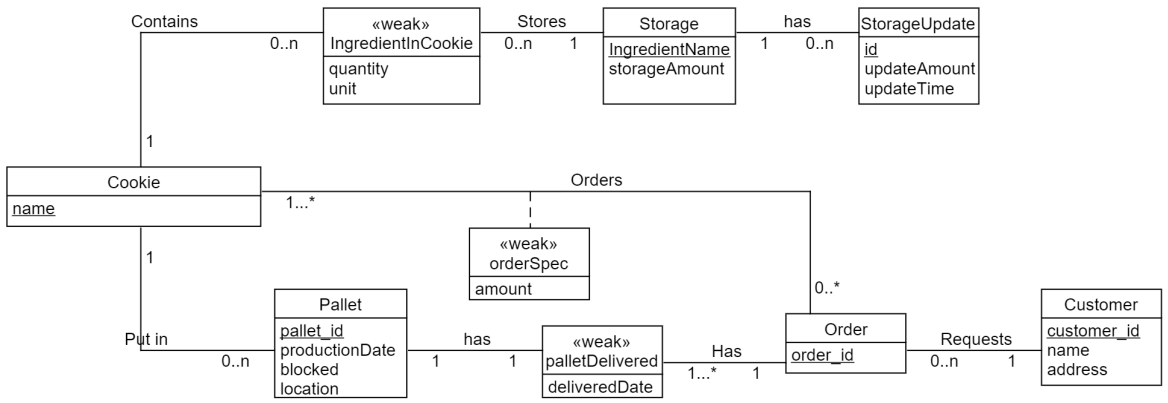
Rapport Krusty Kookies
grupp 1

Gruppmedlemmar: Martin Lysén, Jacob Persson, Joel Dahlquist och Tonny Huynh.

UML diagram



Relations

Primary keys **Foreign keys**

storages:(ingredient_name, storage_amount, storage_unit).

cookies:(name).

pallets:(Pallet_id, productionDate, blocked, location, **name**).

customers:(customer_id, name, address).

storageUpdates:(id, updateTime, updateAmount, **ingredient_name**) .

ingredientInCookies:(Quantity, Unit, **ingredient_name**, **cookie_name**).

orders:(Order_id, **customer_id**).

pallet_Delivered:(Delivered_date, **Pallet_id**, **Order_id**).

orderSpec:(Amount, **cookie_name**, **Order_id**).

Contributions

ER-diagram - Martin, Jacob, Joel, Tonny
Source-kod:

- getCustomers: Joel
- getRawMaterials: Joel
- getCookies: Jacob
- getRecipes: Jacob
- getPallets: Martin
- createPallets: Martin
- reset: Tonny + emotional support (martin)

