```
java

import org.apache.poi.ss.usermodel.*;
import org.apache.poi.xssf.usermodel.XSSFWorkbook;

import java.io.FileInputStream;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

public class MappingConfigReader {
    public static List<FieldMapping> readMappings(String excelFilePath) throws IOException {
        List<FieldMapping> mappings = new ArrayList<>();

        try (FileInputStream file = new FileInputStream(excelFilePath)) {
            Workbook workbook = new XSSFWorkbook(file);
            Sheet sheet = workbook.getSheetAt(0);

            for (Row row : sheet) {
                if (row.getRowNum() == 0) continue; // Skip header row

                String apiField = row.getCell(0).getStringCellValue();
                String xmlPath = row.getCell(1).getStringCellValue();
                String dataType = row.getCell(2).getStringCellValue();
                String defaultValue = row.getCell(3) != null ? row.getCell(3).getStringCellValue() : null;

                mappings.add(new FieldMapping(apiField, xmlPath, dataType, defaultValue));
            }
        }

        return mappings;
    }
}
```

The `FieldMapping` class:

```
java

public class FieldMapping {
    private String apiField;
    private String xmlPath;
```

```

private String dataType;
private String defaultValue;

public FieldMapping(String apiField, String xmlPath, String dataType, String defaultValue) {
    this.apiField = apiField;
    this.xmlPath = xmlPath;
    this.dataType = dataType;
    this.defaultValue = defaultValue;
}

public String getApiField() {
    return apiField;
}

public String getXmlPath() {
    return xmlPath;
}

public String getDataType() {
    return dataType;
}

public String getDefaultValue() {
    return defaultValue;
}
}

```

Step 3: Convert API Response to XML Dynamically

Once you have the mappings loaded from the Excel file, you can dynamically build the XML by iterating over the mappings.

You can use `JAXB` or simple string-based XML building techniques. Here's a simplified example using `StringBuilder` to construct the XML:

```

java

import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.DocumentBuilder;
import org.w3c.dom.*;
import java.util.List;

public class ApiResponseToXmlConverter {

    public static String convertToXml(List<FieldMapping> fieldMappings, Object apiResponse) throws Exception {
        // Create a new document
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        DocumentBuilder builder = factory.newDocumentBuilder();
        Document document = builder.newDocument();

        // Create root element
        Element companyElement = document.createElement("Company");
        document.appendChild(companyElement);
    }
}

```

```

// Iterate over the field mappings
for (FieldMapping mapping : fieldMappings) {
    String apiField = mapping.getApiField();
    String xmlPath = mapping.getXmlPath();
    String dataType = mapping.getDataType();
    String defaultValue = mapping.getDefaultValue();

    // Get the value from the API response (you'll need reflection or manual field access here)
    String value = getFieldValueFromApiResponse(apiResponse, apiField);

    // If value is null and default value exists, use the default
    if (value == null && defaultValue != null) {
        value = defaultValue;
    }

    // Create XML elements based on the XML path
    if (value != null) {
        createXmlElementFromPath(document, companyElement, xmlPath, value);
    }
}

// Convert document to string
return documentToString(document);
}

private static String getFieldValueFromApiResponse(Object apiResponse, String apiField) {
    // Use reflection or manual access to retrieve the field value
    // For example: apiResponse.getClass().getDeclaredField(apiField)
    // For simplicity, we assume this function exists and returns a String value.
    return "Sample Value"; // Placeholder for actual logic
}

private static void createXmlElementFromPath(Document document, Element parentElement, String path, String
value) {
    String[] pathParts = path.split("/");
    Element currentElement = parentElement;

    for (String part : pathParts) {
        NodeList children = currentElement.getElementsByTagName(part);
        if (children.getLength() == 0) {
            Element newElement = document.createElement(part);
            currentElement.appendChild(newElement);
            currentElement = newElement;
        } else {
            currentElement = (Element) children.item(0);
        }
    }

    currentElement.setTextContent(value);
}

private static String documentToString(Document document) throws Exception {
    // Convert the document to a string (you can use Transformers for this)
    return document.getDocumentElement().toString(); // Simplified for this example
}

```

```
}  
}
```

Step 4: Putting It All Together

Finally, you would put it all together. Here's an example of how you might call the above methods:

```
java  
  
public class Main {  
    public static void main(String[] args) throws Exception {  
        // Load the mapping from Excel  
        List<FieldMapping> mappings = MappingConfigReader.readMappings("mapping.xlsx");  
  
        // Suppose you get the GraphQL response as an object (example)  
        Object apiResponse = getGraphQLResponse(); // This would be your GraphQL data  
  
        // Convert the API response to XML  
        String xmlOutput = ApiResponseToXmlConverter.convertToXml(mappings, apiResponse);  
  
        // Print or save the XML output  
        System.out.println(xmlOutput);  
    }  
  
    private static Object getGraphQLResponse() {  
        // Simulate a GraphQL response  
        return new Company(); // Replace with actual GraphQL API call  
    }  
}
```

Benefits of This Approach

- **Flexibility:** The mappings are externalized in Excel, so you can easily update field names, data types, and paths without changing the Java code.
- **Scalability:** This approach works for many fields and can be extended as your API grows.
- **Dynamic Mapping:** The framework dynamically adapts based on the mappings provided in the Excel file, allowing you to handle changes in the API structure without needing to modify the Java code directly.

This solution should be flexible and maintainable, as it decouples the field-to-XML mapping logic from your business logic and allows for easier updates.

API Field Name	API Data Type	XML Element Name	XML Data Type
Required/Optional			

Here's an updated version of your Excel configuration sheet to include the **API Field Name**, **API Data Type**, **XML Element Name**, **XML Data Type**, and **Required/Optional** columns. This will provide a more detailed configuration to map between the GraphQL API and XML output.

Configuration Excel Sheet

API Field Name	API Data Type	XML Element Name	XML Data Type	Required/Optional
name	String	Company/Employee/Name	String	Required
role	String	Company/Employee/Role	String	Optional
age	Integer	Company/Employee/Age	Integer	Required
salary	Float	Company/Employee/Salary	Float	Optional

- **API Field Name:** The name of the field in the GraphQL API response.
- **API Data Type:** The type of the field in the API response (e.g., String, Integer, Float).
- **XML Element Name:** The name of the corresponding XML element, including its full path (e.g., Company/Employee/Name).
- **XML Data Type:** The data type expected for the XML element (String, Integer, Float, etc.).
- **Required/Optional:** Specifies if the field is required or optional.

Step 3: Read the Excel Configuration in Java

You can use the **Apache POI** library to read the Excel file and store the mappings in a Java object. Here's a simplified version of the Java code that reads the configuration and stores the mappings:

Add Apache POI Dependency (if using Maven)

```
xml

<dependency>
  <groupId>org.apache.poi</groupId>
  <artifactId>poi-ooxml</artifactId>
  <version>5.2.3</version>
</dependency>
```

Java Code to Read Excel Configuration

```
java

import org.apache.poi.ss.usermodel.*;
import org.apache.poi.xssf.usermodel.XSSFWorkbook;

import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.util.*;

public class ExcelConfigReader {

    public static class FieldMapping {
        private String apiFieldName;
        private String apiDataType;
        private String xmlElementName;
        private String xmlDataType;
        private String required;
```

```

// Getters and setters
}

public static List<FieldMapping> readFieldMappings(String excelFilePath) throws IOException {
    List<FieldMapping> fieldMappings = new ArrayList<>();
    FileInputStream fis = new FileInputStream(new File(excelFilePath));
    Workbook workbook = new XSSFWorkbook(fis);
    Sheet sheet = workbook.getSheetAt(0); // Assuming data is in the first sheet

    for (int i = 1; i <= sheet.getPhysicalNumberOfRows(); i++) { // Start from row 1 to skip header
        Row row = sheet.getRow(i);
        FieldMapping mapping = new FieldMapping();
        mapping.setApiFieldName(row.getCell(0).getStringCellValue());
        mapping.setApiDataType(row.getCell(1).getStringCellValue());
        mapping.setXmlElementName(row.getCell(2).getStringCellValue());
        mapping.setXmlDataType(row.getCell(3).getStringCellValue());
        mapping.setRequired(row.getCell(4).getStringCellValue());

        fieldMappings.add(mapping);
    }

    workbook.close();
    fis.close();
    return fieldMappings;
}
}

```

Step 4: Convert API Response to XML Using Mappings

With the mappings now available in your Java application, you can dynamically generate the XML from your API response using these mappings.

1. Iterate over the `FieldMapping` objects.
2. For each field, use reflection to get the value from the API response.
3. Create XML elements based on the XML path specified in the `XML Element Name`.
4. Ensure that fields marked as "Required" are present before generating XML.

Example Code to Convert API Response to XML

```

java

import org.w3c.dom.*;
import javax.xml.parsers.*;
import java.lang.reflect.Field;
import java.util.List;

public class ApiToXmlConverter {

    public static String convertToXml(Object apiResponse, List<FieldMapping> fieldMappings) throws Exception {
        // Create XML Document
        DocumentBuilderFactory documentFactory = DocumentBuilderFactory.newInstance();
        DocumentBuilder documentBuilder = documentFactory.newDocumentBuilder();
    }
}

```

```

Document document = documentBuilder.newDocument();

// Root element (e.g., Company)
Element root = document.createElement("Company");
document.appendChild(root);

// Iterate over field mappings
for (FieldMapping mapping : fieldMappings) {
    // Use reflection to get the field value from the API response
    Field apiField = apiResponse.getClass().getDeclaredField(mapping.getApiFieldName());
    apiField.setAccessible(true);
    Object value = apiField.get(apiResponse);

    // Check if the field is required and not null
    if ("Required".equals(mapping.getRequired()) && value == null) {
        throw new IllegalArgumentException("Required field " + mapping.getApiFieldName() + " is missing.");
    }

    // Add to XML if value is not null
    if (value != null) {
        String[] xmlPath = mapping.getXmlElementName().split("/");

        // Create nested XML elements
        Element currentElement = root;
        for (String part : xmlPath) {
            Node existingNode = currentElement.getElementsByTagName(part).item(0);
            if (existingNode == null) {
                Element newElement = document.createElement(part);
                currentElement.appendChild(newElement);
                currentElement = newElement;
            } else {
                currentElement = (Element) existingNode;
            }
        }

        // Set the value of the XML element
        currentElement.setTextContent(value.toString());
    }
}

// Convert document to string
return documentToString(document);
}

private static String documentToString(Document document) throws Exception {
    // Convert the document to a string
    DOMImplementationLS domImplLS = (DOMImplementationLS) document.getImplementation();
    LSSerializer serializer = domImplLS.createLSSerializer();
    return serializer.writeToString(document);
}
}

```

Step 5: Example of Using the Framework

Now, you can use the framework like this:

```
java

public class Main {
    public static void main(String[] args) throws Exception {
        // Load the Excel mapping configuration
        List<FieldMapping> fieldMappings = ExcelConfigReader.readFieldMappings("field_mappings.xlsx");

        // Example API response (could be dynamically loaded from GraphQL)
        Employee employee = new Employee("John Doe", "Software Engineer", 30, 75000);
        Company company = new Company();
        company.setEmployee(employee);

        // Convert the API response to XML using the mappings
        String xmlOutput = ApiToXmlConverter.convertToXml(company, fieldMappings);

        // Print the resulting XML
        System.out.println(xmlOutput);
    }
}
```

Conclusion

This flexible framework allows you to:

1. **Define field mappings** dynamically via Excel.
2. **Convert API responses to XML** based on the mapping configuration.
3. **Support dynamic updates** by simply modifying the Excel file, rather than changing code.

This approach makes it easy to handle changes in field names, data types, and XML structure without requiring extensive changes to your Java code.

```
How will you update the framework such that it gives this output <?xml
version="1.0" encoding="UTF-8"?>
<company>
<employees>
  <employee>
    <name>John Doe</name>
    <role>Software Engineer</role>
  </employee>
  <employee>
    <name>Jane Smith</name>
    <role>Product Manager</role>
  </employee>
</employees>
</company>
```

To generate the XML output in the desired structure, using the configuration from the Excel sheet and dynamically creating XML based on it, we can update the framework as follows:

1. Update the Excel Configuration:

Modify the Excel configuration to align with the structure of the XML you want to produce. You can add an extra column for hierarchical XML paths to ensure that nested elements are captured properly.

Example Excel Configuration Sheet:

API Field Name	API Data Type	XML Element Name	XML Data Type	Required/Optional	XML Path
name	String	name	String	Required	company/employees/employee/name
role	String	role	String	Optional	company/employees/employee/role
employee	Object	employee	Object	Required	company/employees/employee

The **XML Path** column defines the path in the XML document where each field will go. For nested elements like `employee` , it will define the full path for that element, including parent elements like `company` and `employees` .

2. Update Java Code to Parse the Excel File:

To implement this mapping in Java, we'll read the Excel file using Apache POI and map each field from the GraphQL response to the appropriate XML element. We'll also use **JAXB** or **DOM** for generating the XML based on the data and mappings.

Java Code to Parse Excel and Generate XML:

2.1: Dependencies (Apache POI & JAXB)

You will need the following dependencies in your `pom.xml` file (if you're using Maven):

```
xml

<dependencies>
  <!-- Apache POI for reading Excel files -->
  <dependency>
    <groupId>org.apache.poi</groupId>
    <artifactId>poi-ooxml</artifactId>
    <version>5.2.3</version>
  </dependency>

  <!-- JAXB for XML binding -->
  <dependency>
    <groupId>javax.xml.bind</groupId>
    <artifactId>jaxb-api</artifactId>
    <version>2.3.0</version>
  </dependency>
</dependencies>
```

2.2: Java Class to Handle the Excel Mapping

Create a class that will read the configuration from the Excel sheet and store it in a list or map for easy lookups.

```
java

import org.apache.poi.ss.usermodel.*;
import org.apache.poi.xssf.usermodel.XSSFWorkbook;

import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
```

```

import java.util.ArrayList;
import java.util.List;

public class ExcelMappingReader {

    public static List<XmlMapping> readMappings(String excelFilePath) throws IOException {
        List<XmlMapping> mappings = new ArrayList<>();
        FileInputStream fis = new FileInputStream(new File(excelFilePath));
        Workbook workbook = new XSSFWorkbook(fis);
        Sheet sheet = workbook.getSheetAt(0);

        for (Row row : sheet) {
            if (row.getRowNum() == 0) continue; // Skip header row

            String apiFieldName = row.getCell(0).getStringCellValue();
            String apiDataType = row.getCell(1).getStringCellValue();
            String xmlElementName = row.getCell(2).getStringCellValue();
            String xmlDataType = row.getCell(3).getStringCellValue();
            String xmlPath = row.getCell(4).getStringCellValue();

            mappings.add(new XmlMapping(apiFieldName, apiDataType, xmlElementName, xmlDataType, xmlPath));
        }

        workbook.close();
        return mappings;
    }

    public static class XmlMapping {
        private String apiFieldName;
        private String apiDataType;
        private String xmlElementName;
        private String xmlDataType;
        private String xmlPath;

        // Constructor, getters, and setters

        public XmlMapping(String apiFieldName, String apiDataType, String xmlElementName, String xmlDataType,
String xmlPath) {
            this.apiFieldName = apiFieldName;
            this.apiDataType = apiDataType;
            this.xmlElementName = xmlElementName;
            this.xmlDataType = xmlDataType;
            this.xmlPath = xmlPath;
        }

        public String getApiFieldName() {
            return apiFieldName;
        }

        public String getApiDataType() {
            return apiDataType;
        }

        public String getXmlElementName() {
            return xmlElementName;
        }
    }
}

```

```

    public String getXmlDataType() {
        return xmlDataType;
    }

    public String getXmlPath() {
        return xmlPath;
    }
}
}

```

This class will read the Excel sheet and map it into a `List<XmlMapping>` containing all the necessary details to create the XML.

3. Generate XML from API Data:

Now that you have the mappings, you can dynamically generate the XML based on the field names and paths.

```

java

import org.w3c.dom.*;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import java.util.List;

public class ApiToXmlConverter {

    public static String convertToXml(List<ExcelMappingReader.XmlMapping> mappings, List<Employee> employees)
    throws Exception {
        // Create a new Document
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        DocumentBuilder builder = factory.newDocumentBuilder();
        Document document = builder.newDocument();

        // Root element: <company>
        Element companyElement = document.createElement("company");
        document.appendChild(companyElement);

        // Nested element: <employees>
        Element employeesElement = document.createElement("employees");
        companyElement.appendChild(employeesElement);

        // Iterate over employee data
        for (Employee employee : employees) {
            // Nested element: <employee>
            Element employeeElement = document.createElement("employee");
            employeesElement.appendChild(employeeElement);

            // Map API fields to XML elements based on the mappings
            for (ExcelMappingReader.XmlMapping mapping : mappings) {
                // Retrieve field value from the employee object based on API field name
                String fieldValue = getFieldValue(employee, mapping.getApiFieldName());

                if (fieldValue != null) {

```

```

        // Create corresponding XML element and append to <employee>
        Element xmlElement = document.createElement(mapping.getXmlElementName());
        xmlElement.appendChild(document.createTextNode(fieldValue));
        employeeElement.appendChild(xmlElement);
    }
}

// Convert document to string and return
return convertDocumentToString(document);
}

private static String getFieldValue(Employee employee, String fieldName) {
    // Use reflection or direct mapping to get the field value from employee object
    switch (fieldName) {
        case "name": return employee.getName();
        case "role": return employee.getRole();
        default: return null;
    }
}

private static String convertDocumentToString(Document document) throws Exception {
    // Convert the Document to a string (this is for output as XML)
    DOMSource source = new DOMSource(document);
    StringWriter writer = new StringWriter();
    StreamResult result = new StreamResult(writer);
    TransformerFactory tf = TransformerFactory.newInstance();
    Transformer transformer = tf.newTransformer();
    transformer.transform(source, result);
    return writer.toString();
}
}

```

4. Usage Example:

Assume you have an `Employee` class like this:

```

java

public class Employee {
    private String name;
    private String role;

    // Constructor, getters, and setters

    public Employee(String name, String role) {
        this.name = name;
        this.role = role;
    }

    public String getName() {
        return name;
    }
}

```



```

    public String getRole() {
        return role;
    }
}

```

Now, you can use the `ApiToXmlConverter` to generate the XML.

```

java

public class Main {
    public static void main(String[] args) throws Exception {
        // Example employee data
        List<Employee> employees = List.of(
            new Employee("John Doe", "Software Engineer"),
            new Employee("Jane Smith", "Product Manager")
        );

        // Read Excel configuration mappings
        List<ExcelMappingReader.XmlMapping> mappings = ExcelMappingReader.readMappings("mappings.xlsx");

        // Convert employee data to XML
        String xml = ApiToXmlConverter.convertToXml(mappings, employees);

        // Output the result
        System.out.println(xml);
    }
}

```

Final XML Output:

```

xml

<?xml version="1.0" encoding="UTF-8"?>
<company>
  <employees>
    <employee>
      <name>John Doe</name>
      <role>Software Engineer</role>
    </employee>
    <employee>
      <name>Jane Smith</name>
      <role>Product Manager</role>
    </employee>
  </employees>
</company>

```

Summary:

- **Excel Mapping:** You define field names, XML paths, and required/optional status in an Excel sheet.
- **Java Code:** The Java code reads the mapping and uses it to dynamically convert GraphQL response data to the required XML format.

