Documentation

Publicly exposed methods

BasketSplitter

Private parts

JsonLoader

Product

BasketSplitter Documentation

The `BasketSplitter` class is a crucial component of the Java application designed to split a shopping basket into several groups based on the delivery options available for the products in the basket. This class utilizes product configuration loaded from a JSON file to organize the items into optimal groups, ensuring that items with common delivery options are grouped together.

Purpose

The `BasketSplitter` is designed to:

- Load product configurations using the `JsonLoader` class.
- Split a list of product names into multiple baskets based on the delivery options available for each product.
- Ensure that each basket is optimized to contain products with the same delivery option, minimizing the number of separate deliveries, at the same time maximizing the number of products in single delivery.

Installation

In order to install Fat Jar as a maven dependency you can execute this command:

```
mvn install:install-file \
-Dfile=<Path to BasketSplitter.jar> \
-DgroupId=com.ocado.basket \
-DartifactId=BasketSplitter \
-Dversion=1.0 \
-Dpackaging=jar
```

And then you can add dependencies. For example in maven:

```
<dependency>
    <groupId>com.ocado.basket</groupId>
    <artifactId>BasketSplitter</artifactId>
    <version>1.0</version>
</dependency>
```

Usage Example

```
String configFile = "/path/to/config.json";
List<String> itemsToSplit = Arrays.asList("Milk", "Bread", "Apple");

BasketSplitter splitter = new BasketSplitter(configFile);
Map<String, List<String>> baskets = splitter.split(itemsToSplit);

for(Map.Entry<String, List<String>> basket : baskets.entrySet()) {
    System.out.println("Delivery Option: " + basket.getKey()
    + " contains products: " + basket.getValue());
}
```

This example demonstrates how to use the `BasketSplitter` class to divide a list of product names into optimized baskets based on their delivery options. The resulting baskets minimize the number of separate deliveries by grouping items with common delivery options together. The `BasketSplitter` effectively reduces logistical complexity and potentially delivery costs, making it an essential tool for eCommerce platforms and delivery system optimizations.

How it Works

1. **Configuration Loading**: The constructor initializes the class with a path to a configuration file, which is loaded to form a list of products and their delivery options.
2. **Splitting Process**: Given a list of product names, the `split` method organizes these products into separate baskets, ensuring each basket maximizes the number of products sharing the same delivery option.

Public Methods

Constructor

- **Purpose**: Initializes the `BasketSplitter` with product configurations.
- **Signature**: `public BasketSplitter(String absolutePathToConfigFile)`
- **Input**: `String absolutePathToConfigFile` - path to configuration file.

split

- **Purpose**: Splits the provided list of product names into optimized baskets based on delivery options.
- **Signature**: `public Map<String, List<String>> split(List<String> items)`
- **Input**: `List<String> items` - A list of product names to be split.
- **Output**: `Map<String, List<String>>` - A mapping of delivery options to lists of product names.
- **Exceptions**:
  - `BasketSplitter(...)` constructor may produce IOException due to JsonLoader implementation.
- **Logic**:
  - Loads the products to be split from the provided list.
  - Creates DeliveryFrequency Map based on provided products
  - As long as we have any product that were not yet computed (or the delivery option is not null), we perform further splitting.
    - In every iteration we update our `mostFrequentDelivery` which is delivery option that is present in the largest amount of Products.
    - Iterates through Products list and checks if it is possible to deliver it `mostFrequentDelivery` way.
      - If so, adds this particular Product to output list, and deletes it from the list.
      - Hence, we have removed product form list, we have to update our DeliveryFrequencyMap, and decrease value of every key (Delivery method) which was present in recently removed Product.
    - Puts our Delivery option name, and output lists of products into our final `outputBasket`.

    - Updates our `mostFrequentDelivery`
  - Returns our final output basket.