Rapport - Martin, Jacob, Joel, Tonny

Setup

1. Koppla upp sig med LTH:s vpn eller direkt koppla upp sig till LTH:s nätverk.
LU VPN SSL

2. Starta programmet genom att köra Krusty.jar.

3. Gå in på valfri webbläsare och gå till localhost:8888.

Sourcecode

```
package krusty;

import spark.Request;
import spark.Response;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;
import java.util.Scanner;
import java.io.File;
import java.io.FileNotFoundException;
import java.nio.charset.StandardCharsets;

import static krusty.Jsonizer.toJson;

public class Database {
```

```
    /**
     * Modify it to fit your environment and then use this string when
connecting to your database!
     */
    private static final String jdbcString =
"jdbc:mysql://puccini.cs.lth.se/hbg03";
//"jdbc:mysql://localhost/krusty";

    // For use with MySQL or PostgreSQL
    private static final String jdbcUsername = "hbg03";
    private static final String jdbcPassword = "jav922za";

    //lagt till själv!!!!!!!!
    private Connection conn;
    // !!!!!!!!

    /**
     * Create the database object. Connection i performed later
     */
    public Database() {
        conn = null;
    }

    public void connect() {
        // Connect to database here
        try {
            conn = DriverManager.getConnection(jdbcString,
jdbcUsername, jdbcPassword);
        } catch (SQLException e) {
            System.err.println(e);
            e.printStackTrace();
        }
    }

    // TODO: Implement and change output in all methods below!
    public String getCustomers(Request req, Response res) {
        String selectCustomers = "select name, address\n" + "FROM
customers;";
        try (
                PreparedStatement ps =
conn.prepareStatement(selectCustomers);) {
            ResultSet resultSet = ps.executeQuery();
            String json = Jsonizer.toJson(resultSet, "customers");
```

```
            return json;
        } catch (SQLException e) {
            e.printStackTrace();
            return "{\"customers\":[],\"error\":\"Database error
occurred.\"}";
        }

    }

    public String getRawMaterials(Request req, Response res) {
        String selectRawMaterials =
    "SELECT ingredient_name AS name, " +
    "storage_amount AS amount, " +
    "storage_unit AS unit " +
    "FROM storages " +
    "ORDER BY name;";

        try (
                PreparedStatement ps =
conn.prepareStatement(selectRawMaterials);) {
            ResultSet resultSet = ps.executeQuery();
            String json = Jsonizer.toJson(resultSet, "raw-materials");
            return json;
        } catch (SQLException e) {
            // Log error and return an error message or empty JSON
            e.printStackTrace();
            return "{\"\"raw-materials\":[],\"error\":\"Database error
occurred.\"}";
        }
    }

    public String getCookies(Request req, Response res) {
        String query = "SELECT name\n" + "FROM cookies\n" + "ORDER BY
name";

        try (PreparedStatement ps = conn.prepareStatement(query)) {
            ResultSet rs = ps.executeQuery();
            String result = Jsonizer.toJson(rs, "cookies");
            return result;
        }

        catch (SQLException e) {
            e.printStackTrace();
```

```
            return "{\"cookies\":[],\"error\":\"Database error
occurred.\"}";
        }

    }

    public String getRecipes(Request req, Response res) {
        String query = "SELECT *\n" + "FROM ingredientInCookies\n" +
"Order by cookie_name;";

        try (PreparedStatement ps = conn.prepareStatement(query)) {
            ResultSet rs = ps.executeQuery();
            String result = Jsonizer.toJson(rs, "recipes");
            return result;
        }

        catch (SQLException e) {
            e.printStackTrace();
            return "{\"recipes\":[],\"error\":\"Database error
occurred.\"}";
        }

    }

    public String getPallets(Request req, Response res) {
        // Initial SQL query with WHERE 1=1
        String sql = "SELECT p.Pallet_id AS id, p.name AS cookie,
p.productionDate AS production_date, " +
                "IFNULL(c.name, 'null') AS customer, IF(p.blocked,
'yes', 'no') AS blocked " +
                "FROM pallets p " +
                "LEFT JOIN pallet_Delivered pd ON p.Pallet_id =
pd.Pallet_id " +
                "LEFT JOIN orders o ON pd.Order_id = o.Order_id " +
                "LEFT JOIN customers c ON o.customer_id = c.customer_id
" +
                "WHERE 1=1 "; // This condition always evaluates to
true

        // ArrayList to hold parameters for prepared statement
        ArrayList<String> values = new ArrayList<>();
```

```
        // Handling the 'from' query parameter (date produced on or
after)
        String fromParam = req.queryParams("from");
        if (fromParam != null) {
            sql += " AND p.productionDate >= ?";
            values.add(fromParam);
        }

        // Handling the 'to' query parameter (date produced on or
before)
        String toParam = req.queryParams("to");
        if (toParam != null) {
            sql += " AND p.productionDate <= ?";
            values.add(toParam);
        }

        // Handling the 'cookie' query parameter (filter by cookie
name)
        String cookieParam = req.queryParams("cookie");
        if (cookieParam != null) {
            sql += " AND p.name = ?";
            values.add(cookieParam);
        }

        // Handling 'blocked' parameter (filter by blocked status)
        String blockedParam = req.queryParams("blocked");
        if (blockedParam != null) {
            String blockedValue = blockedParam.equalsIgnoreCase("yes")
? "1" : "0";
            sql += " AND p.blocked = ?";
            values.add(blockedValue);
        }

        // Add ORDER BY clause
        sql += " ORDER BY p.productionDate DESC;";

        try (PreparedStatement stmt = conn.prepareStatement(sql)) {
            // Set the values for the prepared statement
            for (int i = 0; i < values.size(); i++) {
                stmt.setString(i + 1, values.get(i));
            }

            // Execute the query and handle the results
```

```
            try (ResultSet rs = stmt.executeQuery()) {
                // Convert ResultSet to JSON
                String json = Jsonizer.toJson(rs, "pallets");
                return json;
            }

        } catch (SQLException e) {
            // Log error and return an error message or empty JSON