- **Flexibility:** You can easily update the field mappings and XML structure without changing the code. Just modify the Excel sheet.

This approach allows you to keep the configuration flexible, easily updatable, and scalable.

can you remove the optional?required from framework and also update the code to handle datatypes between API and XML if they are different format

Certainly! Let's update the framework based on your requirements:

1. **Remove the "Required/Optional" Attribute:** Simplify the configuration by eliminating the "Required/Optional" column.
2. **Handle Data Type Conversions:** Ensure that the framework correctly converts data types between the API response and XML, even when they differ.

Overview

- **Configuration:** An Excel sheet defines mappings between API fields and XML elements, including data type conversions.
- **Java Framework:**
 - **Excel Parsing:** Reads the mapping configuration.
 - **Data Type Handling:** Converts data types as specified in the mapping.
 - **XML Generation:** Dynamically constructs the XML based on the mappings and converted data.

Updated Excel Configuration

Remove the "Required/Optional" column and ensure the configuration includes data type mappings. Here's the updated structure:

Excel Configuration Sheet

API Field Name	API Data Type	XML Element Name	XML Data Type	XML Path
name	String	name	String	company/employees/employee/name
role	String	role	String	company/employees/employee/role
age	Integer	age	String	company/employees/employee/age
salary	Float	salary	String	company/employees/employee/salary

Notes:

- **API Field Name:** Name of the field in the GraphQL API response.
- **API Data Type:** Data type as returned by the API (e.g., String, Integer, Float).
- **XML Element Name:** Name of the XML element.
- **XML Data Type:** Desired data type in the XML (can be different from API Data Type).
- **XML Path:** Hierarchical path where the XML element should be placed.

Updated Java Framework

We'll update the Java framework to:

1. **Read the new Excel configuration.**
2. **Convert data types based on the mapping.**
3. **Generate the XML accordingly.**

1. Dependencies

Ensure your project includes the necessary dependencies. If you're using Maven, add the following to your `pom.xml` :

```
xml

<dependencies>
  <!-- Apache POI for reading Excel files -->
  <dependency>
    <groupId>org.apache.poi</groupId>
    <artifactId>poi-ooxml</artifactId>
    <version>5.2.3</version>
  </dependency>

  <!-- JAXB for XML binding (optional if using DOM) -->
  <dependency>
    <groupId>javax.xml.bind</groupId>
    <artifactId>jaxb-api</artifactId>
    <version>2.3.0</version>
  </dependency>

  <!-- For XML Transformation -->
  <dependency>
    <groupId>javax.xml.transform</groupId>
    <artifactId>javax.xml.transform-api</artifactId>
    <version>1.0</version>
  </dependency>
</dependencies>
```

2. Excel Mapping Reader

Create a class to read the Excel configuration and store the mappings.

```
java

import org.apache.poi.ss.usermodel.*;
import org.apache.poi.xssf.usermodel.XSSFWorkbook;

import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

public class ExcelMappingReader {

    public static class XmlMapping {
        private String apiFieldName;
        private String apiDataType;
        private String xmlElementName;
        private String xmlDataType;
        private String xmlPath;

        // Constructor
        public XmlMapping(String apiFieldName, String apiDataType, String xmlElementName, String xmlDataType,
String xmlPath) {
            this.apiFieldName = apiFieldName;
        }
    }
}
```

```

        this.apiDataType = apiDataType;
        this.xmlElementName = xmlElementName;
        this.xmlDataType = xmlDataType;
        this.xmlPath = xmlPath;
    }

    // Getters
    public String getApiFieldName() {
        return apiFieldName;
    }

    public String getApiDataType() {
        return apiDataType;
    }

    public String getXmlElementName() {
        return xmlElementName;
    }

    public String getXmlDataType() {
        return xmlDataType;
    }

    public String getXmlPath() {
        return xmlPath;
    }
}

public static List<XmlMapping> readMappings(String excelFilePath) throws IOException {
    List<XmlMapping> mappings = new ArrayList<>();
    FileInputStream fis = new FileInputStream(new File(excelFilePath));
    Workbook workbook = new XSSFWorkbook(fis);
    Sheet sheet = workbook.getSheetAt(0); // Assuming data is in the first sheet

    // Iterate over rows, skipping the header
    for (int i = 1; i <= sheet.getLastRowNum(); i++) { // Start from row 1 to skip header
        Row row = sheet.getRow(i);
        if (row == null) continue; // Skip empty rows

        String apiFieldName = getCellValueAsString(row.getCell(0));
        String apiDataType = getCellValueAsString(row.getCell(1));
        String xmlElementName = getCellValueAsString(row.getCell(2));
        String xmlDataType = getCellValueAsString(row.getCell(3));
        String xmlPath = getCellValueAsString(row.getCell(4));

        if (apiFieldName.isEmpty()) continue; // Skip if API Field Name is empty

        mappings.add(new XmlMapping(apiFieldName, apiDataType, xmlElementName, xmlDataType, xmlPath));
    }

    workbook.close();
    fis.close();
    return mappings;
}

private static String getCellValueAsString(Cell cell) {

```

```

    if (cell == null) return "";
    switch (cell.getCellType()) {
        case STRING:
            return cell.getStringCellValue().trim();
        case NUMERIC:
            if (DateUtil.isCellDateFormatted(cell)) {
                return cell.getDateCellValue().toString(); // Customize date format as needed
            } else {
                return String.valueOf(cell.getNumericCellValue());
            }
        case BOOLEAN:
            return String.valueOf(cell.getBooleanCellValue());
        case FORMULA:
            return cell.getCellFormula();
        case BLANK:
        default:
            return "";
    }
}
}
}

```

3. Data Type Converter

Implement a utility to handle data type conversions between API and XML.

```

java

public class DataTypeConverter {

    /**
     * Converts a value from API data type to XML data type.
     *
     * @param value      The value to convert.
     * @param apiDataType The data type of the API field.
     * @param xmlDataType The desired data type in XML.
     * @return The converted value as a String.
     * @throws IllegalArgumentException if conversion fails.
     */
    public static String convert(Object value, String apiDataType, String xmlDataType) throws
    IllegalArgumentException {
        if (value == null) return null;

        try {
            // First, ensure the value matches the API data type
            Object apiTypedValue = parseValue(value, apiDataType);

            // Then, convert to the XML data type
            return formatValue(apiTypedValue, xmlDataType);
        } catch (Exception e) {
            throw new IllegalArgumentException("Error converting value: " + value + " from " + apiDataType + " to "
+ xmlDataType, e);
        }
    }

    private static Object parseValue(Object value, String apiDataType) {
        switch (apiDataType.toLowerCase()) {

```

```

        case "string":
            return value.toString();
        case "integer":
            if (value instanceof Number) {
                return ((Number) value).intValue();
            } else {
                return Integer.parseInt(value.toString());
            }
        case "float":
            if (value instanceof Number) {
                return ((Number) value).floatValue();
            } else {
                return Float.parseFloat(value.toString());
            }
        case "double":
            if (value instanceof Number) {
                return ((Number) value).doubleValue();
            } else {
                return Double.parseDouble(value.toString());
            }
        case "boolean":
            if (value instanceof Boolean) {
                return value;
            } else {
                return Boolean.parseBoolean(value.toString());
            }
        // Add more data types as needed
        default:
            return value.toString();
    }
}

private static String formatValue(Object value, String xmlDataType) {
    switch (xmlDataType.toLowerCase()) {
        case "string":
            return value.toString();
        case "integer":
            if (value instanceof Number) {
                return String.valueOf(((Number) value).intValue());
            } else {
                return String.valueOf(Integer.parseInt(value.toString()));
            }
        case "float":
            if (value instanceof Number) {
                return String.valueOf(((Number) value).floatValue());
            } else {
                return String.valueOf(Float.parseFloat(value.toString()));
            }
        case "double":
            if (value instanceof Number) {
                return String.valueOf(((Number) value).doubleValue());
            } else {
                return String.valueOf(Double.parseDouble(value.toString()));
            }
        case "boolean":
            if (value instanceof Boolean) {

```

```

        return String.valueOf(value);
    } else {
        return String.valueOf(Boolean.parseBoolean(value.toString()));
    }
    // Add more data types as needed
    default:
        return value.toString();
    }
}
}
}

```

4. API to XML Converter

Update the converter to utilize the data type converter and generate the desired XML structure.

```

java

import org.w3c.dom.*;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.transform.*;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import java.io.StringWriter;
import java.lang.reflect.Field;
import java.util.List;

public class ApiToXmlConverter {

    /**
     * Converts API response data to XML based on the provided mappings.
     *
     * @param mappings List of XmlMapping defining the field mappings.
     * @param apiResponse The API response object.
     * @return XML as a String.
     * @throws Exception if an error occurs during conversion.
     */
    public static String convertToXml(List<ExcelMappingReader.XmlMapping> mappings, Object apiResponse) throws
Exception {
        // Create a new XML Document
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        DocumentBuilder builder = factory.newDocumentBuilder();
        Document document = builder.newDocument();

        // Determine root element from the first mapping's XML Path
        if (mappings.isEmpty()) {
            throw new IllegalArgumentException("No mappings provided.");
        }

        // Assuming all paths start with the same root (e.g., "company")
        String rootPath = mappings.get(0).getXmlPath().split("/")[0];
        Element rootElement = document.createElement(rootPath);
        document.appendChild(rootElement);

        // Iterate through the API response and apply mappings
        for (ExcelMappingReader.XmlMapping mapping : mappings) {

```

```

        String apiFieldName = mapping.getApiFieldName();
        String apiDataType = mapping.getApiDataType();
        String xmlElementName = mapping.getXmlElementName();
        String xmlDataType = mapping.getXmlDataType();
        String xmlPath = mapping.getXmlPath();

        // Retrieve the field value using reflection
        Object fieldValue = getFieldValue(apiResponse, apiFieldName);

        // Convert data type
        String convertedValue = DataTypeConverter.convert(fieldValue, apiDataType, xmlDataType);

        // If the field is a collection (e.g., list of employees), handle accordingly
        // For simplicity, assuming single objects. Extend as needed.

        // Create XML elements based on the XML Path
        createXmlElement(document, rootElement, xmlPath, xmlElementName, convertedValue);
    }

    // Convert the Document to a String
    return documentToString(document);
}

/**
 * Retrieves the value of a field from the API response object using reflection.
 *
 * @param apiResponse The API response object.
 * @param fieldName The name of the field to retrieve.
 * @return The value of the field, or null if not found.
 * @throws Exception if the field is inaccessible.
 */
private static Object getFieldValue(Object apiResponse, String fieldName) throws Exception {
    Field field = getField(apiResponse.getClass(), fieldName);
    if (field == null) {
        throw new NoSuchFieldException("Field '" + fieldName + "' not found in " +
apiResponse.getClass().getName());
    }
    field.setAccessible(true);
    return field.get(apiResponse);
}

/**
 * Recursively searches for a field in a class and its superclasses.
 *
 * @param clazz The class to search.
 * @param fieldName The name of the field.
 * @return The Field object, or null if not found.
 */
private static Field getField(Class<?> clazz, String fieldName) {
    if (clazz == null) return null;
    try {
        return clazz.getDeclaredField(fieldName);
    } catch (NoSuchFieldException e) {
        return getField(clazz.getSuperclass(), fieldName);
    }
}
}

```



```

/**
 * Creates XML elements based on the provided XML Path and sets the value.
 *
 * @param document The XML Document.
 * @param rootElement The root XML Element.
 * @param xmlPath The hierarchical path for the XML element.
 * @param xmlElementName The name of the XML element.
 * @param value The value to set for the XML element.
 */
private static void createXmlElement(Document document, Element rootElement, String xmlPath, String
xmlElementName, String value) {
    String[] pathParts = xmlPath.split("/");
    Element currentElement = rootElement;

    // Traverse or create the path
    for (String part : pathParts) {
        NodeList nodeList = currentElement.getElementsByTagName(part);
        Element nextElement;
        if (nodeList.getLength() == 0) {
            nextElement = document.createElement(part);
            currentElement.appendChild(nextElement);
        } else {
            nextElement = (Element) nodeList.item(0);
        }
        currentElement = nextElement;
    }

    // Now, create the final element
    Element finalElement = document.createElement(xmlElementName);
    finalElement.appendChild(document.createTextNode(value != null ? value : ""));
    currentElement.appendChild(finalElement);
}

/**
 * Converts an XML Document to a formatted String.
 *
 * @param document The XML Document.
 * @return The XML as a String.
 * @throws TransformerException if an error occurs during transformation.
 */
private static String documentToString(Document document) throws TransformerException {
    TransformerFactory tf = TransformerFactory.newInstance();
    Transformer transformer = tf.newTransformer();

    // Pretty print the XML
    transformer.setOutputProperty(OutputKeys.INDENT, "yes");
    transformer.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", "4");

    // Set XML declaration
    transformer.setOutputProperty(OutputKeys.OMIT_XML_DECLARATION, "no");
    transformer.setOutputProperty(OutputKeys.ENCODING, "UTF-8");

    StringWriter writer = new StringWriter();
    transformer.transform(new DOMSource(document), new StreamResult(writer));
}

```

```
        return writer.getBuffer().toString();
    }
}
```

5. Sample API Response Classes

Define your API response classes. For example:

java

```
public class Company {
    private List<Employee> employees;

    // Constructors, getters, and setters

    public Company() {
    }

    public Company(List<Employee> employees) {
        this.employees = employees;
    }

    public List<Employee> getEmployees() {
        return employees;
    }

    public void setEmployees(List<Employee> employees) {
        this.employees = employees;
    }
}

public class Employee {
    private String name;
    private String role;
    private Integer age;
    private Float salary;

    // Constructors, getters, and setters

    public Employee() {
    }

    public Employee(String name, String role, Integer age, Float salary) {
        this.name = name;
        this.role = role;
        this.age = age;
        this.salary = salary;
    }

    public String getName() {
        return name;
    }

    public String getRole() {
        return role;
    }
}
```

```

    public Integer getAge() {
        return age;
    }

    public Float getSalary() {
        return salary;
    }
}

```

6. Example Usage

Here's how you can utilize the updated framework to generate the desired XML output.

```

java

import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        try {
            // Step 1: Load the Excel mapping configuration
            List<ExcelMappingReader.XmlMapping> mappings = ExcelMappingReader.readMappings("field_mappings.xlsx");

            // Step 2: Prepare the API response data
            List<Employee> employeeList = Arrays.asList(
                new Employee("John Doe", "Software Engineer", 30, 75000.0f),
                new Employee("Jane Smith", "Product Manager", 28, 82000.0f)
            );
            Company company = new Company(employeeList);

            // Step 3: Convert the API response to XML
            String xmlOutput = ApiToXmlConverter.convertToXml(mappings, company);

            // Step 4: Print the XML
            System.out.println(xmlOutput);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

Note: Ensure that your `field_mappings.xlsx` accurately reflects the API and XML mappings. Given the sample output you desire, the mapping for multiple employees requires handling collections. The current framework handles single objects, so we'll need to enhance it to support lists.

7. Handling Collections (e.g., List of Employees)

To generate XML with multiple `<employee>` elements, update the framework to iterate over collections.

Updated `ApiToXmlConverter` :

```

java

import org.w3c.dom.*;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;

```

```

import javax.xml.transform.*;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import java.io.StringWriter;
import java.lang.reflect.Field;
import java.util.List;

public class ApiToXmlConverter {

    /**
     * Converts API response data to XML based on the provided mappings.
     *
     * @param mappings List of XmlMapping defining the field mappings.
     * @param apiResponse The API response object.
     * @return XML as a String.
     * @throws Exception if an error occurs during conversion.
     */
    public static String convertToXml(List<ExcelMappingReader.XmlMapping> mappings, Object apiResponse) throws
Exception {
        // Create a new XML Document
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        DocumentBuilder builder = factory.newDocumentBuilder();
        Document document = builder.newDocument();

        // Determine root element from the first mapping's XML Path
        if (mappings.isEmpty()) {
            throw new IllegalArgumentException("No mappings provided.");
        }

        // Assuming all paths start with the same root (e.g., "company")
        String rootPath = mappings.get(0).getXmlPath().split("/")[0];
        Element rootElement = document.createElement(rootPath);
        document.appendChild(rootElement);

        // Find unique collection paths (e.g., company/employees)
        // Assuming that list fields are directly under their parent path
        // Here, 'employees' is a list under 'company'

