Introduction

Oscar, the owner of the wine shop "In vino veritas", requested a web site that allows his customers to book the wine bottles of his catalog.

Oscar needs to:

- Keep track of the availability of bottle of wines in the warehouse
- Keep track of the orders of bottle of wines
- Keep track of the customer base with billing and shipping information
- Provide a good service to customers with good response times

Exercise 1

Describe in words (and/or with a diagram) the architecture would you use to help Oscar to store wine reservations and retrieve them quickly and efficiently.

Exercise 2a

Note: based on your personal preference, you can choose between completing Exercise 2a or Exercise 2b.

Implement the web page that allows the customers to book the wine bottles from the catalog. Oscar needs to know:

- the name, surname and date of birth of the customer (only adult can reserve a bottle of wine)
- the mail address of the customer
- shipping address (italian cities are many and Oscar wants to help his customers to select them easily)
- the name and quantity of bottles of wine

Using the technologies you prefer, create a web page for Oscar paying attention to appearance, usability and validation of all fields.

Expected output: a fiddle (https://jsfiddle.net/) or the compressed folder with your solution

Exercise 2b

Note: based on your personal preference, you can choose between completing Exercise 2a or Exercise 2b.

Implement a REST API used by the warehouse system to perform create/read/update/delete (CRUD) operations on the bottles of wine. Oscar wants the warehouse to be able to:

- List the type of wines available
- Add a new wine to the catalog
- Display the available bottles of a specific wine
- Update the available bottles of a specific wine
- Delete a wine from the catalog

Using Java language, implement a REST endpoint to perform this operation

All the operations can be done in-memory, you are not required to use a database.

Bonus goal: package the REST endpoint as microservice, so that it can easily scale on multiple nodes

Expected output: compressed folder with your solution

Excercise 3

Oscar reported that a portion of the product handling payment is too slow. After a debugging activity, you find out that the bad performances are due to multiple slow transactions performed by a piece of Java code.

Your goal is to make the code much faster: currently the code requires more than 2 hours to complete, it should take, at maximum, 20 minutes.

Propose an alternate version for the code below that improve performances of the processTransactions() method. A few requirements:

- You cannot modify doTransaction() and printTransactions() methods
- doTransaction() method must be called 1000 times. The sleep call simulates the slow transaction.
- printTransactions() method should be called at least every 30 seconds during the transaction processing.

Expected output: an optimized variant of the TransactionProcessor class.

```
import java.util.EnumMap;
import java.util.HashMap;
import java.util.Map;

public class TransactionProcessor {
	/** The number of transactions */
	private static final int NUM_TRANSACTIONS = 1000;
	/** The status of a transaction */
	private static enum Status { RUNNING, OK, FAILURE }
	/** The status of transactions */
	private HashMap<Integer, Status> transactionStatus = new HashMap<>();

	/**
	* Perform the complex transaction. This method must be called by the exercise and cannot be changed
	* @param input the input of the transaction
	* @return the output of the transaction
	*/
	final protected double doTransaction(double input) throws InterruptedException {
		// --- You cannot modify this method ---
		Thread.sleep(10000);
		return input * 100;
	}

	/**
	* Print the number of transactions. This method must be called by the exercise and cannot be changed
	* @param transactions an object describing the transaction status
	*/
	final protected void printTransactions(Map<?, Status> transactions) {
		// --- You cannot modify this method ---
		EnumMap<Status, Integer> counts = new EnumMap<>(Status.class);
		for (Status s : Status.values()) {
			counts.put(s, 0);
		}
		for (Status s : transactions.values()) {
			counts.put(s, counts.get(s) + 1);
		}
		System.out.printf("- %d Ok transactions, %d Running transactions, "
				+ "%d Failed transactions. Completed percentage: %s%%\n",
```

```
                counts.get(Status.OK), counts.get(Status.RUNNING),
                counts.get(Status.FAILURE),
                (counts.get(Status.OK) + counts.get(Status.FAILURE))
                * 100.0 / NUM_TRANSACTIONS);
    }

    /**
    * Process all transactions
    * @return the output of all transactions
    */
    public double processTransactions() {
        double result = 0.0;
        for (int i=0; i<NUM_TRANSACTIONS; i++) {
            try {
                transactionStatus.put(i, Status.RUNNING);
                result += doTransaction(i);
                transactionStatus.put(i, Status.OK);
                printTransactions(transactionStatus);
            } catch (InterruptedException ex) {
                System.out.println("Transaction failed");
                transactionStatus.put(i, Status.FAILURE);
            }
        }
        return result;
    }

    /**
    * Main method. Display the result and execution time.
    * @param args (not used)
    */
    public static void main(String[] args) {
        long startTime = System.currentTimeMillis();
        TransactionProcessor tp = new TransactionProcessor();
        double result = tp.processTransactions();
        System.out.printf("The result is: %f . "
                + "Elapsed time: %s seconds\n",
                result,
                (System.currentTimeMillis() - startTime) / 1000.0);
    }
}
```