            e.printStackTrace();
            return "{\"pallets\":[],\"error\":\"Database error
occurred.\"}";
        }
    }

    /**
     * @param req
     * @param res
     * @return
     * @throws SQLException
     */
public String reset(Request req, Response res) throws SQLException,
FileNotFoundException {
    String disableForeignKeyChecks = "SET FOREIGN_KEY_CHECKS=0;";
    String enableForeignKeyChecks = "SET FOREIGN_KEY_CHECKS=1;";

    try (Statement stmt = conn.createStatement()) {
        conn.setAutoCommit(false);
        stmt.execute(disableForeignKeyChecks); // Disable foreign key
checks

        // Each table, ready to be truncated
        String[] tablesToClear = {
            "pallet_Delivered",
            "cookies",
            "pallets",
            "storages",
            "customers",
            "storageUpdates",
            "ingredientInCookies",
            "orders",
            "orderSpec"
        };
        //Loop for TRUNCATING each table
```

```
        for (String table : tablesToClear) {
            String clearTableQuery = "TRUNCATE TABLE " + table;
            stmt.executeUpdate(clearTableQuery);
        }

        // Read SQL commands from file
        File sqlFile = new
File("krusty-skeleton\\src\\main\\resources\\public\\initial-data.sql")
;
        try (Scanner scanner = new
Scanner(sqlFile,StandardCharsets.UTF_8.name())) {
            StringBuilder sql = new StringBuilder();
            while (scanner.hasNextLine()) {
                String line = scanner.nextLine().trim();
                if (!line.isEmpty()) {
                    sql.append(line);
                    if (line.endsWith(";")) {
                        // Execute SQL
                        stmt.executeUpdate(sql.toString());
                        // Clear StringBuilder for the next command
                        sql.setLength(0);
                    }
                }
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
            return "{\"status\": \"error\", \"message\": \"SQL file not
found\"}";
        }

        stmt.execute(enableForeignKeyChecks); // Enable foreign key
checks
        conn.commit();

    } catch (SQLException e) {
        conn.rollback();
        e.printStackTrace();
        return "{\"status\": \"error\", \"message\": \"Failed to reset
database\"}";
    } finally {
        conn.setAutoCommit(true);
    }
```

```
    return "{\"status\": \"ok\", \"message\": \"Database reset
successful\"}";
}


public String createPallet(Request req, Response res) {
    String cookieName = req.queryParams("cookie");
    if (cookieName == null || cookieName.isEmpty()) {
        res.status(400); // Bad Request
        return "{\"status\":\"error\",\"message\":\"Missing or empty
'cookie' parameter\"}";
    }

    String checkCookieSql = "SELECT COUNT(*) FROM cookies WHERE name =
?";
    String insertPalletSql = "INSERT INTO pallets (productionDate,
blocked, location, name) VALUES (NOW(), false, ?, ?)";
    String selectIngredientsSql = "SELECT ingredient_name, quantity
FROM ingredientInCookies WHERE cookie_name = ?";
    String updateStoragesSql = "UPDATE storages SET storage_amount =
storage_amount - ? WHERE ingredient_name = ?";

    try (
            PreparedStatement checkCookieStmt =
conn.prepareStatement(checkCookieSql);
            PreparedStatement insertPalletStmt =
conn.prepareStatement(insertPalletSql,
                    Statement.RETURN_GENERATED_KEYS);
            PreparedStatement selectIngredientsStmt =
conn.prepareStatement(selectIngredientsSql);
            PreparedStatement updateStoragesStmt =
conn.prepareStatement(updateStoragesSql)) {

        conn.setAutoCommit(false); // Start transaction

        // Check if the specified cookie exists
        checkCookieStmt.setString(1, cookieName);
        try (ResultSet cookieResult = checkCookieStmt.executeQuery()) {
            if (cookieResult.next() && cookieResult.getInt(1) > 0) {
                // Get ingredients for the specified cookie
                selectIngredientsStmt.setString(1, cookieName);
                try (ResultSet ingredientsResult =
selectIngredientsStmt.executeQuery()) {
```

```
                    // Create the pallet
                    int randomLocation = (int) (Math.random() * 99) +
1; // Random location between 1 and 99
                    insertPalletStmt.setInt(1, randomLocation);
                    insertPalletStmt.setString(2, cookieName);
                    int affectedRows =
insertPalletStmt.executeUpdate();
                    if (affectedRows > 0) {
                        // Retrieve and process generated pallet ID
                        try (ResultSet generatedKeys =
insertPalletStmt.getGeneratedKeys()) {
                            if (generatedKeys.next()) {
                                long palletId =
generatedKeys.getLong(1);

                                // Deduct ingredients from storages
                                while (ingredientsResult.next()) {
                                    String ingredientName =
ingredientsResult.getString("ingredient_name");
                                    int quantity =
ingredientsResult.getInt("quantity")*54;
                                    updateStoragesStmt.setInt(1,
quantity);
                                    updateStoragesStmt.setString(2,
ingredientName);
                                    updateStoragesStmt.addBatch();
                                }

                                // Execute batch update for storages
                                updateStoragesStmt.executeBatch();

                                conn.commit(); // Commit transaction
                                res.status(201); // Created
                                return
String.format("{\"status\":\"ok\",\"id\":%d}", palletId);
                            }
                        }
                    }
                }
            }
        }
        // If execution reaches here, handle errors
        conn.rollback(); // Rollback transaction
```

```
        res.status(500); // Internal Server Error
        return "{\"status\":\"error\",\"message\":\"Failed to create
pallet\"}";
    } catch (SQLException e) {
        // Log the exception
        e.printStackTrace();
        res.status(500); // Internal Server Error
        return "{\"status\":\"error\",\"message\":\"An unexpected error
occurred\"}";
    }
}
}
```