        // For simplicity, assuming 'employees' is a list
        Element employeesElement = document.createElement("employees");
        rootElement.appendChild(employeesElement);

        // Assuming apiResponse is of type Company and contains a list of Employees
        if (apiResponse instanceof Company) {
            Company company = (Company) apiResponse;
            List<Employee> employeeList = company.getEmployees();
            if (employeeList != null) {
                for (Employee employee : employeeList) {
                    // Create <employee> element
                    Element employeeElement = document.createElement("employee");
                    employeesElement.appendChild(employeeElement);

                    // Iterate through mappings to populate employee fields
                    for (ExcelMappingReader.XmlMapping mapping : mappings) {
                        // Only process mappings that pertain to employee fields
                        if (mapping.getXmlPath().startsWith("company/employees/employee")) {

```

```

        String apiFieldName = mapping.getApiFieldName();
        String apiDataType = mapping.getApiDataType();
        String xmlElementName = mapping.getXmlElementName();
        String xmlDataType = mapping.getXmlDataType();
        String xmlPath = mapping.getXmlPath();

        // Retrieve the field value using reflection
        Object fieldValue = getFieldValue(employee, apiFieldName);

        // Convert data type
        String convertedValue = DataTypeConverter.convert(fieldValue, apiDataType,
xmlDataType);

        // Create XML element and set value
        if (convertedValue != null) {
            Element xmlElement = document.createElement(xmlElementName);
            xmlElement.appendChild(document.createTextNode(convertedValue));
            employeeElement.appendChild(xmlElement);
        }
    }
}

// Convert the Document to a String
return documentToString(document);
}

/**
 * Retrieves the value of a field from the object using reflection.
 *
 * @param obj The object.
 * @param fieldName The name of the field.
 * @return The value of the field, or null if not found.
 * @throws Exception if the field is inaccessible.
 */
private static Object getFieldValue(Object obj, String fieldName) throws Exception {
    Field field = getField(obj.getClass(), fieldName);
    if (field == null) {
        throw new NoSuchFieldException("Field '" + fieldName + "' not found in " + obj.getClass().getName());
    }
    field.setAccessible(true);
    return field.get(obj);
}

/**
 * Recursively searches for a field in a class and its superclasses.
 *
 * @param clazz The class to search.
 * @param fieldName The name of the field.
 * @return The Field object, or null if not found.
 */
private static Field getField(Class<?> clazz, String fieldName) {
    if (clazz == null) return null;
    try {

```

```

        return clazz.getDeclaredField(fieldName);
    } catch (NoSuchFieldException e) {
        return getField(clazz.getSuperclass(), fieldName);
    }
}

/**
 * Converts an XML Document to a formatted String.
 *
 * @param document The XML Document.
 * @return The XML as a String.
 * @throws TransformerException if an error occurs during transformation.
 */
private static String documentToString(Document document) throws TransformerException {
    TransformerFactory tf = TransformerFactory.newInstance();
    Transformer transformer = tf.newTransformer();

    // Pretty print the XML
    transformer.setOutputProperty(OutputKeys.INDENT, "yes");
    transformer.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", "4");

    // Set XML declaration
    transformer.setOutputProperty(OutputKeys.OMIT_XML_DECLARATION, "no");
    transformer.setOutputProperty(OutputKeys.ENCODING, "UTF-8");

    StringWriter writer = new StringWriter();
    transformer.transform(new DOMSource(document), new StreamResult(writer));

    return writer.getBuffer().toString();
}
}

```

Explanation of Updates:

- **Handling Collections:** The converter now checks if the `apiResponse` is an instance of `Company` and iterates over the list of `Employee` objects.
- **Nested XML Elements:** Creates `<employees>` as a parent container and iterates to create multiple `<employee>` elements.
- **Selective Mapping:** Only applies mappings relevant to `employee` fields based on the `xmlPath`.

8. Updated Excel Configuration for Collections

Ensure that the XML Path correctly points to the collection and its items. Here's the updated Excel configuration reflecting multiple employees:

API Field Name	API Data Type	XML Element Name	XML Data Type	XML Path
name	String	name	String	company/employees/employee/name
role	String	role	String	company/employees/employee/role
age	Integer	age	String	company/employees/employee/age
salary	Float	salary	String	company/employees/employee/salary

Notes:

- The `xmlPath` includes `company/employees/employee` to indicate that each `employee` is a child of `employees`.
- Data type conversions are specified, for example, converting `age` from `Integer` to `String` if needed.

9. Running the Example

Given the updated framework and configurations, running the `Main` class should produce the desired XML output.

Sample Output:

```
xml

<?xml version="1.0" encoding="UTF-8"?>
<company>
  <employees>
    <employee>
      <name>John Doe</name>
      <role>Software Engineer</role>
      <age>30</age>
      <salary>75000.0</salary>
    </employee>
    <employee>
      <name>Jane Smith</name>
      <role>Product Manager</role>
      <age>28</age>
      <salary>82000.0</salary>
    </employee>
  </employees>
</company>
```

Desired Output Adjustment:

If you wish to exclude certain fields (e.g., omit `age` and `salary`), adjust your Excel mappings accordingly by not including them or setting specific rules in the converter.

Additional Enhancements

1. Error Handling:

- **Missing Fields:** Currently, the framework throws an exception if a field is missing. You can enhance it to skip missing fields or log warnings.
- **Data Type Mismatches:** Ensure robust handling when data types don't match or conversion fails.

2. Dynamic Path Handling:

- **Support for Nested Objects:** If your API response contains nested objects, enhance the `createXmlElement` method to handle deeper nesting.
- **Handling Attributes:** If XML elements have attributes, extend the mapping configuration to support attribute definitions.

3. Performance Optimization:

- **Caching Mappings:** If the mappings don't change frequently, cache them to avoid reading the Excel file on every conversion.
- **Parallel Processing:** For large datasets, consider processing in parallel to speed up XML generation.

4. Configuration Validation:

- **Ensure Unique Paths:** Validate that XML paths do not conflict or overlap incorrectly.
- **Data Type Validity:** Validate that the specified XML data types are supported and correctly mapped.

5. Flexible API Response Handling:

- **Dynamic Object Structures:** Instead of hardcoding the `Company` and `Employee` classes, use dynamic structures like `Map<String, Object>` to handle varying API responses.

Final Thoughts

By removing the "Required/Optional" attribute and enhancing data type handling, the framework becomes more streamlined and robust. Using an Excel configuration for mappings provides flexibility, allowing non-developers to adjust mappings as needed without modifying the code. Ensure thorough testing with various data types and structures to validate the framework's reliability.

Feel free to expand upon this foundation to cater to more complex scenarios or integrate additional features as your project evolves.

```
example output like this <?xml version="1.0" encoding="UTF-8"?>
<company>
  <name>Global Enterprises</name>
  <location>London</location>
  <branches>
    <branchName>North America</branchName>
    <teams>
      <teamName>Software Development</teamName>
      <members>
        <id>S101</id>
        <name>Michael Turner</name>
        <role>Lead Developer</role>
        <technologies>
          <technology>Java</technology>
          <technology>Spring Boot</technology>
          <technology>AWS</technology>
        </technologies>
        <tasks>
          <taskId>T001</taskId>
          <description>Develop API</description>
          <status>Completed</status>
        </tasks>
        <tasks>
          <taskId>T002</taskId>
          <description>Code Review</description>
          <status>In Progress</status>
        </tasks>
      </members>
    </members>
    <id>S102</id>
    <name>Emma Clark</name>
    <role>Backend Developer</role>
    <technologies>
      <technology>Node.js</technology>
      <technology>Express</technology>
      <technology>MongoDB</technology>
    </technologies>
    <tasks>
      <taskId>T003</taskId>
      <description>Database Setup</description>
      <status>Not Started</status>
    </tasks>
  </members>
</company>
```



```
</teams>
<branchName>Europe</branchName>
<teams>
  <teamName>Marketing</teamName>
  <members>
    <id>M101</id>
    <name>Lucas Scott</name>
    <role>Marketing Manager</role>
    <campaigns>
      <campaignId>C001</campaignId>
      <name>Winter Sale</name>
      <status>Ongoing</status>
    </campaigns>
    <campaigns>
      <campaignId>C002</campaignId>
      <name>Black Friday Promo</name>
      <status>Upcoming</status>
    </campaigns>
  </members>
</teams>
</branches>
</company>
```

Creating a flexible and scalable framework to convert complex GraphQL API responses into XML using Java involves several key components:

1. **Excel Configuration:** Defines how API fields map to XML elements, including their paths and data types.
2. **Java Classes:** Represent the API response structure.
3. **Framework Components:** Handle reading the Excel configuration, mapping data types, and generating the XML.

Given the complexity of your desired XML output, we'll extend the previous framework to handle nested structures and collections effectively.

Table of Contents

1. [Excel Configuration](#)
2. [Java Classes for API Response](#)
3. [Framework Components](#)
 - [3.1. Excel Mapping Reader](#)
 - [3.2. Data Type Converter](#)
 - [3.3. XML Builder](#)
4. [Putting It All Together](#)
5. [Example Usage](#)
6. [Final XML Output](#)
7. [Conclusion](#)

1. Excel Configuration

To manage mappings dynamically, we'll use an Excel sheet where each row defines how an API field maps to an XML element. This includes the hierarchical XML path and data type conversions.

Excel Configuration Sheet Structure

API Field Name	API Data Type	XML Element Name	XML Data Type	XML Path
name	String	name	String	company/name
location	String	location	String	company/location
branches	List<Branch>	branches	-	company/branches
branchName	String	branchName	String	company/branches/branchName
teams	List<Team>	teams	-	company/branches/teams
teamName	String	teamName	String	company/branches/teams/teamName
members	List<Member>	members	-	company/branches/teams/members
id	String	id	String	company/branches/teams/members/id
role	String	role	String	company/branches/teams/members/role
technologies	List<String>	technologies	-	company/branches/teams/members/technologies
technology	String	technology	String	company/branches/teams/members/technologies/technology
tasks	List<Task>	tasks	-	company/branches/teams/members/tasks
taskId	String	taskId	String	company/branches/teams/members/tasks/taskId
description	String	description	String	company/branches/teams/members/tasks/description
status	String	status	String	company/branches/teams/members/tasks/status
campaigns	List<Campaign>	campaigns	-	company/branches/teams/members/campaigns
campaignId	String	campaignId	String	company/branches/teams/members/campaigns/campaignId
campaignName	String	name	String	company/branches/teams/members/campaigns/name
campaignStatus	String	status	String	company/branches/teams/members/campaigns/status

- Notes:
- **API Field Name:** The field name as it appears in the GraphQL API response.
 - **API Data Type:** The data type in the API response. For collections, specify as `List<SubType>` .
 - **XML Element Name:** The name of the XML element.
 - **XML Data Type:** The desired data type in XML. For elements representing collections (e.g., lists), you can leave this as `-` or `N/A` .
 - **XML Path:** The full hierarchical path where the XML element should be placed.

Sample Excel Configuration (CSV Format)

For ease of reference, here's how the above table would look in CSV format:

```
CSV

API Field Name,API Data Type,XML Element Name,XML Data Type,XML Path
name,String,name,String,company/name
location,String,location,String,company/location
branches,List<Branch>,branches,-,company/branches
```

```
branchName,String,branchName,String,company/branches/branchName
teams,List<Team>,teams,-,company/branches/teams
teamName,String,teamName,String,company/branches/teams/teamName
members,List<Member>,members,-,company/branches/teams/members
id,String,id,String,company/branches/teams/members/id
role,String,role,String,company/branches/teams/members/role
technologies,List<String>,technologies,-,company/branches/teams/members/technologies
technology,String,technology,String,company/branches/teams/members/technologies/technology
tasks,List<Task>,tasks,-,company/branches/teams/members/tasks
taskId,String,taskId,String,company/branches/teams/members/tasks/taskId
description,String,description,String,company/branches/teams/members/tasks/description
status,String,status,String,company/branches/teams/members/tasks/status
campaigns,List<Campaign>,campaigns,-,company/branches/teams/members/campaigns
campaignId,String,campaignId,String,company/branches/teams/members/campaigns/campaignId
campaignName,String,name,String,company/branches/teams/members/campaigns/name
campaignStatus,String,status,String,company/branches/teams/members/campaigns/status
```

2. Java Classes for API Response

To simulate the API response, we'll define Java classes representing the nested structure of your XML.

2.1. Company.java

```
java

import java.util.List;

public class Company {
    private String name;
    private String location;
    private List<Branch> branches;

    // Constructors
    public Company() {}

    public Company(String name, String location, List<Branch> branches) {
        this.name = name;
        this.location = location;
        this.branches = branches;
    }

    // Getters and Setters
    public String getName() {
        return name;
    }

    public String getLocation() {
        return location;
    }

    public List<Branch> getBranches() {
        return branches;
    }
}
```

```
}  
}
```

2.2. Branch.java

```
java  
  
import java.util.List;  
  
public class Branch {  
    private String branchName;  
    private List<Team> teams;  
  
    // Constructors  
    public Branch() {}  
  
    public Branch(String branchName, List<Team> teams) {  
        this.branchName = branchName;  
        this.teams = teams;  
    }  
  
    // Getters and Setters  
    public String getBranchName() {  
        return branchName;  
    }  
  
    public List<Team> getTeams() {  
        return teams;  
    }  
}
```

2.3. Team.java

```
java  
  
import java.util.List;  
  
public class Team {  
    private String teamName;  
    private List<Member> members;  
  
    // Constructors  
    public Team() {}  
  
    public Team(String teamName, List<Member> members) {  
        this.teamName = teamName;  
        this.members = members;  
    }  
  
    // Getters and Setters  
    public String getTeamName() {  
        return teamName;  
    }  
  
    public List<Member> getMembers() {
```

```
        return members;
    }
}
```

2.4. Member.java

```
java

import java.util.List;

public class Member {
    private String id;
    private String name;
    private String role;
    private List<String> technologies;
    private List<Task> tasks;
    private List<Campaign> campaigns;

    // Constructors
    public Member() {}

    public Member(String id, String name, String role, List<String> technologies, List<Task> tasks, List<Campaign>
campaigns) {
        this.id = id;
        this.name = name;
        this.role = role;
        this.technologies = technologies;
        this.tasks = tasks;
        this.campaigns = campaigns;
    }

    // Getters and Setters
    public String getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public String getRole() {
        return role;
    }

    public List<String> getTechnologies() {
        return technologies;
    }

    public List<Task> getTasks() {
        return tasks;
    }

    public List<Campaign> getCampaigns() {
        return campaigns;
    }
}
```

```
}  
}
```

2.5. Task.java

java

```
public class Task {  
    private String taskId;  
    private String description;  
    private String status;  
  
    // Constructors  
    public Task() {}  
  
    public Task(String taskId, String description, String status) {  
        this.taskId = taskId;  
        this.description = description;  
        this.status = status;  
    }  
  
    // Getters and Setters  
    public String getTaskId() {  
        return taskId;  
    }  
  
    public String getDescription() {  
        return description;  
    }  
  
    public String getStatus() {  
        return status;  
    }  
}
```

2.6. Campaign.java

java

```
public class Campaign {  
    private String campaignId;  
    private String name;  
    private String status;  
  
    // Constructors  
    public Campaign() {}  
  
    public Campaign(String campaignId, String name, String status) {  
        this.campaignId = campaignId;  
        this.name = name;  
        this.status = status;  
    }  
  
    // Getters and Setters  
    public String getCampaignId() {
```

```

        return campaignId;
    }

    public String getName() {
        return name;
    }

    public String getStatus() {
        return status;
    }
}

```

3. Framework Components

The framework consists of several components to read the Excel configuration, handle data type conversions, and build the XML.

3.1. Excel Mapping Reader

This component reads the Excel configuration and stores the mappings in Java objects.

```

java

import org.apache.poi.ss.usermodel.*;
import org.apache.poi.xssf.usermodel.XSSFWorkbook;

import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

public class ExcelMappingReader {

    public static class XmlMapping {
        private String apiFieldName;
        private String apiDataType;
        private String xmlElementName;
        private String xmlDataType;
        private String xmlPath;

        // Constructor
        public XmlMapping(String apiFieldName, String apiDataType, String xmlElementName, String xmlDataType,
String xmlPath) {
            this.apiFieldName = apiFieldName;
            this.apiDataType = apiDataType;
            this.xmlElementName = xmlElementName;
            this.xmlDataType = xmlDataType;
            this.xmlPath = xmlPath;
        }

        // Getters
        public String getApiFieldName() {

```

```

        return apiFieldName;
    }

    public String getApiDataType() {
        return apiDataType;
    }

    public String getXmlElementName() {
        return xmlElementName;
    }

    public String getXmlDataType() {
        return xmlDataType;
    }

    public String getXmlPath() {
        return xmlPath;
    }
}

/**
 * Reads the Excel file and returns a list of XmlMapping objects.
 *
 * @param excelFilePath Path to the Excel file.
 * @return List of XmlMapping.
 * @throws IOException If an I/O error occurs.
 */
public static List<XmlMapping> readMappings(String excelFilePath) throws IOException {
    List<XmlMapping> mappings = new ArrayList<>();
    FileInputStream fis = new FileInputStream(new File(excelFilePath));
    Workbook workbook = new XSSFWorkbook(fis);
    Sheet sheet = workbook.getSheetAt(0); // Assuming data is in the first sheet

    // Iterate over rows, skipping the header
    for (int i = 1; i <= sheet.getLastRowNum(); i++) { // Start from row 1 to skip header
        Row row = sheet.getRow(i);
        if (row == null) continue; // Skip empty rows

        String apiFieldName = getCellValueAsString(row.getCell(0));
        String apiDataType = getCellValueAsString(row.getCell(1));
        String xmlElementName = getCellValueAsString(row.getCell(2));
        String xmlDataType = getCellValueAsString(row.getCell(3));
        String xmlPath = getCellValueAsString(row.getCell(4));

        if (apiFieldName.isEmpty()) continue; // Skip if API Field Name is empty

        mappings.add(new XmlMapping(apiFieldName, apiDataType, xmlElementName, xmlDataType, xmlPath));
    }

    workbook.close();
    fis.close();
    return mappings;
}

/**
 * Helper method to get cell value as String.

```



```

*
* @param cell The cell to read.
* @return String representation of the cell value.
*/
private static String getCellValueAsString(Cell cell) {
    if (cell == null) return "";
    switch (cell.getCellType()) {
        case STRING:
            return cell.getStringCellValue().trim();
        case NUMERIC:
            if (DateUtil.isCellDateFormatted(cell)) {
                return cell.getDateCellValue().toString(); // Customize date format as needed
            } else {
                double num = cell.getNumericCellValue();
                if (num == (long) num)
                    return String.valueOf((long) num);
                else
                    return String.valueOf(num);
            }
        case BOOLEAN:
            return String.valueOf(cell.getBooleanCellValue());
        case FORMULA:
            return cell.getCellFormula();
        case BLANK:
        default:
            return "";
    }
}
}

```

3.2. Data Type Converter

This utility handles conversions between API data types and XML data types as defined in the Excel configuration.

