Database mini-project report

BERDNYK Mariia
DUONG Thi Thanh Tu

For our project, we are using HelloFresh as our model application. Therefore, we have configured our XML to encompass all the key features found in HelloFresh. These include weekly meal choices, a variety of recipes categorized differently, feedback options, delivery details and user commands.

XML model

In our XML database, we have adopted a specific structuring approach to ensure efficiency and consistency across our data. Here's an overview:

Conception :

The most notable thing to note is that we have structured some elements to be stored in separate nodes so that they can be "reusable". For example, the same ingredient together with its name and image can be included in different recipes. To establish references between different parts of the database, we utilize unique ID references. These reusable values include ingredients, categories, tags, utensils, allergens, meals, users, difficulties, price types, and nutritional components. Also, for some values we introduced the enums values in our XML schema and DTD. For instance, the user can leave only 1-4 as the feedback mark.

To start with, we have the root element *hello_fresh* that contains all elements in our whole model including the lists of *meals* and the other element lists for the meal (*ingredients*, *allergens, tags*, *categories*, *price types, nutritional components, utensils, difficulties)* and also other elements such as *weekly menus*, *users*, *commands*, *feedbacks, favorites, deliveries*.

Our most important element, *meal* consists of :

- General information : *id, name, description, headline*
- *image*: With the image element that is used also in ingredient, instruction, etc
- *tags* and *categories*: tags are dedicated to describe the nature of the recipe (vegan, gluten-free, protein smart, ...) meanwhile the categories are divided into 4 different categories (dish, culture, particular, plan) for the easier filtering of recipes. Category dish is based on the type of dish (pasta, bowl dish, meatball, ...), meanwhile culture is based on the culture of the dish (Italian, Mexican, Indian, …), particular is the particularity of the dish (Most recent, Kid-friendly, …) and the plan is used for better suggestion of meals based on the user's liking (Meat & Veggie, Veggie, Family Friendly, …)
- *recipe*: contains the cooking time, level of difficulty, list of ingredients' references and steps of cooking process
- *utensils*: references of utensils used in this meal with the references from the list of utensils
- *price type*: We have 3 different price types: normal, prime and luxury. The normal one is the price of most meals. There are some meal that you would need to pay some extra money to get (example: €2.5 for prime and €4.5 for luxury one)

- *nutritional value*: having the attribute of value together with the reference of the nutritional component containing the unit and name.

Our *weekly menu* contains the start date, end date together with the list of meals of that week.

Our *command* consists of:

- *user*: reference of the user whom the command belongs to
- *meal list*: a list of references of meals that the user chose
- *serving number*: so that we can easily know the amount of ingredients needed by multiplying the amount of ingredients in the meal's recipe with this element
- Pricing elements: *total price, discount, final total*.
  Total price is calculated by the sum of value of the price type in the meal list.
- *delivery*: reference of the delivery containing the date chosen by the user, status (confirmed/preparing/delivered/…) and the reference of the user's address
- *command date*: The date where that command was validated

Advantage : One notable advantage of our approach is the minimization of data repetition. By segregating unique dish-specific values from reusable ones, we avoid redundancy and ensure efficient storage of information. This not only conserves storage space but also enhances data consistency and simplifies maintenance tasks. Also, we always tried our nodes/elements/attributes to be atomic (only one value for each row in the table). **Disadvantage :** However, a disadvantage of this approach is the inherent complexity introduced by the utilization of ID references. Additionally, the need to consistently update and synchronize IDs across different parts of the database could introduce overhead and increase the likelihood of inconsistencies if not managed meticulously. Thus, while our approach reduces redundancy, it may necessitate careful handling and robust management of ID references to mitigate complexity-related challenges.

Scenarios

We have developed 6 scenarios. The first 3 are interconnected in HTML web format. The 4th scenario selects values and generates a detailed file depending on the input in XML format, while the 5th scenario is in JSON format. The 6th scenario stands apart from the previous 5 and traverses through all the XML to retrieve values that are difficult to obtain using a standard XSL/XSLT template. The scenarios are as follows:

1. List all the dishes & their recipes.
2. List all the weekly menus.
3. Display recipes by categories.
4. Select commands that have been validated on a specific day.
5. Select meals depending on the budget.
6. Calculate the average rating of the meals and combine 2 dishes with the highest average mark to fit in the nutritional diet provided by the user (+- 5 % measurable error).

In scenarios 4 and 5, we create different attributes and nodes.