Private Methods

mapMaxValueKey

- **Purpose**: Finds the key with the maximum value in a given `HashMap<String, Integer>`.
- **Signature**: `private String mapMaxValueKey(Map<String, Integer> hashMap)`.
- **Input**: `Map<String, Integer> hashMap` - HashMap to operate on.
- **Output**: `String` - The key with the highest value, or `null` if the map is empty.
- **Logic**:
  - Finds first maximum value in HashMap using `stream()` and `Comparator`
  - Returns the key associated with maximum value.

generateFrequencyMap

- **Purpose**: Counts how many items have each specific delivery option available.
- **Signature**: `private Map<String, Integer> generateFrequencyMap(List<Product> productsToSplit)`
- **Input**: `List<Product> productsToSplit` - a list of products which is used to initialize our HashMap.
- **Output**: `Map<String, Integer>` - A mapping of delivery options to their occurrence count in the provided products list.
- **Logic**:
  - Initializes new empty `HashMap<String, Integer>`.
  - For each product in `productsToSplit` checks its delivery options, which is taken as a key for our HashMap.
    - If entry with such key doesn't exist, meaning we have not yet computed product witch such delivery option, create new entry with initial value of 1.
    - If entry witch such key already exists, which implies that we have already added it to our map, we add new product, by increasing value by 1.
  - Return our HashMap.

loadProductsToSplit

- **Purpose**: Changes list of Strings into a list of `Product` objects.
- **Signature**: `private List<Product> loadProductsToSplit(List<String> items)`.
- **Input**: `List<String> items` - The names of the products to be split.

- **Output**: `List<Product>` - The filtered list of `Product` objects corresponding to the provided names.
- **Logic**:
  - Initializes new empty `List<Product>`.
  - For each entry in config checks if provided `items` list contains such an entry.
    - if so - adds it to output list.
  - Returns list of `Product` objects.

JsonLoader Documentation

Overview

The `JsonLoader` class is a utility class designed to facilitate the loading of product configurations from a JSON file within the context of a shopping basket application. This class is part of the `com.ocado.basket` package and leverages the Jackson library to deserialize JSON content into Java objects. The primary functionality of this class is encapsulated in the `loadConfig` method, which reads a JSON file from a given path and translates it into a list of `Product` objects.

Key Features

- **JSON Deserialization**: Converts JSON data into Java objects, specifically into a list of `Product` instances.
- **File Handling**: Reads JSON content directly from a file system using a file path.
- **Error Handling**: Implements basic error handling by printing the stack trace of any exceptions that occur during the file reading and JSON parsing processes.

Method Detail

`loadConfig`

The `loadConfig` method is the core of the `JsonLoader` class, designed to load and parse a JSON file that contains product configurations.

Signature

```
public static List<Product> loadConfig(String JSONPath)
```

Parameters

- `JSONPath`: A `String` representing the file system path to the JSON file that contains the product configurations.

Returns

- A `List<Product>`: This list contains `Product` objects instantiated from the JSON file. Each `Product` represents a product configuration as defined in the JSON.

Exceptions

- If an exception occurs during the reading or parsing process, method produces IOException

Implementation Details

1. **Initialization**: A new empty list of `Product` objects is created.
2. **ObjectMapper Creation**: An instance of `ObjectMapper` from the Jackson library is instantiated to handle the deserialization of the JSON file.
3. **Reading and Parsing JSON**:
    - The method attempts to read the JSON file located at `JSONPath` and parse it into a `Map<String, List<String>>`. Each entry in the map represents a product, with the key being the product name and the value being a list of strings related to the product (possibly representing attributes or categories).
    - It iterates through each entry in the map, creating a new `Product` object with the key as the product name and the value as the product's attributes or categories. Each `Product` object is then added to the list of products.
4. **Returns list of products objects**

Dependencies

- **Jackson Databind**: This class requires the Jackson library to deserialize JSON content. The `ObjectMapper` class from `com.fasterxml.jackson.databind` is used for this purpose.

Summary

The `JsonLoader` class provides a straightforward and efficient way to load product configurations from a JSON file, translating them into a list of `Product` objects that can be used within a shopping basket application. By leveraging the Jackson library for JSON parsing, this class abstracts the complexities of file and data handling, offering a simple interface for loading product data.

Product Documentation

The `Product.java` file is a crucial component of a Java application designed to manage and represent products, particularly focusing on their delivery options.

Overview

`Product` is a simple Java record that encapsulates two pieces of information about a product:

- **Product Name**
- **Delivery Options**

It provides a straightforward way to instantiate product objects with these attributes and retrieve their values, adhering to principles of immutability and encapsulation.

Attributes

- `productName` (`String`): The name of the product.
- `deliveryOptions` (`List<String>`): A list of delivery options available for the product.

Conclusion

The `Product.java` class provides a clean and efficient way to model products with their associated delivery options in a Java application. Its design adheres to best practices in object-oriented programming, making it a reusable and easy-to-understand component in the broader context of the application it belongs to.