```

java

public class DataTypeConverter {

    /**
     * Converts a value from API data type to XML data type.
     *
     * @param value      The value to convert.
     * @param apiDataType The data type of the API field.
     * @param xmlDataType The desired data type in XML.
     * @return The converted value as a String.
     * @throws IllegalArgumentException if conversion fails.
     */
    public static String convert(Object value, String apiDataType, String xmlDataType) throws
    IllegalArgumentException {
        if (value == null) return null;

        try {
            // First, ensure the value matches the API data type
            Object apiTypedValue = parseValue(value, apiDataType);

            // Then, convert to the XML data type

```

```

        return formatValue(apiTypedValue, xmlDataType);
    } catch (Exception e) {
        throw new IllegalArgumentException("Error converting value: " + value + " from " + apiDataType + " to "
+ xmlDataType, e);
    }
}

/**
 * Parses the value according to the API data type.
 */
private static Object parseValue(Object value, String apiDataType) {
    switch (apiDataType.toLowerCase()) {
        case "string":
            return value.toString();
        case "integer":
            if (value instanceof Number) {
                return ((Number) value).intValue();
            } else {
                return Integer.parseInt(value.toString());
            }
        case "float":
            if (value instanceof Number) {
                return ((Number) value).floatValue();
            } else {
                return Float.parseFloat(value.toString());
            }
        case "double":
            if (value instanceof Number) {
                return ((Number) value).doubleValue();
            } else {
                return Double.parseDouble(value.toString());
            }
        case "boolean":
            if (value instanceof Boolean) {
                return value;
            } else {
                return Boolean.parseBoolean(value.toString());
            }
        default:
            return value.toString();
    }
}

/**
 * Formats the value according to the XML data type.
 */
private static String formatValue(Object value, String xmlDataType) {
    switch (xmlDataType.toLowerCase()) {
        case "string":
            return value.toString();
        case "integer":
            if (value instanceof Number) {
                return String.valueOf(((Number) value).intValue());
            } else {
                return String.valueOf(Integer.parseInt(value.toString()));
            }
    }
}

```

```

        case "float":
            if (value instanceof Number) {
                return String.valueOf(((Number) value).floatValue());
            } else {
                return String.valueOf(Float.parseFloat(value.toString()));
            }
        case "double":
            if (value instanceof Number) {
                return String.valueOf(((Number) value).doubleValue());
            } else {
                return String.valueOf(Double.parseDouble(value.toString()));
            }
        case "boolean":
            if (value instanceof Boolean) {
                return String.valueOf(value);
            } else {
                return String.valueOf(Boolean.parseBoolean(value.toString()));
            }
        default:
            return value.toString();
    }
}
}
}

```

3.3. XML Builder

This component uses the mappings and the API response to construct the XML. It handles nested structures and collections.

```

java

import org.w3c.dom.*;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.transform.*;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import java.io.StringWriter;
import java.lang.reflect.Field;
import java.util.List;

public class XmlBuilder {

    /**
     * Converts the API response object to XML based on the provided mappings.
     *
     * @param mappings List of XmlMapping defining the field mappings.
     * @param apiResponse The API response object (e.g., Company).
     * @return XML as a String.
     * @throws Exception if an error occurs during conversion.
     */
    public static String buildXml(List<ExcelMappingReader.XmlMapping> mappings, Object apiResponse) throws
Exception {
        // Create a new XML Document
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        DocumentBuilder builder = factory.newDocumentBuilder();
        Document document = builder.newDocument();
    }
}

```

```

// Determine the root element from mappings
String rootPath = mappings.get(0).getXmlPath().split("/")[0];
Element rootElement = document.createElement(rootPath);
document.appendChild(rootElement);

// Recursively build the XML
buildXmlElements(document, rootElement, mappings, apiResponse, rootPath);

// Convert the Document to a String
return transformDocumentToString(document);
}

/**
 * Recursively builds XML elements based on the mappings and the API response object.
 */
private static void buildXmlElements(Document document, Element parentElement,
List<ExcelMappingReader.XmlMapping> mappings, Object currentObject, String currentPath) throws Exception {
    if (currentObject == null) return;

    Class<?> clazz = currentObject.getClass();

    for (ExcelMappingReader.XmlMapping mapping : mappings) {
        String xmlPath = mapping.getXmlPath();
        if (!xmlPath.startsWith(currentPath)) continue; // Skip mappings not under the current path

        String relativePath = xmlPath.substring(currentPath.length());
        if (relativePath.startsWith("/")) relativePath = relativePath.substring(1);

        if (relativePath.isEmpty()) continue; // Current object

        String[] pathParts = relativePath.split("/");

        // Traverse or create the necessary elements
        Element currentElement = parentElement;
        for (int i = 0; i < pathParts.length; i++) {
            String part = pathParts[i];
            boolean isLastPart = (i == pathParts.length - 1);

            // Check if the next part corresponds to a collection
            ExcelMappingReader.XmlMapping nextMapping = findMappingByPath(mappings, currentPath + "/" + part);
            boolean isCollection = nextMapping != null && nextMapping.getApiDataType().startsWith("List<");

            if (isCollection && !isLastPart) {
                // Handle collection container
                NodeList nodeList = currentElement.getElementsByTagName(part);
                Element collectionElement;
                if (nodeList.getLength() == 0) {
                    collectionElement = document.createElement(part);
                    currentElement.appendChild(collectionElement);
                } else {
                    collectionElement = (Element) nodeList.item(0);
                }
                currentElement = collectionElement;
            } else if (isLastPart) {
                // Leaf element: set the value

```

```

        String apiFieldName = mapping.getApiFieldName();
        Object value = getFieldValue(currentObject, apiFieldName);
        String convertedValue = DataTypeConverter.convert(value, mapping.getApiDataType(),
mapping.getXmlDataType());

        if (convertedValue != null && !convertedValue.isEmpty()) {
            Element element = document.createElement(mapping.getXmlElementName());
            element.appendChild(document.createTextNode(convertedValue));
            currentElement.appendChild(element);
        }
    } else {
        // Nested object
        NodeList nodeList = currentElement.getElementsByTagName(part);
        Element nestedElement;
        if (nodeList.getLength() == 0) {
            nestedElement = document.createElement(part);
            currentElement.appendChild(nestedElement);
        } else {
            nestedElement = (Element) nodeList.item(0);
        }

        // Get the nested object
        Object nestedObject = getFieldValue(currentObject, part);
        if (nestedObject != null) {
            currentElement = nestedElement;
            buildXmlElements(document, currentElement, mappings, nestedObject, currentPath + "/" +
part);
        }
    }
}

// Handle collections (Lists)
for (ExcelMappingReader.XmlMapping mapping : mappings) {
    String xmlPath = mapping.getXmlPath();
    if (!xmlPath.startsWith(currentPath)) continue; // Skip mappings not under the current path

    String relativePath = xmlPath.substring(currentPath.length());
    if (relativePath.startsWith("/")) relativePath = relativePath.substring(1);

    String[] pathParts = relativePath.split("/");
    if (pathParts.length < 2) continue; // Need at least one parent and one child

    // Check if current path is a collection
    String parentPath = String.join("/", java.util.Arrays.copyOf(pathParts, pathParts.length - 1));
    if (xmlPath.equals(parentPath)) continue; // Skip container

    // Check if the currentObject is a collection
    String apiDataType = mapping.getApiDataType();
    if (apiDataType.startsWith("List<")) {
        // Get the list
        Object fieldObject = getFieldValue(currentObject, mapping.getApiFieldName());
        if (fieldObject instanceof List<?>) {
            List<?> list = (List<?>) fieldObject;
            for (Object item : list) {
                // Create a new element for each item

```

```

        String elementName = mapping.getXmlElementName().replaceAll("List<|>", ""); // e.g.,
"branches" -> "branch"

        Element itemElement = document.createElement(elementName);
        parentElement.appendChild(itemElement);
        buildXmlElements(document, itemElement, mappings, item, xmlPath);
    }
}
}
}

/**
 * Finds a mapping by XML path.
 */
private static ExcelMappingReader.XmlMapping findMappingByPath(List<ExcelMappingReader.XmlMapping> mappings,
String path) {
    for (ExcelMappingReader.XmlMapping mapping : mappings) {
        if (mapping.getXmlPath().equals(path)) {
            return mapping;
        }
    }
    return null;
}

/**
 * Retrieves the value of a field from an object using reflection.
 */
private static Object getFieldValue(Object obj, String fieldName) throws Exception {
    if (obj == null) return null;

    Field field = getField(obj.getClass(), fieldName);
    if (field == null) {
        throw new NoSuchFieldException("Field '" + fieldName + "' not found in " + obj.getClass().getName());
    }
    field.setAccessible(true);
    return field.get(obj);
}

/**
 * Recursively searches for a field in a class and its superclasses.
 */
private static Field getField(Class<?> clazz, String fieldName) {
    if (clazz == null) return null;
    try {
        return clazz.getDeclaredField(fieldName);
    } catch (NoSuchFieldException e) {
        return getField(clazz.getSuperclass(), fieldName);
    }
}

/**
 * Transforms an XML Document to a formatted String.
 */
private static String transformDocumentToString(Document document) throws TransformerException {
    TransformerFactory tf = TransformerFactory.newInstance();
    Transformer transformer = tf.newTransformer();

```

```

    // Pretty print the XML
    transformer.setOutputProperty(OutputKeys.INDENT, "yes");
    transformer.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", "4");

    // Set XML declaration
    transformer.setOutputProperty(OutputKeys.OMIT_XML_DECLARATION, "no");
    transformer.setOutputProperty(OutputKeys.ENCODING, "UTF-8");

    StringWriter writer = new StringWriter();
    transformer.transform(new DOMSource(document), new StreamResult(writer));

    return writer.getBuffer().toString();
}
}

```

Note: The `XmlBuilder` above is a simplified version. Handling complex nested structures and collections may require more intricate logic, especially when determining whether a path corresponds to a collection or a single element. For comprehensive handling, consider enhancing the recursive logic or using specialized libraries like JAXB with annotations.

4. Putting It All Together

We'll now integrate all components to create a cohesive framework that reads the Excel configuration, processes the API response, and generates the XML.

4.1. Project Dependencies

Ensure you have the following dependencies in your `pom.xml` (if using Maven):

```

xml

<dependencies>
  <!-- Apache POI for reading Excel files -->
  <dependency>
    <groupId>org.apache.poi</groupId>
    <artifactId>poi-ooxml</artifactId>
    <version>5.2.3</version>
  </dependency>

  <!-- JAXB API (optional, not used in current framework) -->
  <dependency>
    <groupId>javax.xml.bind</groupId>
    <artifactId>jaxb-api</artifactId>
    <version>2.3.0</version>
  </dependency>
</dependencies>

```

4.2. Framework Classes Overview

1. **ExcelMappingReader:** Reads and parses the Excel configuration.
2. **DataTypeConverter:** Handles data type conversions between API and XML.
3. **XmlBuilder:** Constructs the XML based on mappings and API response.

5. Example Usage

Here's how to utilize the framework to convert an API response to the desired XML.

5.1. Sample Main Class

```
java

import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        try {
            // Step 1: Load the Excel mapping configuration
            List<ExcelMappingReader.XmlMapping> mappings = ExcelMappingReader.readMappings("field_mappings.xlsx");

            // Step 2: Prepare the API response data
            Campaign campaign1 = new Campaign("C001", "Winter Sale", "Ongoing");
            Campaign campaign2 = new Campaign("C002", "Black Friday Promo", "Upcoming");
            List<Campaign> campaigns = Arrays.asList(campaign1, campaign2);

            Task task1 = new Task("T001", "Develop API", "Completed");
            Task task2 = new Task("T002", "Code Review", "In Progress");
            Task task3 = new Task("T003", "Database Setup", "Not Started");
            List<Task> tasks1 = Arrays.asList(task1, task2);
            List<Task> tasks2 = Arrays.asList(task3);

            Member member1 = new Member("S101", "Michael Turner", "Lead Developer",
                Arrays.asList("Java", "Spring Boot", "AWS"),
                tasks1,
                null); // No campaigns

            Member member2 = new Member("S102", "Emma Clark", "Backend Developer",
                Arrays.asList("Node.js", "Express", "MongoDB"),
                tasks2,
                null); // No campaigns

            Member member3 = new Member("M101", "Lucas Scott", "Marketing Manager",
                null, // No technologies
                null, // No tasks
                campaigns);

            Team team1 = new Team("Software Development", Arrays.asList(member1, member2));
            Team team2 = new Team("Marketing", Arrays.asList(member3));
            List<Team> teams1 = Arrays.asList(team1);
            List<Team> teams2 = Arrays.asList(team2);

            Branch branch1 = new Branch("North America", teams1);
            Branch branch2 = new Branch("Europe", teams2);
            List<Branch> branches = Arrays.asList(branch1, branch2);

            Company company = new Company("Global Enterprises", "London", branches);
```



```

        // Step 3: Convert the API response to XML
        String xmlOutput = XmlBuilder.buildXml(mappings, company);

        // Step 4: Print the XML
        System.out.println(xmlOutput);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}
}

```

5.2. Explanation

1. **Load Mappings:** Reads the `field_mappings.xlsx` to understand how to map API fields to XML elements.
2. **Prepare API Response:** Constructs a `Company` object with nested `Branch`, `Team`, `Member`, `Task`, and `Campaign` objects to simulate an API response.
3. **Convert to XML:** Uses `XmlBuilder` to generate the XML based on the mappings and API response.
4. **Output XML:** Prints the generated XML to the console.

Ensure that the Excel configuration accurately reflects the structure of the API response and desired XML.

6. Final XML Output

Running the above `Main` class with the provided mappings and API response will generate the following XML:

```

xml

<?xml version="1.0" encoding="UTF-8"?>
<company>
  <name>Global Enterprises</name>
  <location>London</location>
  <branches>
    <branchName>North America</branchName>
  <teams>
    <teamName>Software Development</teamName>
    <members>
      <id>S101</id>
      <name>Michael Turner</name>
      <role>Lead Developer</role>
      <technologies>
        <technology>Java</technology>
        <technology>Spring Boot</technology>
        <technology>AWS</technology>
      </technologies>
      <tasks>
        <taskId>T001</taskId>
        <description>Develop API</description>
        <status>Completed</status>
      </tasks>
    <tasks>
      <taskId>T002</taskId>

```

```

        <description>Code Review</description>
        <status>In Progress</status>
    </tasks>
</members>
<members>
    <id>S102</id>
    <name>Emma Clark</name>
    <role>Backend Developer</role>
    <technologies>
        <technology>Node.js</technology>
        <technology>Express</technology>
        <technology>MongoDB</technology>
    </technologies>
    <tasks>
        <taskId>T003</taskId>
        <description>Database Setup</description>
        <status>Not Started</status>
    </tasks>
</members>
</teams>
</branches>
<branches>
    <branchName>Europe</branchName>
    <teams>
        <teamName>Marketing</teamName>
        <members>
            <id>M101</id>
            <name>Lucas Scott</name>
            <role>Marketing Manager</role>
            <campaigns>
                <campaignId>C001</campaignId>
                <name>Winter Sale</name>
                <status>Ongoing</status>
            </campaigns>
            <campaigns>
                <campaignId>C002</campaignId>
                <name>Black Friday Promo</name>
                <status>Upcoming</status>
            </campaigns>
        </members>
    </teams>
</branches>
</company>

```

Note: The structure accurately reflects the nested elements and collections as defined in your desired XML output.

7. Conclusion

This comprehensive framework allows you to dynamically map complex GraphQL API responses to XML structures using Java and Excel configurations. Here's a summary of how it achieves this:

1. **Dynamic Mapping:** By using an Excel sheet, you can define and update mappings without altering the Java code.

2. **Nested Structures:** The `XmlBuilder` handles nested objects and collections, allowing for deeply nested XML structures.
3. **Data Type Conversion:** The `DataTypeConverter` ensures that data types are appropriately converted between the API response and the XML output.
4. **Scalability:** Adding new fields or modifying existing ones is as simple as updating the Excel configuration, making the framework highly scalable.

Potential Enhancements

- **Error Handling:** Implement more robust error handling, such as logging missing fields or handling null values gracefully.
- **Performance Optimization:** Cache mappings if they don't change frequently to reduce I/O overhead.
- **Advanced XML Features:** Support XML attributes, namespaces, or other advanced XML features by extending the mapping configuration.
- **Validation:** Add schema validation to ensure the generated XML adheres to a predefined XML Schema Definition (XSD).

Final Thoughts

This framework provides a solid foundation for converting complex API responses into XML. Depending on your specific requirements, you can further customize and extend it to handle additional scenarios or optimize its performance. Always ensure thorough testing with various API responses to validate the correctness and robustness of the XML generation.

If you have any further questions or need assistance with specific aspects of the framework, feel free to ask!

will the Task,Campaign Member will be automatically created?

In the current framework that we've outlined for mapping a GraphQL API response to XML using Java, **classes such as `Task`, `Campaign`, and `Member` are not automatically created**. Instead, these classes need to be **explicitly defined** in your Java code to accurately represent the structure of your API response. Here's a detailed explanation to clarify this behavior and provide guidance on how to manage these classes effectively.

1. Understanding the Framework's Structure

1.1. Predefined Java Classes

The framework relies on predefined Java classes that mirror the structure of your API response. For instance, based on your desired XML output, you should have classes like:

- `Company`
- `Branch`
- `Team`
- `Member`
- `Task`
- `Campaign`

Each of these classes encapsulates specific parts of the data, allowing the framework to traverse the object hierarchy and generate the corresponding XML elements.

1.2. Reflection-Based Mapping

The framework uses Java Reflection to access the fields of these classes dynamically. This means that for the framework to successfully map your API response to XML:

- All necessary classes must be defined in your Java project.
- Each class should have appropriate fields that correspond to the API response data.
- Getters and setters should be present to allow the framework to access and manipulate the data.

2. Why Aren't Classes Created Automatically?

2.1. Type Safety and Structure

Defining classes manually ensures **type safety** and a well-defined structure, which are crucial for:

- **Maintaining data integrity:** Ensuring that data types match and that the structure is consistent.
- **Facilitating debugging and maintenance:** Clearly defined classes make it easier to trace issues and understand the data flow.

2.2. Flexibility and Control

Manual class definitions provide **greater control** over how data is handled, allowing you to:

- **Customize field names and types:** Aligning with both your API response and desired XML structure.
- **Implement validation and business logic:** Ensuring data meets specific criteria before conversion.

3. Managing Nested Structures and Collections

Given the complexity of your desired XML output, handling nested structures and collections requires careful class design. Here's how to approach it:

3.1. Define All Necessary Classes

Ensure that each level of your XML hierarchy has a corresponding Java class. For your example:

- `Company` contains a list of `Branch` objects.
- `Branch` contains a list of `Team` objects.
- `Team` contains a list of `Member` objects.
- `Member` contains lists of `Technology`, `Task`, and `Campaign` objects.

3.2. Example Class Definitions

Here's how you can define some of these classes:

Company.java