Scenario 1 & 2

In scenario 1, which is structured for displaying meals and their recipes on a webpage, the XSLT scope includes:

1. **XPath** (e.g. *select="recipe/difficulty_ref/@idref"*)**:** XPath is used extensively to navigate and select specific elements within the XML document being transformed.
2. **XSLT Templates**: The XSLT stylesheet contains many templates (*<xsl:template>*) that define how different parts of the XML document should be transformed into HTML. They help us organize the transformation logic in a structured manner. Each template defines how a specific element or group of elements in the XML source should be transformed. It facilitates the HTML generation.
3. **Conditional Rendering**: Conditional statements (*<xsl:if>, <xsl:choose>*) are chosen to conditionally render certain elements or data based on the presence or absence of specific XML elements or attributes. For example, displaying the specific nutritional values (energy) only if they are provided in the XML data.
4. **Key-based Lookup**: The stylesheet utilizes key-based lookup (*<xsl:key>*) to efficiently retrieve related data from the XML document. Keys are defined for elements which are connected with other nodes by id, such as difficulty levels, ingredients, allergens, utensils etc.
5. **JS functions**: functions such as *concat()*, *position()* etc are used to manipulate data and control the transformation process. For example, in a recipe, *concat()* is used to concatenate the value and unit of the total time.

The disadvantage of the XSL scenario 1 might be the usage of XPath Expressions for the recipe utils (such as prep_time and total_time) in a very long string: *"concat(recipe/total_time/time_amount/@value, ' ', recipe/total_time/time_amount/@unit)"*. It does not look so aesthetic, readable and reusable.

The *scenario 2* does not contain anything new from the previous scenario.

Scenario 3

In scenario 3, which involves displaying meal categories and their respective meals, along with the capability for users to view meal details, several new XSLT elements are utilized to achieve this functionality:

1. **Iteration** (*<xsl:for-each select="//category[not(@type = preceding-sibling::category/@type)]">*): This loop iterates over unique category types. It ensures that only distinct category types are processed, avoiding duplication.
2. **Sorting** (*<xsl:sort select="@type"/>*): Inside the loop, categories are sorted based on their attribute in alphabetical order.
3. **Variable Assignment** (*<xsl:variable name="currentType" select="@type"/>*): This element assigns the current category type to a variable for later selection within the loop.

Scenario 4

In scenario 4, the XSLT code is designed to select commands made on a specific date from an XML document. Here are the new XSLT elements used for this scenario:

1. **Copying Attributes** (*<xsl:copy-of select="@*"/>*): This instruction copies attributes from one element to another. It's used to preserve attributes when restructuring the new XML document instance.

Scenario 5

In scenario 5, the XSLT code is tailored to select meals based on a specified budget from an XML document. Here are the new XSLT elements used for this scenario:

1. **Stripping Space** (*<xsl:strip-space elements="*"/>*): This tag perceives the spaces created in XSLT file format and improves readability of the resulting json file.

JSON Schema

We built our JSON schema before starting to write the scenario, and that was the correct decision. The advantages of our JSON schema include:

1. **Modularity and Reusability**: The schema utilizes the "*definitions*" section to define reusable schema fragments for various components of meal data, which are referenced from different parts of the meal (e.g. amount).
2. **Validation**: JSON schemas can be used for data validation, ensuring that meal data adheres to the specified structure and constraints. This helps to maintain data integrity and quality. We ensure this by specifying which properties are required and which type of the properties is expected.
3. **Extensibility**: The schema can be easily extended or modified to accommodate new requirements or changes in the data model without affecting existing implementations.

The obvious disadvantage, which is common for all complex systems is complexity of References: The extensive use of references ($ref) in the schema, especially for ingredients, utensils, categories, and tags, can introduce complexity, making the schema more challenging to understand and manage.

Scenario 6

For scenario 6, which is calculating the average depending on feedback to each meal and combining 2 dishes to fit in the nutritional diet provided by the user, we selected the Document Object Model (DOM) parser. DOM is an official recommendation of the World Wide Web Consortium (W3C). It defines an interface that enables programs to access and update the style, structure, and contents of XML documents.

The idea of the parser is to get back a tree structure that contains all the elements of your document. The advantages of DOM are a common interface for manipulating document structures. One of its design goals is that Java code written for one DOM-compliant parser should run on any other DOM-compliant parser without having to do any modifications. So, for our scenario, what we do is:

1. We retrieve the list of all meals (meal_list tag) and the list of feedbacks (feedbacks_list);
2. We calculate the average from feedback and sort the resulting values;
3. We iterate the list of sorted meals, generate combinations of 2 meals and check if the sum of all their nutritions fits the provided diet (diet.json file inside the java-scenario directory) (+- 5 % measurable error). If yes, we print this tuple of meals.