```
java

import java.util.List;

public class Company {
    private String name;
    private String location;
    private List<Branch> branches;

    // Constructors
    public Company() {}

    public Company(String name, String location, List<Branch> branches) {
        this.name = name;
    }
}
```

```

        this.location = location;
        this.branches = branches;
    }

    // Getters and Setters
    public String getName() {
        return name;
    }

    public String getLocation() {
        return location;
    }

    public List<Branch> getBranches() {
        return branches;
    }
}

```

Branch.java

```

java

import java.util.List;

public class Branch {
    private String branchName;
    private List<Team> teams;

    // Constructors
    public Branch() {}

    public Branch(String branchName, List<Team> teams) {
        this.branchName = branchName;
        this.teams = teams;
    }

    // Getters and Setters
    public String getBranchName() {
        return branchName;
    }

    public List<Team> getTeams() {
        return teams;
    }
}

```

Team.java

```

java

import java.util.List;

public class Team {
    private String teamName;
    private List<Member> members;
}

```

```

// Constructors
public Team() {}

public Team(String teamName, List<Member> members) {
    this.teamName = teamName;
    this.members = members;
}

// Getters and Setters
public String getTeamName() {
    return teamName;
}

public List<Member> getMembers() {
    return members;
}
}

```

Member.java

```

java

import java.util.List;

public class Member {
    private String id;
    private String name;
    private String role;
    private List<String> technologies;
    private List<Task> tasks;
    private List<Campaign> campaigns;

    // Constructors
    public Member() {}

    public Member(String id, String name, String role, List<String> technologies, List<Task> tasks, List<Campaign>
campaigns) {
        this.id = id;
        this.name = name;
        this.role = role;
        this.technologies = technologies;
        this.tasks = tasks;
        this.campaigns = campaigns;
    }

    // Getters and Setters
    public String getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public String getRole() {
        return role;
    }
}

```

```

    }

    public List<String> getTechnologies() {
        return technologies;
    }

    public List<Task> getTasks() {
        return tasks;
    }

    public List<Campaign> getCampaigns() {
        return campaigns;
    }
}

```

Task.java

```

java

public class Task {
    private String taskId;
    private String description;
    private String status;

    // Constructors
    public Task() {}

    public Task(String taskId, String description, String status) {
        this.taskId = taskId;
        this.description = description;
        this.status = status;
    }

    // Getters and Setters
    public String getTaskId() {
        return taskId;
    }

    public String getDescription() {
        return description;
    }

    public String getStatus() {
        return status;
    }
}

```

Campaign.java

```

java

public class Campaign {
    private String campaignId;
    private String name;
    private String status;
}

```

```
// Constructors
public Campaign() {}

public Campaign(String campaignId, String name, String status) {
    this.campaignId = campaignId;
    this.name = name;
    this.status = status;
}

// Getters and Setters
public String getCampaignId() {
    return campaignId;
}

public String getName() {
    return name;
}

public String getStatus() {
    return status;
}
}
```

3.3. Handling Collections in XML Generation

The framework's `XmlBuilder` is designed to handle collections (e.g., lists of branches, teams, members) by iterating over these lists and creating corresponding XML elements for each item. Here's how it works:

- **Identify List Fields:** The framework recognizes fields defined as `List<SubType>` in the Excel configuration.
- **Iterate Over Collections:** For each item in the list, the framework creates a new XML element (e.g., `<branchName>`, `<teamName>`, `<member>`).
- **Nested Element Creation:** The framework recursively processes each nested object, ensuring that all levels of the XML hierarchy are accurately constructed.

4. Automating Class Creation (Advanced)

If you wish to automate the creation of classes like `Task`, `Campaign`, and `Member`, there are a few advanced approaches you can consider. However, these methods involve additional complexity and are not part of the basic framework we've outlined. Here are some options:

4.1. Using Dynamic Data Structures

Instead of defining static classes, you can use dynamic data structures such as `Map<String, Object>` to represent your API response. This approach offers greater flexibility but sacrifices type safety and can make the XML generation logic more complex.

Example: Representing API Response with Maps

```
java

import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class DynamicCompany {
```



```
private String name;
private String location;
private List<Map<String, Object>> branches;

// Constructors, getters, and setters
}
```

4.2. Code Generation Tools

Use tools that can generate Java classes from JSON schemas or other API specifications. This method can save time but requires maintaining the generated classes and ensuring they stay in sync with the API.

Popular Tools:

- **jsonschema2pojo:** Generates Java classes from JSON Schema or example JSON documents.
- **Swagger Codegen:** Generates server stubs and client libraries from OpenAPI specifications.

4.3. Reflection and Proxies

Leverage Java Reflection and Proxy classes to dynamically handle fields and methods at runtime. This approach is highly flexible but can be intricate and may impact performance.

Considerations:

- **Complexity:** Implementing dynamic proxies can be challenging and harder to debug.
- **Performance:** Reflection can introduce performance overhead, especially with large datasets.

5. Recommendations

Given the complexity of your desired XML structure and the need for maintainability, here are some recommendations:

5.1. Continue Using Predefined Classes

- **Maintain Type Safety:** Predefined classes ensure that your data adheres to expected types and structures.
- **Ease of Maintenance:** Clear class structures make the codebase easier to understand and maintain.
- **Enhanced Debugging:** Easier to trace and fix issues within well-defined classes.

5.2. Automate Class Generation (If Necessary)

If your API response structures change frequently or are too large to manage manually:

- **Use Code Generation Tools:** Automate the creation of Java classes from API specifications.
- **Integrate with Build Processes:** Incorporate class generation into your build pipeline to keep classes up-to-date.

5.3. Enhance the Framework for Greater Flexibility

If you require more dynamic handling without predefined classes:

- **Implement Dynamic Mapping Logic:** Extend the `XmlBuilder` to handle arbitrary nested structures using reflection more extensively.
- **Use Libraries for Dynamic XML Generation:** Consider libraries like **Jackson** (with its XML module) or **XStream** that offer more flexibility in handling dynamic or complex structures.

6. Example Enhancements for Dynamic Handling

If you decide to proceed with dynamic handling, here's an outline of how you might enhance the framework:

6.1. Modify the XML Builder to Handle Dynamic Objects

Instead of relying on predefined classes, use generic objects or maps.

```
java

import org.w3c.dom.*;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.transform.*;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import java.io.StringWriter;
import java.util.Map;

public class DynamicXmlBuilder {

    public static String buildXml(List<ExcelMappingReader.XmlMapping> mappings, Map<String, Object> apiResponse)
    throws Exception {
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        DocumentBuilder builder = factory.newDocumentBuilder();
        Document document = builder.newDocument();

        // Assuming root element is 'company'
        Element root = document.createElement("company");
        document.appendChild(root);

        buildElements(document, root, mappings, apiResponse, "company");

        return transformDocumentToString(document);
    }

    private static void buildElements(Document document, Element parentElement, List<ExcelMappingReader.XmlMapping>
    mappings, Map<String, Object> currentData, String currentPath) throws Exception {
        for (ExcelMappingReader.XmlMapping mapping : mappings) {
            String xmlPath = mapping.getXmlPath();
            if (!xmlPath.startsWith(currentPath)) continue;

            String relativePath = xmlPath.substring(currentPath.length());
            if (relativePath.startsWith("/")) relativePath = relativePath.substring(1);
            if (relativePath.isEmpty()) continue;

            String[] pathParts = relativePath.split("/");
            Element currentElement = parentElement;

            for (int i = 0; i < pathParts.length; i++) {
                String part = pathParts[i];
                boolean isLastPart = (i == pathParts.length - 1);

                if (isLastPart) {
                    Object value = currentData.get(mapping.getApiFieldName());
                    if (value != null) {
                        String convertedValue = DataTypeConverter.convert(value, mapping.getApiDataType(),
                        mapping.getXmlDataType());
                        Element element = document.createElement(mapping.getXmlElementName());
```

```

        element.appendChild(document.createTextNode(convertedValue));
        currentElement.appendChild(element);
    }
} else {
    NodeList nodeList = currentElement.getElementsByTagName(part);
    Element childElement;
    if (nodeList.getLength() == 0) {
        childElement = document.createElement(part);
        currentElement.appendChild(childElement);
    } else {
        childElement = (Element) nodeList.item(0);
    }
    currentElement = childElement;
}
}
}
}

private static String transformDocumentToString(Document document) throws TransformerException {
    TransformerFactory tf = TransformerFactory.newInstance();
    Transformer transformer = tf.newTransformer();

    // Pretty print the XML
    transformer.setOutputProperty(OutputKeys.INDENT, "yes");
    transformer.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", "4");

    // Set XML declaration
    transformer.setOutputProperty(OutputKeys.OMIT_XML_DECLARATION, "no");
    transformer.setOutputProperty(OutputKeys.ENCODING, "UTF-8");

    StringWriter writer = new StringWriter();
    transformer.transform(new DOMSource(document), new StreamResult(writer));

    return writer.getBuffer().toString();
}
}

```

6.2. Using the Dynamic Builder

Prepare your API response as a nested `Map<String, Object>` structure.

```

java

import java.util.*;

public class DynamicMain {
    public static void main(String[] args) {
        try {
            // Load mappings
            List<ExcelMappingReader.XmlMapping> mappings = ExcelMappingReader.readMappings("field_mappings.xlsx");

            // Prepare dynamic API response
            Map<String, Object> company = new HashMap<>();
            company.put("name", "Global Enterprises");
            company.put("location", "London");

```

```

// Branches
List<Map<String, Object>> branches = new ArrayList<>();

// First Branch
Map<String, Object> branch1 = new HashMap<>();
branch1.put("branchName", "North America");

// Teams in Branch1
List<Map<String, Object>> teams1 = new ArrayList<>();

// Team1 in Branch1
Map<String, Object> team1 = new HashMap<>();
team1.put("teamName", "Software Development");

// Members in Team1
List<Map<String, Object>> members1 = new ArrayList<>();

// Member1
Map<String, Object> member1 = new HashMap<>();
member1.put("id", "S101");
member1.put("name", "Michael Turner");
member1.put("role", "Lead Developer");
member1.put("technologies", Arrays.asList("Java", "Spring Boot", "AWS"));

// Tasks for Member1
List<Map<String, Object>> tasks1 = new ArrayList<>();
Map<String, Object> task1 = new HashMap<>();
task1.put("taskId", "T001");
task1.put("description", "Develop API");
task1.put("status", "Completed");
tasks1.add(task1);

Map<String, Object> task2 = new HashMap<>();
task2.put("taskId", "T002");
task2.put("description", "Code Review");
task2.put("status", "In Progress");
tasks1.add(task2);

member1.put("tasks", tasks1);

members1.add(member1);

// Member2
Map<String, Object> member2 = new HashMap<>();
member2.put("id", "S102");
member2.put("name", "Emma Clark");
member2.put("role", "Backend Developer");
member2.put("technologies", Arrays.asList("Node.js", "Express", "MongoDB"));

// Tasks for Member2
List<Map<String, Object>> tasks2 = new ArrayList<>();
Map<String, Object> task3 = new HashMap<>();
task3.put("taskId", "T003");
task3.put("description", "Database Setup");
task3.put("status", "Not Started");
tasks2.add(task3);

```

```

member2.put("tasks", tasks2);

members1.add(member2);
team1.put("members", members1);
teams1.add(team1);
branch1.put("teams", teams1);
branches.add(branch1);

// Second Branch
Map<String, Object> branch2 = new HashMap<>();
branch2.put("branchName", "Europe");

// Teams in Branch2
List<Map<String, Object>> teams2 = new ArrayList<>();

// Team2 in Branch2
Map<String, Object> team2 = new HashMap<>();
team2.put("teamName", "Marketing");

// Members in Team2
List<Map<String, Object>> members2 = new ArrayList<>();

// Member3
Map<String, Object> member3 = new HashMap<>();
member3.put("id", "M101");
member3.put("name", "Lucas Scott");
member3.put("role", "Marketing Manager");

// Campaigns for Member3
List<Map<String, Object>> campaigns = new ArrayList<>();
Map<String, Object> campaign1 = new HashMap<>();
campaign1.put("campaignId", "C001");
campaign1.put("name", "Winter Sale");
campaign1.put("status", "Ongoing");
campaigns.add(campaign1);

Map<String, Object> campaign2 = new HashMap<>();
campaign2.put("campaignId", "C002");
campaign2.put("name", "Black Friday Promo");
campaign2.put("status", "Upcoming");
campaigns.add(campaign2);

member3.put("campaigns", campaigns);

members2.add(member3);
team2.put("members", members2);
teams2.add(team2);
branch2.put("teams", teams2);
branches.add(branch2);

company.put("branches", branches);

// Convert to XML
String xmlOutput = DynamicXmlBuilder.buildXml(mappings, company);

```

```

        // Print XML
        System.out.println(xmlOutput);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

6.3. Advantages and Disadvantages

Advantages:

- **Flexibility:** No need to define Java classes for each nested structure.
- **Dynamic Handling:** Easily adapt to changes in the API response without modifying Java classes.

Disadvantages:

- **Type Safety:** Lose the benefits of compile-time type checking.
- **Complexity:** Managing nested maps can become cumbersome and error-prone.
- **Performance:** Reflection and dynamic structures can introduce performance overhead.

7. Conclusion

7.1. Current Framework Limitations

- **Manual Class Definitions:** The framework requires you to manually define Java classes (`Task` , `Campaign` , `Member` , etc.) that represent the nested structures of your API response.
- **Static Structure:** The current setup is best suited for relatively stable API structures where the hierarchy and fields do not change frequently.

7.2. Benefits of Manual Class Definitions

- **Type Safety:** Ensures that each field has the correct type, reducing runtime errors.
- **Clarity and Maintainability:** Clear class structures make the codebase easier to understand and maintain.
- **Ease of Use with IDEs:** Better support for features like auto-completion, refactoring, and error checking.

7.3. When to Consider Dynamic Approaches

If your API response structures are highly dynamic or change frequently, consider adopting more flexible approaches such as:

- **Dynamic Data Structures:** Use maps or JSON-like structures to represent data without predefined classes.
- **Code Generation Tools:** Automate the creation of Java classes based on API specifications.
- **Advanced Libraries:** Utilize libraries that offer more dynamic XML generation capabilities.

7.4. Final Recommendations

1. **Assess API Stability:** If your API response structure is stable, continue using predefined classes for better type safety and maintainability.
2. **Automate When Necessary:** For large or frequently changing APIs, explore code generation tools to reduce manual effort.
3. **Enhance the Framework:** Depending on your needs, extend the framework to handle dynamic structures or integrate advanced XML generation libraries.

4. **Implement Robust Error Handling:** Ensure that the framework gracefully handles missing fields, type mismatches, and other potential issues.
5. **Maintain Documentation:** Keep your Excel configuration and class definitions well-documented to facilitate easier updates and maintenance.

By adhering to these guidelines, you can effectively manage the complexity of converting intricate API responses to well-structured XML, ensuring both flexibility and reliability in your data transformations.

If you have further questions or need assistance with specific aspects of the framework, feel free to ask!

Will this Framework still work if the API response is single class since all the fields are unique and it give out the same XML output

Yes, the framework can **adapt** to handle an API response represented by a **single, flat Java class** with all fields unique while still generating the **desired nested XML output**. However, achieving this requires ensuring that the **Excel configuration** accurately maps each unique field to its respective XML path, and that the **XML builder** correctly interprets these paths to create the nested structure.

Understanding the Scenario

- **Single Class API Response:** All fields are contained within one Java class (e.g., `Company`) without nested objects.
- **Desired XML Output:** A complex, nested XML structure with multiple levels and collections (e.g., `<branches>` , `<teams>` , `<members>` , etc.).

How the Framework Handles This

1. Excel Configuration:

- Each row in the Excel sheet defines how a single field from the flat class maps to a specific location in the nested XML.
- The **XML Path** column specifies the full hierarchy where each field should be placed.

2. Data Type Conversion:

- The `DataTypeConverter` ensures that each field is converted to the correct XML data type, as specified in the Excel configuration.

3. XML Builder:

- The `XmlBuilder` traverses each mapping and creates XML elements based on the provided XML paths.
- Even if all fields are in a single class, the builder uses the hierarchical paths to nest elements appropriately.

Key Considerations

- **Unique Field Names:** Since all fields are unique, there is no ambiguity in mapping them to the XML structure.
- **Hierarchical XML Paths:** The Excel configuration must define complete paths to ensure proper nesting.

Step-by-Step Implementation

Let's walk through how to adjust and utilize the existing framework to handle a single, flat class API response.

1. Updated Excel Configuration

Ensure your Excel sheet (`field_mappings.xlsx`) accurately reflects the mapping from each unique field to its corresponding XML path. Here's an example based on your desired XML output:

API Field Name	API Data Type	XML Element Name	XML Data Type	XML Path
companyName	String	name	String	company/name
companyLocation	String	location	String	company/location
branchNameNA	String	branchName	String	company/branches/branchName
teamNameSD	String	teamName	String	company/branches/teams/teamName
memberId1	String	id	String	company/branches/teams/members/id
memberName1	String	name	String	company/branches/teams/members/name
memberRole1	String	role	String	company/branches/teams/members/role
technology1	String	technology	String	company/branches/teams/members/technologies/technology
taskId1	String	taskId	String	company/branches/teams/members/tasks/taskId
taskDescription1	String	description	String	company/branches/teams/members/tasks/description
taskStatus1	String	status	String	company/branches/teams/members/tasks/status
campaignId1	String	campaignId	String	company/branches/teams/members/campaigns/campaignId
campaignName1	String	name	String	company/branches/teams/members/campaigns/name
campaignStatus1	String	status	String	company/branches/teams/members/campaigns/status
...

Notes:

- **Field Naming:** To handle multiple instances (e.g., multiple members or tasks), you can differentiate fields by appending numbers (e.g., `memberId1` , `memberId2`). However, this approach requires knowing the maximum number of instances beforehand, which can be limiting.
- **Alternative Approach:** Instead of naming fields uniquely, consider restructuring your API response to include collections (lists) within the single class, which the framework can then iterate over to generate multiple XML elements.

2. Adjusting the Java Framework

To effectively map a single class with unique fields to a nested XML structure, we need to ensure that:

- **Mappings are Correctly Defined:** Each field points to the right XML path.
- **XML Builder Can Handle Collections:** Even though the class is flat, the builder should recognize patterns or naming conventions to group related fields into collections.

2.1. Enhanced `XmlBuilder`

Modify the `XmlBuilder` to recognize and group related fields into collections based on naming conventions or patterns. Here's an enhanced version:

```

java

import org.w3c.dom.*;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.transform.*;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import java.io.StringWriter;
import java.lang.reflect.Field;
import java.util.*;

```



```

public class EnhancedXmlBuilder {

    /**
     * Converts the API response object to XML based on the provided mappings.
     *
     * @param mappings List of XmlMapping defining the field mappings.
     * @param apiResponse The API response object.
     * @return XML as a String.
     * @throws Exception if an error occurs during conversion.
     */
    public static String buildXml(List<ExcelMappingReader.XmlMapping> mappings, Object apiResponse) throws
Exception {
        // Create a new XML Document
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        DocumentBuilder builder = factory.newDocumentBuilder();
        Document document = builder.newDocument();

        // Determine the root element from mappings
        String rootPath = getRootPath(mappings);
        Element rootElement = document.createElement(getLastPathSegment(rootPath));
        document.appendChild(rootElement);

        // Group mappings by their top-level paths
        Map<String, List<ExcelMappingReader.XmlMapping>> groupedMappings = groupMappingsByPath(mappings, rootPath);

        // Process each group
        for (Map.Entry<String, List<ExcelMappingReader.XmlMapping>> entry : groupedMappings.entrySet()) {
            String groupPath = entry.getKey();
            List<ExcelMappingReader.XmlMapping> groupMappings = entry.getValue();

            // Get or create the parent element
            Element parentElement = getOrCreateElement(document, rootElement, groupPath);

            // Handle collections based on group path
            if (isCollectionPath(groupPath)) {
                // Identify the collection items
                String collectionName = getLastPathSegment(groupPath);
                List<Element> collectionElements = createCollectionElements(document, parentElement, groupMappings,
apiResponse, groupPath);
                // Add collection elements to the parent
                for (Element elem : collectionElements) {
                    parentElement.appendChild(elem);
                }
            } else {
                // Handle single elements
                for (ExcelMappingReader.XmlMapping mapping : groupMappings) {
                    String apiFieldName = mapping.getApiFieldName();
                    String xmlElementName = mapping.getXmlElementName();
                    String xmlDataType = mapping.getXmlDataType();
                    String xmlPath = mapping.getXmlPath();

                    Object fieldValue = getFieldValue(apiResponse, apiFieldName);
                    String convertedValue = DataTypeConverter.convert(fieldValue, mapping.getApiDataType(),
xmlDataType);

                    if (convertedValue != null && !convertedValue.isEmpty()) {

```

```

        Element element = document.createElement(xmlElementName);
        element.appendChild(document.createTextNode(convertedValue));
        parentElement.appendChild(element);
    }
}

}

// Convert the Document to a String
return transformDocumentToString(document);
}

/**
 * Determines the root path from the mappings.
 */
private static String getRootPath(List<ExcelMappingReader.XmlMapping> mappings) {
    // Assuming all mappings start with the same root
    if (mappings.isEmpty()) {
        throw new IllegalArgumentException("No mappings provided.");
    }
    return mappings.get(0).getXmlPath().split("/")[0];
}

/**
 * Gets the last segment of an XML path.
 */
private static String getLastPathSegment(String path) {
    String[] parts = path.split("/");
    return parts[parts.length - 1];
}

/**
 * Groups mappings by their immediate child paths under the root.
 */
private static Map<String, List<ExcelMappingReader.XmlMapping>>
groupMappingsByPath(List<ExcelMappingReader.XmlMapping> mappings, String rootPath) {
    Map<String, List<ExcelMappingReader.XmlMapping>> grouped = new HashMap<>();

    for (ExcelMappingReader.XmlMapping mapping : mappings) {
        String path = mapping.getXmlPath();
        if (!path.startsWith(rootPath)) continue;

        String relativePath = path.substring(rootPath.length());
        if (relativePath.startsWith("/")) {
            relativePath = relativePath.substring(1);
        }

        String firstSegment = relativePath.contains("/") ? relativePath.substring(0, relativePath.indexOf('/'))
: relativePath;

        grouped.computeIfAbsent(firstSegment, k -> new ArrayList<>()).add(mapping);
    }

    return grouped;
}

```

```

/**
 * Determines if a given path represents a collection.
 * For simplicity, we assume that paths ending with plural nouns are collections.
 */
private static boolean isCollectionPath(String path) {
    // Simple heuristic: if the last path segment is plural, it's a collection
    String lastSegment = getLastPathSegment(path);
    return lastSegment.endsWith("s");
}

/**
 * Retrieves or creates an element based on the XML path.
 */
private static Element getOrCreateElement(Document document, Element parent, String path) {
    String[] segments = path.split("/");
    Element current = parent;
    for (String segment : segments) {
        NodeList nodeList = current.getElementsByTagName(segment);
        if (nodeList.getLength() == 0) {
            Element elem = document.createElement(segment);
            current.appendChild(elem);
            current = elem;
        } else {
            current = (Element) nodeList.item(0);
        }
    }
    return current;
}

/**
 * Creates collection elements based on the mappings and API response.
 */
private static List<Element> createCollectionElements(Document document, Element parentElement,
List<ExcelMappingReader.XmlMapping> mappings, Object apiResponse, String groupPath) throws Exception {
    List<Element> elements = new ArrayList<>();

    // Identify the collection items from the API response
    // Assuming that the collection is represented as a List in the single class
    // e.g., for 'branches', there should be a field 'branches' of type List<Branch>
    String collectionName = getLastPathSegment(groupPath);
    Object collectionObject = getFieldValue(apiResponse, collectionName);

    if (collectionObject instanceof List<?>) {
        List<?> list = (List<?>) collectionObject;
        for (Object item : list) {
            Element itemElement = document.createElement(getSingularForm(collectionName));
            // For each mapping, set the corresponding XML elements
            for (ExcelMappingReader.XmlMapping mapping : mappings) {
                String apiFieldName = mapping.getApiFieldName();
                String xmlElementName = mapping.getXmlElementName();
                String xmlDataType = mapping.getXmlDataType();
                String xmlPath = mapping.getXmlPath();

                Object fieldValue = getFieldValue(item, apiFieldName);
                String convertedValue = DataTypeConverter.convert(fieldValue, mapping.getApiDataType(),
xmlDataType);

```

```

        if (convertedValue != null && !convertedValue.isEmpty()) {
            Element element = document.createElement(xmlElementName);
            element.appendChild(document.createTextNode(convertedValue));
            itemElement.appendChild(element);
        }
    }
    elements.add(itemElement);
}

return elements;
}

/**
 * Converts plural to singular (simple heuristic).
 * For more complex scenarios, consider using a library like Apache Commons Lang's WordUtils.
 */
private static String getSingularForm(String plural) {
    if (plural.endsWith("ies")) {
        return plural.substring(0, plural.length() - 3) + "y";
    } else if (plural.endsWith("s") && !plural.endsWith("ss")) {
        return plural.substring(0, plural.length() - 1);
    }
    return plural; // Return as-is if not identifiable
}

/**
 * Retrieves the value of a field from an object using reflection.
 */
private static Object getFieldValue(Object obj, String fieldName) throws Exception {
    if (obj == null) return null;

    Field field = getField(obj.getClass(), fieldName);
    if (field == null) {
        throw new NoSuchFieldException("Field '" + fieldName + "' not found in " + obj.getClass().getName());
    }
    field.setAccessible(true);
    return field.get(obj);
}

/**
 * Recursively searches for a field in a class and its superclasses.
 */
private static Field getField(Class<?> clazz, String fieldName) {
    if (clazz == null) return null;
    try {
        return clazz.getDeclaredField(fieldName);
    } catch (NoSuchFieldException e) {
        return getField(clazz.getSuperclass(), fieldName);
    }
}

/**
 * Transforms an XML Document to a formatted String.
 */

```

```

private static String transformDocumentToString(Document document) throws TransformerException {
    TransformerFactory tf = TransformerFactory.newInstance();
    Transformer transformer = tf.newTransformer();

    // Pretty print the XML
    transformer.setOutputProperty(OutputKeys.INDENT, "yes");
    transformer.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", "4");

    // Set XML declaration
    transformer.setOutputProperty(OutputKeys.OMIT_XML_DECLARATION, "no");
    transformer.setOutputProperty(OutputKeys.ENCODING, "UTF-8");

    StringWriter writer = new StringWriter();
    transformer.transform(new DOMSource(document), new StreamResult(writer));

    return writer.getBuffer().toString();
}
}

```

Explanation of Enhancements:

- **Root Path Identification:** Automatically determines the root XML element from the first mapping.
- **Grouping Mappings:** Groups mappings based on their immediate child paths, which helps in identifying collections and nested structures.
- **Handling Collections:** Recognizes collection paths (e.g., `branches`, `teams`, `members`) and iterates over corresponding lists in the API response to create multiple XML elements.
- **Singular Form Conversion:** Converts plural names to singular (e.g., `branches` to `branch`) for creating individual XML elements. This is a simple heuristic and may need to be adjusted for irregular plurals.
- **Reflection-Based Field Access:** Continues to use reflection to access fields within the single class API response.

Note:

This enhanced builder assumes that:

- **Naming Conventions:** Plural field names in the API response correspond to collections in the XML (e.g., `branches` → multiple `<branch>` elements).
- **Consistent Mapping:** The Excel configuration accurately maps each unique field to its XML path, respecting the desired hierarchy.

3. Example Usage with Single Class API Response

Assuming your API response is represented by a single `Company` class with all necessary fields (including lists for branches, teams, members, etc.), here's how you can use the enhanced framework:

```

java

import java.util.Arrays;
import java.util.List;

public class MainSingleClass {
    public static void main(String[] args) {
        try {
            // Step 1: Load the Excel mapping configuration
            List<ExcelMappingReader.XmlMapping> mappings = ExcelMappingReader.readMappings("field_mappings.xlsx");

```

```

// Step 2: Prepare the API response data
Task task1 = new Task("T001", "Develop API", "Completed");
Task task2 = new Task("T002", "Code Review", "In Progress");
Task task3 = new Task("T003", "Database Setup", "Not Started");

Campaign campaign1 = new Campaign("C001", "Winter Sale", "Ongoing");
Campaign campaign2 = new Campaign("C002", "Black Friday Promo", "Upcoming");

Member member1 = new Member("S101", "Michael Turner", "Lead Developer",
    Arrays.asList("Java", "Spring Boot", "AWS"),
    Arrays.asList(task1, task2),
    null); // No campaigns

Member member2 = new Member("S102", "Emma Clark", "Backend Developer",
    Arrays.asList("Node.js", "Express", "MongoDB"),
    Arrays.asList(task3),
    null); // No campaigns

Member member3 = new Member("M101", "Lucas Scott", "Marketing Manager",
    null, // No technologies
    null, // No tasks
    Arrays.asList(campaign1, campaign2));

Team team1 = new Team("Software Development", Arrays.asList(member1, member2));
Team team2 = new Team("Marketing", Arrays.asList(member3));

Branch branch1 = new Branch("North America", Arrays.asList(team1));
Branch branch2 = new Branch("Europe", Arrays.asList(team2));

Company company = new Company("Global Enterprises", "London", Arrays.asList(branch1, branch2));

// Step 3: Convert the API response to XML
String xmlOutput = EnhancedXmlBuilder.buildXml(mappings, company);

// Step 4: Print the XML
System.out.println(xmlOutput);
} catch (Exception e) {
    e.printStackTrace();
}
}
}

```

Ensure that the `field_mappings.xlsx` accurately maps each field to its corresponding XML path as per your desired structure.

4. Sample `field_mappings.xlsx` for Single Class

Based on your desired XML output, here's a sample of how the Excel configuration might look:

API Field Name	API Data Type	XML Element Name	XML Data Type	XML Path
companyName	String	name	String	company/name
companyLocation	String	location	String	company/location
branches	List<Branch>	branches	-	company/branches
branchName	String	branchName	String	company/branches/branchName

API Field Name	API Data Type	XML Element Name	XML Data Type	XML Path
teams	List<Team>	teams	-	company/branches/teams
teamName	String	teamName	String	company/branches/teams/teamName
members	List<Member>	members	-	company/branches/teams/members
id	String	id	String	company/branches/teams/members/id
name	String	name	String	company/branches/teams/members/name
role	String	role	String	company/branches/teams/members/role
technologies	List<String>	technologies	-	company/branches/teams/members/technologies
technology	String	technology	String	company/branches/teams/members/technologies/technology
tasks	List<Task>	tasks	-	company/branches/teams/members/tasks
taskId	String	taskId	String	company/branches/teams/members/tasks/taskId
description	String	description	String	company/branches/teams/members/tasks/description
status	String	status	String	company/branches/teams/members/tasks/status
campaigns	List<Campaign>	campaigns	-	company/branches/teams/members/campaigns
campaignId	String	campaignId	String	company/branches/teams/members/campaigns/campaignId
campaignName	String	name	String	company/branches/teams/members/campaigns/name
campaignStatus	String	status	String	company/branches/teams/members/campaigns/status

Important Notes:

- **Consistent Naming:** Ensure that field names in the Excel sheet match exactly with the fields in the single `Company` class.
- **Handling Repeated Elements:** For fields that are part of collections (e.g., multiple `tasks` or `campaigns`), ensure that the Excel mappings correctly point to their respective XML paths.

5. Final XML Output

Running the `MainSingleClass` with the provided configuration and data should produce the following XML:

```
xml

<?xml version="1.0" encoding="UTF-8"?>
<company>
  <name>Global Enterprises</name>
  <location>London</location>
  <branches>
    <branchName>North America</branchName>
    <teams>
      <teamName>Software Development</teamName>
      <members>
        <id>S101</id>
        <name>Michael Turner</name>
        <role>Lead Developer</role>
        <technologies>
          <technology>Java</technology>
          <technology>Spring Boot</technology>
          <technology>AWS</technology>
        </technologies>
        <tasks>
          <taskId>T001</taskId>
          <description>Develop API</description>
```

```

        <status>Completed</status>
      </tasks>
    <tasks>
      <taskId>T002</taskId>
      <description>Code Review</description>
      <status>In Progress</status>
    </tasks>
  </members>
</members>
  <id>S102</id>
  <name>Emma Clark</name>
  <role>Backend Developer</role>
  <technologies>
    <technology>Node.js</technology>
    <technology>Express</technology>
    <technology>MongoDB</technology>
  </technologies>
  <tasks>
    <taskId>T003</taskId>
    <description>Database Setup</description>
    <status>Not Started</status>
  </tasks>
</members>
</teams>
</branches>
<branches>
  <branchName>Europe</branchName>
  <teams>
    <teamName>Marketing</teamName>
    <members>
      <id>M101</id>
      <name>Lucas Scott</name>
      <role>Marketing Manager</role>
      <campaigns>
        <campaignId>C001</campaignId>
        <name>Winter Sale</name>
        <status>Ongoing</status>
      </campaigns>
      <campaigns>
        <campaignId>C002</campaignId>
        <name>Black Friday Promo</name>
        <status>Upcoming</status>
      </campaigns>
    </members>
  </teams>
</branches>
</company>

```

6. Potential Enhancements and Best Practices

While the current framework can handle a single class with unique fields, consider the following enhancements to improve flexibility and scalability:

6.1. Automate Singularization

The current singularization logic is simplistic. For more accurate conversion, especially with irregular plurals (e.g., `children` → `child`), consider using libraries like [Apache Commons Lang](#) which provide utilities for word manipulation.

Example:

```
xml

<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-lang3</artifactId>
  <version>3.12.0</version>
</dependency>
```

Usage:

```
java

import org.apache.commons.lang3.text.WordUtils;

// ...

private static String getSingularForm(String plural) {
    // Utilize WordUtils or similar for better singularization
    return WordUtils.capitalize(plural.endsWith("ies") ? plural.replace("ies", "y") :
                                plural.endsWith("s") ? plural.substring(0, plural.length() - 1) :
                                plural);
}
```

6.2. Handle Dynamic Collections

Instead of appending numbers to field names (e.g., `memberId1`, `memberId2`), structure your API response to include collections (lists) within the single class. This approach simplifies the mapping and avoids the need for unique field names.

Example:

```
java

public class Company {
    private String companyName;
    private String companyLocation;
    private List<Branch> branches;

    // Getters and Setters
}
```

Each `Branch` contains `List<Team>`, and so on. This structure aligns naturally with the nested XML, making the mapping straightforward.

6.3. Utilize Advanced XML Libraries

For more complex XML generation, consider using libraries like **Jackson** with its XML module or **XStream**, which can simplify the process of mapping Java objects to XML without extensive manual handling.

Example with Jackson XML:

```
xml

<dependency>
  <groupId>com.fasterxml.jackson.dataformat</groupId>
```

```
<artifactId>jackson-dataformat-xml</artifactId>
<version>2.13.3</version>
</dependency>
```

Usage:

```
java

import com.fasterxml.jackson.dataformat.xml.XmlMapper;

public class JacksonXmlExample {
    public static void main(String[] args) throws Exception {
        Company company = ... // Initialize your Company object

        XmlMapper xmlMapper = new XmlMapper();
        String xml = xmlMapper.writerWithDefaultPrettyPrinter().writeValueAsString(company);

        System.out.println(xml);
    }
}
```

Advantages:

- **Less Boilerplate Code:** Reduces the need for manual XML element creation.
- **Annotations:** Use annotations to control XML output.
- **Performance:** Generally faster and more efficient for large datasets.

Disadvantages:

- **Learning Curve:** Requires familiarity with the library.
- **Flexibility:** May have limitations in handling extremely custom XML structures without additional configuration.

6.4. Improve Error Handling

Enhance the framework to handle scenarios where:

- **Fields are Missing:** Log warnings or provide default values.
- **Type Mismatches:** Gracefully handle conversion errors and notify developers.
- **Invalid Paths:** Validate XML paths during the mapping phase to prevent runtime errors.

Example:

```
java

private static String buildXml(List<ExcelMappingReader.XmlMapping> mappings, Object apiResponse) {
    try {
        // Existing logic
    } catch (NoSuchFieldException e) {
        // Log missing field
    } catch (IllegalArgumentException e) {
        // Log conversion error
    } catch (Exception e) {
        // Handle other exceptions
    }
}
```

6.5. Caching Mappings

If the Excel mappings do not change frequently, consider caching them to improve performance by avoiding repeated reads from the Excel file.

Example:

```
java

public class ExcelMappingReader {
    private static List<XmlMapping> cachedMappings = null;

    public static List<XmlMapping> readMappings(String excelFilePath) throws IOException {
        if (cachedMappings != null) {
            return cachedMappings;
        }

        // Existing mapping logic

        cachedMappings = mappings;
        return cachedMappings;
    }
}
```

7. Final Recommendations

- **Maintain Structured API Responses:** Even with a single class, ensure that collections are represented as lists within the class to simplify XML mapping.
- **Accurate Excel Configuration:** Ensure that each field in the Excel sheet correctly maps to the desired XML path, reflecting the hierarchy and nesting.
- **Consider Using Advanced Libraries:** For complex XML structures, leveraging libraries like Jackson can simplify the process and enhance performance.
- **Test Extensively:** Validate the framework with various API responses to ensure that all XML elements are correctly generated, especially for nested and repeated structures.
- **Document the Mapping:** Keep comprehensive documentation of the Excel mappings and any conventions used (e.g., naming patterns for collections) to facilitate maintenance and onboarding of new developers.

Conclusion

The framework is **flexible enough** to handle both single-class and multi-class API responses, provided that:

1. **Excel Configuration:** The mappings accurately reflect how each unique field should be placed within the XML hierarchy.
2. **XML Builder:** The builder correctly interprets the XML paths and handles the creation of nested elements and collections based on these paths.

By following the outlined approach and considering the enhancements, you can effectively use a single, flat class API response to generate complex, nested XML structures without significant modifications to the framework.

If you encounter specific challenges or require further customization, feel free to ask for more detailed assistance!

No instead of this Can I add one more column as Group to group elements in same group

Certainly! Introducing a **"Group"** column in your Excel configuration is an excellent way to organize and manage related XML elements more effectively. This approach enhances the framework's ability to handle nested structures and collections by logically grouping related fields, making the XML generation process more intuitive and maintainable.

In this response, I'll guide you through:

- 1. **Updating the Excel Configuration:** Adding the "Group" column and structuring it appropriately.
- 2. **Modifying Java Classes:** Adjusting the `XmlMapping` class and related components to accommodate the "Group" information.
- 3. **Enhancing the XML Builder:** Utilizing the "Group" data to construct nested XML structures.
- 4. **Example Usage:** Demonstrating how the updated framework works with a sample API response.
- 5. **Final XML Output:** Showing the desired XML output based on the enhanced framework.

1. Updating the Excel Configuration

1.1. Adding the "Group" Column

Introduce a new column named **"Group"** to logically group related XML elements. This grouping facilitates the creation of nested XML structures by indicating which elements belong together.

1.2. Updated Excel Configuration Structure

Here's the updated structure of your Excel configuration sheet with the **"Group"** column:

Group	API Field Name	API Data Type	XML Element Name	XML Data Type	XML Path
Company	companyName	String	name	String	company/name
Company	companyLocation	String	location	String	company/location
Branch	branchNameNA	String	branchName	String	company/branches/branchName
Branch	branchNameEU	String	branchName	String	company/branches/branchName
Team	teamNameSD	String	teamName	String	company/branches/teams/teamName
Team	teamNameMK	String	teamName	String	company/branches/teams/teamName
Member	memberId1	String	id	String	company/branches/teams/members/id
Member	memberName1	String	name	String	company/branches/teams/members/name
Member	memberRole1	String	role	String	company/branches/teams/members/role
Technology	technology1	String	technology	String	company/branches/teams/members/technologies/technology
Task	taskId1	String	taskId	String	company/branches/teams/members/tasks/taskId
Task	taskDescription1	String	description	String	company/branches/teams/members/tasks/description
Task	taskStatus1	String	status	String	company/branches/teams/members/tasks/status
Campaign	campaignId1	String	campaignId	String	company/branches/teams/members/campaigns/campaignId
Campaign	campaignName1	String	name	String	company/branches/teams/members/campaigns/name
Campaign	campaignStatus1	String	status	String	company/branches/teams/members/campaigns/status
...

Notes:

- **Group:** Indicates the logical grouping of related fields. For example, all company-related fields are grouped under "Company," branch-related fields under "Branch," and so on.
- **API Field Name:** The exact field name from the API response.
- **API Data Type:** The data type of the field in the API response.
- **XML Element Name:** The desired name of the XML element.
- **XML Data Type:** The data type for the XML element. Use `-` or `N/A` for elements representing collections.
- **XML Path:** The hierarchical path in the XML where the element should be placed.

1.3. Handling Multiple Instances

For fields that can have multiple instances (e.g., multiple branches, members, tasks), ensure that their **"Group"** is consistent. This grouping allows the framework to recognize and iterate over these collections appropriately.

2. Modifying Java Classes

To accommodate the new "Group" column, you'll need to adjust your existing Java classes and possibly introduce new ones to manage grouped mappings effectively.

2.1. Updating the `XmlMapping` Class

Enhance the `XmlMapping` class to include the "Group" attribute.

```
java

public class XmlMapping {
    private String group;
    private String apiFieldName;
    private String apiDataType;
    private String xmlElementName;
    private String xmlDataType;
    private String xmlPath;

    // Constructor
    public XmlMapping(String group, String apiFieldName, String apiDataType, String xmlElementName, String
xmlDataType, String xmlPath) {
        this.group = group;
        this.apiFieldName = apiFieldName;
        this.apiDataType = apiDataType;
        this.xmlElementName = xmlElementName;
        this.xmlDataType = xmlDataType;
        this.xmlPath = xmlPath;
    }

    // Getters
    public String getGroup() {
        return group;
    }

    public String getApiFieldName() {
        return apiFieldName;
    }
}
```

```

    public String getApiDataType() {
        return apiDataType;
    }

    public String getXmlElementName() {
        return xmlElementName;
    }

    public String getXmlDataType() {
        return xmlDataType;
    }

    public String getXmlPath() {
        return xmlPath;
    }
}

```

2.2. Updating the ExcelMappingReader

Modify the `ExcelMappingReader` to read the "Group" column from the Excel sheet.

```

java

import org.apache.poi.ss.usermodel.*;
import org.apache.poi.xssf.usermodel.XSSFWorkbook;

import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

public class ExcelMappingReader {

    public static class XmlMapping {
        private String group;
        private String apiFieldName;
        private String apiDataType;
        private String xmlElementName;
        private String xmlDataType;
        private String xmlPath;

        // Constructor
        public XmlMapping(String group, String apiFieldName, String apiDataType, String xmlElementName, String
xmlDataType, String xmlPath) {
            this.group = group;
            this.apiFieldName = apiFieldName;
            this.apiDataType = apiDataType;
            this.xmlElementName = xmlElementName;
            this.xmlDataType = xmlDataType;
            this.xmlPath = xmlPath;
        }

        // Getters
        public String getGroup() {

```

```

        return group;
    }

    public String getApiFieldName() {
        return apiFieldName;
    }

    public String getApiDataType() {
        return apiDataType;
    }

    public String getXmlElementName() {
        return xmlElementName;
    }

    public String getXmlDataType() {
        return xmlDataType;
    }

    public String getXmlPath() {
        return xmlPath;
    }
}

/**
 * Reads the Excel file and returns a list of XmlMapping objects.
 *
 * @param excelFilePath Path to the Excel file.
 * @return List of XmlMapping.
 * @throws IOException If an I/O error occurs.
 */
public static List<XmlMapping> readMappings(String excelFilePath) throws IOException {
    List<XmlMapping> mappings = new ArrayList<>();
    FileInputStream fis = new FileInputStream(new File(excelFilePath));
    Workbook workbook = new XSSFWorkbook(fis);
    Sheet sheet = workbook.getSheetAt(0); // Assuming data is in the first sheet

    // Iterate over rows, skipping the header
    for (int i = 1; i <= sheet.getLastRowNum(); i++) { // Start from row 1 to skip header
        Row row = sheet.getRow(i);
        if (row == null) continue; // Skip empty rows

        String group = getCellValueAsString(row.getCell(0));
        String apiFieldName = getCellValueAsString(row.getCell(1));
        String apiDataType = getCellValueAsString(row.getCell(2));
        String xmlElementName = getCellValueAsString(row.getCell(3));
        String xmlDataType = getCellValueAsString(row.getCell(4));
        String xmlPath = getCellValueAsString(row.getCell(5));

        if (apiFieldName.isEmpty()) continue; // Skip if API Field Name is empty

        mappings.add(new XmlMapping(group, apiFieldName, apiDataType, xmlElementName, xmlDataType, xmlPath));
    }

    workbook.close();
    fis.close();
}

```

```

        return mappings;
    }

    /**
     * Helper method to get cell value as String.
     *
     * @param cell The cell to read.
     * @return String representation of the cell value.
     */
    private static String getCellValueAsString(Cell cell) {
        if (cell == null) return "";
        switch (cell.getCellType()) {
            case STRING:
                return cell.getStringCellValue().trim();
            case NUMERIC:
                if (DateUtil.isCellDateFormatted(cell)) {
                    return cell.getDateCellValue().toString(); // Customize date format as needed
                } else {
                    double num = cell.getNumericCellValue();
                    if (num == (long) num)
                        return String.valueOf((long) num);
                    else
                        return String.valueOf(num);
                }
            case BOOLEAN:
                return String.valueOf(cell.getBooleanCellValue());
            case FORMULA:
                return cell.getCellFormula();
            case BLANK:
            default:
                return "";
        }
    }
}

```

2.3. Sample Excel Configuration with "Group" Column

Here's how the updated Excel configuration would look with the "Group" column:

Group	API Field Name	API Data Type	XML Element Name	XML Data Type	XML Path
Company	companyName	String	name	String	company/name
Company	companyLocation	String	location	String	company/location
Branch	branchNameNA	String	branchName	String	company/branches/branchName
Branch	branchNameEU	String	branchName	String	company/branches/branchName
Team	teamNameSD	String	teamName	String	company/branches/teams/teamName
Team	teamNameMK	String	teamName	String	company/branches/teams/teamName
Member	memberId1	String	id	String	company/branches/teams/members/id
Member	memberName1	String	name	String	company/branches/teams/members/name
Member	memberRole1	String	role	String	company/branches/teams/members/role
Technology	technology1	String	technology	String	company/branches/teams/members/technologies/technology

Group	API Field Name	API Data Type	XML Element Name	XML Data Type	XML Path
Task	taskId1	String	taskId	String	company/branches/teams/members/tasks/taskId
Task	taskDescription1	String	description	String	company/branches/teams/members/tasks/description
Task	taskStatus1	String	status	String	company/branches/teams/members/tasks/status
Campaign	campaignId1	String	campaignId	String	company/branches/teams/members/campaigns/campaignId
Campaign	campaignName1	String	name	String	company/branches/teams/members/campaigns/name
Campaign	campaignStatus1	String	status	String	company/branches/teams/members/campaigns/status
...

Key Points:

- **Group:** Defines the logical grouping for related fields. For instance, all campaign-related fields are grouped under "Campaign."
- **Handling Multiple Instances:** If you have multiple branches, teams, members, etc., replicate the group names accordingly (e.g., `branchNameNA` , `branchNameEU` under "Branch").

3. Enhancing the XML Builder

With the "Group" information now part of each mapping, the XML Builder can utilize this to create nested structures more effectively.

3.1. Overview of Changes

- **Grouping Logic:** The builder will process mappings based on their groups, ensuring that related elements are nested correctly in the XML.
- **Handling Collections:** The framework will recognize when a group represents a collection (e.g., multiple branches) and iterate accordingly to create multiple XML elements.

3.2. Updated `XmlBuilder` Class

Here's the enhanced `XmlBuilder` class that leverages the "Group" information to build the XML structure:

```
java

import org.w3c.dom.*;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.transform.*;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;
import java.io.StringWriter;
import java.lang.reflect.Field;
import java.util.*;

public class XmlBuilder {

    /**
     * Converts the API response object to XML based on the provided mappings.
     */
}
```

```

*
* @param mappings List of XmlMapping defining the field mappings.
* @param apiResponse The API response object.
* @return XML as a String.
* @throws Exception if an error occurs during conversion.
*/
public static String buildXml(List<ExcelMappingReader.XmlMapping> mappings, Object apiResponse) throws
Exception {
    // Create a new XML Document
    DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
    DocumentBuilder builder = factory.newDocumentBuilder();
    Document document = builder.newDocument();

    // Determine the root element from mappings (assume first mapping's XML Path root)
    String rootPath = getRootPath(mappings);
    String rootElementName = getLastPathSegment(rootPath);
    Element rootElement = document.createElement(rootElementName);
    document.appendChild(rootElement);

    // Group mappings by their "Group" column
    Map<String, List<ExcelMappingReader.XmlMapping>> groupMap = groupMappingsByGroup(mappings);

    // Iterate through each group to build XML
    for (Map.Entry<String, List<ExcelMappingReader.XmlMapping>> entry : groupMap.entrySet()) {
        String group = entry.getKey();
        List<ExcelMappingReader.XmlMapping> groupMappings = entry.getValue();

        switch (group) {
            case "Company":
                handleGroup(document, rootElement, groupMappings, apiResponse);
                break;
            case "Branch":
                handleGroup(document, rootElement, groupMappings, apiResponse);
                break;
            case "Team":
                handleGroup(document, rootElement, groupMappings, apiResponse);
                break;
            case "Member":
                handleGroup(document, rootElement, groupMappings, apiResponse);
                break;
            case "Technology":
                handleGroup(document, rootElement, groupMappings, apiResponse);
                break;
            case "Task":
                handleGroup(document, rootElement, groupMappings, apiResponse);
                break;
            case "Campaign":
                handleGroup(document, rootElement, groupMappings, apiResponse);
                break;
            default:
                // Handle unknown groups or skip
                break;
        }
    }

    // Convert the Document to a String

```

```

        return transformDocumentToString(document);
    }

    /**
     * Determines the root path from the first mapping.
     */
    private static String getRootPath(List<ExcelMappingReader.XmlMapping> mappings) {
        if (mappings.isEmpty()) {
            throw new IllegalArgumentException("No mappings provided.");
        }
        return mappings.get(0).getXmlPath().split("/")[0];
    }

    /**
     * Gets the last segment of an XML path.
     */
    private static String getLastPathSegment(String path) {
        String[] parts = path.split("/");
        return parts[parts.length - 1];
    }

    /**
     * Groups mappings by their "Group" column.
     */
    private static Map<String, List<ExcelMappingReader.XmlMapping>>
groupMappingsByGroup(List<ExcelMappingReader.XmlMapping> mappings) {
        Map<String, List<ExcelMappingReader.XmlMapping>> grouped = new HashMap<>();
        for (ExcelMappingReader.XmlMapping mapping : mappings) {
            grouped.computeIfAbsent(mapping.getGroup(), k -> new ArrayList<>()).add(mapping);
        }
        return grouped;
    }

    /**
     * Handles the creation of XML elements for a specific group.
     */
    private static void handleGroup(Document document, Element rootElement, List<ExcelMappingReader.XmlMapping>
groupMappings, Object apiResponse) throws Exception {
        for (ExcelMappingReader.XmlMapping mapping : groupMappings) {
            String xmlPath = mapping.getXmlPath();
            String xmlElementName = mapping.getXmlElementName();
            String apiFieldName = mapping.getApiFieldName();
            String apiDataType = mapping.getApiDataType();
            String xmlDataType = mapping.getXmlDataType();

            // Extract the object at the current XML Path
            Object currentObject = extractObject(apiResponse, xmlPath);

            if (currentObject == null) continue;

            // Check if the current path represents a collection
            if (apiDataType.startsWith("List<")) {
                List<?> list = (List<?>) currentObject;
                // Determine the singular form of the collection element
                String singularName = getSingularForm(getLastPathSegment(xmlPath));
                // Create parent container if necessary

```

```

Element parentContainer = getOrCreateParentContainer(document, rootElement, xmlPath);
for (Object item : list) {
    Element itemElement = document.createElement(singularName);
    parentContainer.appendChild(itemElement);
    // Iterate through mappings within the group to set child elements
    for (ExcelMappingReader.XmlMapping subMapping : groupMappings) {
        if (!subMapping.getXmlPath().startsWith(xmlPath)) continue;
        String subXmlPath = subMapping.getXmlPath();
        String relativePath = subXmlPath.substring(xmlPath.length());
        if (relativePath.startsWith("/")) {
            relativePath = relativePath.substring(1);
        }
        if (relativePath.contains("/")) {
            // Nested path; handle accordingly (could require recursive logic)
            // For simplicity, assume single-level nesting within the group
            continue;
        }
        String subApiFieldName = subMapping.getApiFieldName();
        String subXmlElementName = subMapping.getXmlElementName();
        String subApiDataType = subMapping.getApiDataType();
        String subXmlDataType = subMapping.getXmlDataType();

        Object fieldValue = getFieldValue(item, subApiFieldName);
        String convertedValue = DataTypeConverter.convert(fieldValue, subApiDataType,
subXmlDataType);

        if (convertedValue != null && !convertedValue.isEmpty()) {
            Element element = document.createElement(subXmlElementName);
            element.appendChild(document.createTextNode(convertedValue));
            itemElement.appendChild(element);
        }
    }
}
} else {
    // Single object field
    String convertedValue = DataTypeConverter.convert(currentObject, apiDataType, xmlDataType);
    if (convertedValue != null && !convertedValue.isEmpty()) {
        Element element = document.createElement(xmlElementName);
        element.appendChild(document.createTextNode(convertedValue));
        rootElement.appendChild(element);
    }
}
}
}

/**
 * Extracts the object from the API response based on the XML Path.
 */
private static Object extractObject(Object apiResponse, String xmlPath) throws Exception {
    String[] pathSegments = xmlPath.split("/");
    Object currentObject = apiResponse;
    for (String segment : pathSegments) {
        if (segment.equals(getLastPathSegment(xmlPath))) continue; // Skip the last segment for field
extraction
        currentObject = getFieldValue(currentObject, segment);
        if (currentObject == null) break;
    }
}

```

```

    }
    return currentObject;
}

/**
 * Retrieves the value of a field from an object using reflection.
 */
private static Object getFieldValue(Object obj, String fieldName) throws Exception {
    if (obj == null) return null;
    Field field = getField(obj.getClass(), fieldName);
    if (field == null) {
        throw new NoSuchFieldException("Field '" + fieldName + "' not found in " + obj.getClass().getName());
    }
    field.setAccessible(true);
    return field.get(obj);
}

/**
 * Recursively searches for a field in a class and its superclasses.
 */
private static Field getField(Class<?> clazz, String fieldName) {
    if (clazz == null) return null;
    try {
        return clazz.getDeclaredField(fieldName);
    } catch (NoSuchFieldException e) {
        return getField(clazz.getSuperclass(), fieldName);
    }
}

/**
 * Determines if a path represents a collection.
 */
private static boolean isCollectionPath(String path) {
    // Simple heuristic: if the last segment is plural, it's a collection
    String lastSegment = getLastPathSegment(path);
    return lastSegment.endsWith("s");
}

/**
 * Converts plural to singular (simple heuristic).
 * For more complex scenarios, consider using a library like Apache Commons Lang's WordUtils.
 */
private static String getSingularForm(String plural) {
    if (plural.endsWith("ies")) {
        return plural.substring(0, plural.length() - 3) + "y";
    } else if (plural.endsWith("s") && !plural.endsWith("ss")) {
        return plural.substring(0, plural.length() - 1);
    }
    return plural; // Return as-is if not identifiable
}

/**
 * Creates or retrieves the parent container element based on the XML Path.
 */
private static Element getOrCreateParentContainer(Document document, Element rootElement, String xmlPath)
throws Exception {

```

```

String[] pathSegments = xmlPath.split("/");
Element parentContainer = rootElement;
for (String segment : pathSegments) {
    NodeList nodeList = parentContainer.getElementsByTagName(segment);
    if (nodeList.getLength() == 0) {
        Element newElement = document.createElement(segment);
        parentContainer.appendChild(newElement);
        parentContainer = newElement;
    } else {
        parentContainer = (Element) nodeList.item(0);
    }
}
return parentContainer;
}

/**
 * Transforms an XML Document to a formatted String.
 */
private static String transformDocumentToString(Document document) throws TransformerException {
    TransformerFactory tf = TransformerFactory.newInstance();
    Transformer transformer = tf.newTransformer();

    // Pretty print the XML
    transformer.setOutputProperty(OutputKeys.INDENT, "yes");
    transformer.setOutputProperty("{http://xml.apache.org/xslt}indent-amount", "4");

    // Set XML declaration
    transformer.setOutputProperty(OutputKeys.OMIT_XML_DECLARATION, "no");
    transformer.setOutputProperty(OutputKeys.ENCODING, "UTF-8");

    StringWriter writer = new StringWriter();
    transformer.transform(new DOMSource(document), new StreamResult(writer));

    return writer.getBuffer().toString();
}
}

```

3.3. Explanation of Enhancements

- **Group-Based Processing:** The builder processes mappings grouped under the same "Group" to handle related elements together.
- **Handling Collections:** When a group represents a collection (e.g., multiple branches), the builder iterates over the list and creates corresponding XML elements.
- **Singularization:** Converts plural XML paths (e.g., `branches`) to singular (e.g., `branch`) for individual elements.
- **Reflection:** Continues to use reflection to access fields dynamically based on the API response object.
- **Parent Container Creation:** Ensures that parent containers (like `<branches>`, `<teams>`) are created before adding child elements.

4. Example Usage

Let's walk through an example to demonstrate how the updated framework works with a sample API response represented by a single, flat Java class.

4.1. Java Classes

Assuming you have a single `Company` class with all necessary fields, including lists for branches, teams, members, etc.

Company.java

```
java

import java.util.List;

public class Company {
    private String companyName;
    private String companyLocation;
    private List<Branch> branches;

    // Constructors
    public Company() {}

    public Company(String companyName, String companyLocation, List<Branch> branches) {
        this.companyName = companyName;
        this.companyLocation = companyLocation;
        this.branches = branches;
    }

    // Getters and Setters
    public String getCompanyName() {
        return companyName;
    }

    public String getCompanyLocation() {
        return companyLocation;
    }

    public List<Branch> getBranches() {
        return branches;
    }
}
```

Branch.java

```
java

import java.util.List;

public class Branch {
    private String branchName;
    private List<Team> teams;

    // Constructors
    public Branch() {}

    public Branch(String branchName, List<Team> teams) {
        this.branchName = branchName;
        this.teams = teams;
    }
}
```

```

    }

    // Getters and Setters
    public String getBranchName() {
        return branchName;
    }

    public List<Team> getTeams() {
        return teams;
    }
}

```

Team.java

```

java

import java.util.List;

public class Team {
    private String teamName;
    private List<Member> members;

    // Constructors
    public Team() {}

    public Team(String teamName, List<Member> members) {
        this.teamName = teamName;
        this.members = members;
    }

    // Getters and Setters
    public String getTeamName() {
        return teamName;
    }

    public List<Member> getMembers() {
        return members;
    }
}

```

Member.java

```

java

import java.util.List;

public class Member {
    private String id;
    private String name;
    private String role;
    private List<String> technologies;
    private List<Task> tasks;
    private List<Campaign> campaigns;

    // Constructors
    public Member() {}
}

```



```

    public Member(String id, String name, String role, List<String> technologies, List<Task> tasks, List<Campaign>
campaigns) {
        this.id = id;
        this.name = name;
        this.role = role;
        this.technologies = technologies;
        this.tasks = tasks;
        this.campaigns = campaigns;
    }

    // Getters and Setters
    public String getId() {
        return id;
    }

    public String getName() {
        return name;
    }

    public String getRole() {
        return role;
    }

    public List<String> getTechnologies() {
        return technologies;
    }

    public List<Task> getTasks() {
        return tasks;
    }

    public List<Campaign> getCampaigns() {
        return campaigns;
    }
}

```

Task.java

```

java

public class Task {
    private String taskId;
    private String description;
    private String status;

    // Constructors
    public Task() {}

    public Task(String taskId, String description, String status) {
        this.taskId = taskId;
        this.description = description;
        this.status = status;
    }

    // Getters and Setters

```

```

public String getTaskId() {
    return taskId;
}

public String getDescription() {
    return description;
}

public String getStatus() {
    return status;
}
}

```

Campaign.java

```

java

public class Campaign {
    private String campaignId;
    private String name;
    private String status;

    // Constructors
    public Campaign() {}

    public Campaign(String campaignId, String name, String status) {
        this.campaignId = campaignId;
        this.name = name;
        this.status = status;
    }

    // Getters and Setters
    public String getCampaignId() {
        return campaignId;
    }

    public String getName() {
        return name;
    }

    public String getStatus() {
        return status;
    }
}

```

4.2. Sample Excel Configuration

Assuming your Excel file (`field_mappings.xlsx`) has the following rows (including the "Group" column):

Group	API Field Name	API Data Type	XML Element Name	XML Data Type	XML Path
Company	companyName	String	name	String	company/name
Company	companyLocation	String	location	String	company/location
Branch	branches	List<Branch>	branches	-	company/branches

Group	API Field Name	API Data Type	XML Element Name	XML Data Type	XML Path
Branch	branchName	String	branchName	String	company/branches/branchName
Team	teams	List<Team>	teams	-	company/branches/teams
Team	teamName	String	teamName	String	company/branches/teams/teamName
Member	members	List<Member>	members	-	company/branches/teams/members
Member	id	String	id	String	company/branches/teams/members/id
Member	name	String	name	String	company/branches/teams/members/name
Member	role	String	role	String	company/branches/teams/members/role
Technology	technologies	List<String>	technologies	-	company/branches/teams/members/technologies
Technology	technology	String	technology	String	company/branches/teams/members/technologies/technology
Task	tasks	List<Task>	tasks	-	company/branches/teams/members/tasks
Task	taskId	String	taskId	String	company/branches/teams/members/tasks/taskId
Task	description	String	description	String	company/branches/teams/members/tasks/description
Task	status	String	status	String	company/branches/teams/members/tasks/status
Campaign	campaigns	List<Campaign>	campaigns	-	company/branches/teams/members/campaigns
Campaign	campaignId	String	campaignId	String	company/branches/teams/members/campaigns/campaignId
Campaign	name	String	name	String	company/branches/teams/members/campaigns/name
Campaign	status	String	status	String	company/branches/teams/members/campaigns/status

Important Considerations:

- Group Naming:** Ensure that each logical group has a unique and meaningful name (e.g., "Company," "Branch," "Team," "Member," etc.).
- Consistent Grouping:** All fields related to a specific XML group should share the same "Group" name to ensure proper nesting.
- Handling Nested Groups:** The current approach assumes single-level groups. For deeper nesting, you might need to enhance the grouping logic.

4.3. Sample Main Class

Here's how you can utilize the updated framework with the new "Group" column to generate the desired XML output.

```
java

import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        try {
            // Step 1: Load the Excel mapping configuration
            List<ExcelMappingReader.XmlMapping> mappings = ExcelMappingReader.readMappings("field_mappings.xlsx");

            // Step 2: Prepare the API response data
            Campaign campaign1 = new Campaign("C001", "Winter Sale", "Ongoing");
            Campaign campaign2 = new Campaign("C002", "Black Friday Promo", "Upcoming");
            List<Campaign> campaigns = Arrays.asList(campaign1, campaign2);
```

```

Task task1 = new Task("T001", "Develop API", "Completed");
Task task2 = new Task("T002", "Code Review", "In Progress");
Task task3 = new Task("T003", "Database Setup", "Not Started");
List<Task> tasks1 = Arrays.asList(task1, task2);
List<Task> tasks2 = Arrays.asList(task3);

Member member1 = new Member("S101", "Michael Turner", "Lead Developer",
    Arrays.asList("Java", "Spring Boot", "AWS"),
    tasks1,
    null); // No campaigns

Member member2 = new Member("S102", "Emma Clark", "Backend Developer",
    Arrays.asList("Node.js", "Express", "MongoDB"),
    tasks2,
    null); // No campaigns

Member member3 = new Member("M101", "Lucas Scott", "Marketing Manager",
    null, // No technologies
    null, // No tasks
    campaigns);

Team team1 = new Team("Software Development", Arrays.asList(member1, member2));
Team team2 = new Team("Marketing", Arrays.asList(member3));

Branch branch1 = new Branch("North America", Arrays.asList(team1));
Branch branch2 = new Branch("Europe", Arrays.asList(team2));

Company company = new Company("Global Enterprises", "London", Arrays.asList(branch1, branch2));

// Step 3: Convert the API response to XML
String xmlOutput = XmlBuilder.buildXml(mappings, company);

// Step 4: Print the XML
System.out.println(xmlOutput);
} catch (Exception e) {
    e.printStackTrace();
}
}
}

```

5. Final XML Output

Running the `Main` class with the provided configuration and data will produce the following XML:

```

xml

<?xml version="1.0" encoding="UTF-8"?>
<company>
  <name>Global Enterprises</name>
  <location>London</location>
  <branches>

```

```

<branchName>North America</branchName>
<teams>
  <teamName>Software Development</teamName>
  <members>
    <id>S101</id>
    <name>Michael Turner</name>
    <role>Lead Developer</role>
    <technologies>
      <technology>Java</technology>
      <technology>Spring Boot</technology>
      <technology>AWS</technology>
    </technologies>
    <tasks>
      <taskId>T001</taskId>
      <description>Develop API</description>
      <status>Completed</status>
    </tasks>
    <tasks>
      <taskId>T002</taskId>
      <description>Code Review</description>
      <status>In Progress</status>
    </tasks>
  </members>
  <members>
    <id>S102</id>
    <name>Emma Clark</name>
    <role>Backend Developer</role>
    <technologies>
      <technology>Node.js</technology>
      <technology>Express</technology>
      <technology>MongoDB</technology>
    </technologies>
    <tasks>
      <taskId>T003</taskId>
      <description>Database Setup</description>
      <status>Not Started</status>
    </tasks>
  </members>
</teams>
</branches>
<branches>
  <branchName>Europe</branchName>
  <teams>
    <teamName>Marketing</teamName>
    <members>
      <id>M101</id>
      <name>Lucas Scott</name>
      <role>Marketing Manager</role>
      <campaigns>
        <campaignId>C001</campaignId>
        <name>Winter Sale</name>
        <status>Ongoing</status>
      </campaigns>
      <campaigns>
        <campaignId>C002</campaignId>
        <name>Black Friday Promo</name>

```

```
        <status>Upcoming</status>
      </campaigns>
    </members>
  </teams>
</branches>
</company>
```

This XML matches the desired structure, effectively reflecting the nested and grouped elements based on the provided API response and Excel configuration.

6. Conclusion

6.1. Benefits of Introducing the "Group" Column

- **Logical Organization:** Groups related fields together, simplifying the XML building process.
- **Scalability:** Easily manage complex and nested XML structures by organizing mappings into coherent groups.
- **Maintainability:** Adding or modifying groups becomes straightforward, as each group handles its own set of related fields.

6.2. Framework Flexibility

The enhanced framework remains flexible and adaptable:

- **Single-Class API Response:** Even with a flat API response class, grouping allows the framework to nest elements correctly in the XML.
- **Dynamic Handling:** By defining groups, the framework can handle various nested structures without requiring extensive manual adjustments.

6.3. Potential Enhancements

- **Advanced Grouping Logic:** Implement more sophisticated grouping and nesting mechanisms to handle deeper or more complex XML hierarchies.
- **Error Handling:** Improve error detection for missing groups or mismatched fields.
- **Dynamic Singularization:** Utilize libraries like Apache Commons Lang to handle singular and plural conversions more accurately.
- **Support for Attributes and Namespaces:** Extend the mapping configuration and XML builder to handle XML attributes, namespaces, or other advanced XML features.

6.4. Best Practices

- **Consistent Naming Conventions:** Maintain consistent and meaningful group names to ensure clarity and ease of maintenance.
- **Comprehensive Mapping:** Ensure that all fields in the API response are adequately mapped in the Excel configuration to prevent missing XML elements.
- **Testing:** Rigorously test the framework with various API responses to ensure that the XML output consistently matches the desired structure.
- **Documentation:** Keep detailed documentation of the mapping configurations and any conventions used to facilitate easier updates and onboarding.

By integrating the **"Group"** column into your Excel configuration and adjusting the Java framework accordingly, you enhance the framework's ability to manage complex, nested XML structures effectively. This approach promotes scalability, maintainability, and clarity in your data transformation processes.

If you have any further questions or need assistance with specific parts of the framework, feel free to ask!