IMSE Milestone 2

Danara Abushinova 51823968

Juni 2023

Running the project

In order to run the project, open the command line and navigate to the project path with "cd <path to the root directory>", then run the command "docker compose build", after a successful execution run "docker compose up". You can now and open the link `http://localhost:8089` for HTTP in your browser and observe the website.

You can login as an administrator to the system with the username: "admin" and the password: "admin". Alternatively you can login as a customer using the username: "user" and the password: "user"(if you are logging in as a user, please open http://localhost:8089/product-list directly after the logging, as the implementation of the user is not complete; you can now surf on the website as a user).

Tech-Stack

The purpose of the project was to implement a website with an integrated database using two different methods and compare them. The first one is RDBMS, for which I used PostgreSQL database and the second one is the NoSQL database, for which I have decided to choose MongoDB.

Here are the technical characteristics of the project:

Backend:

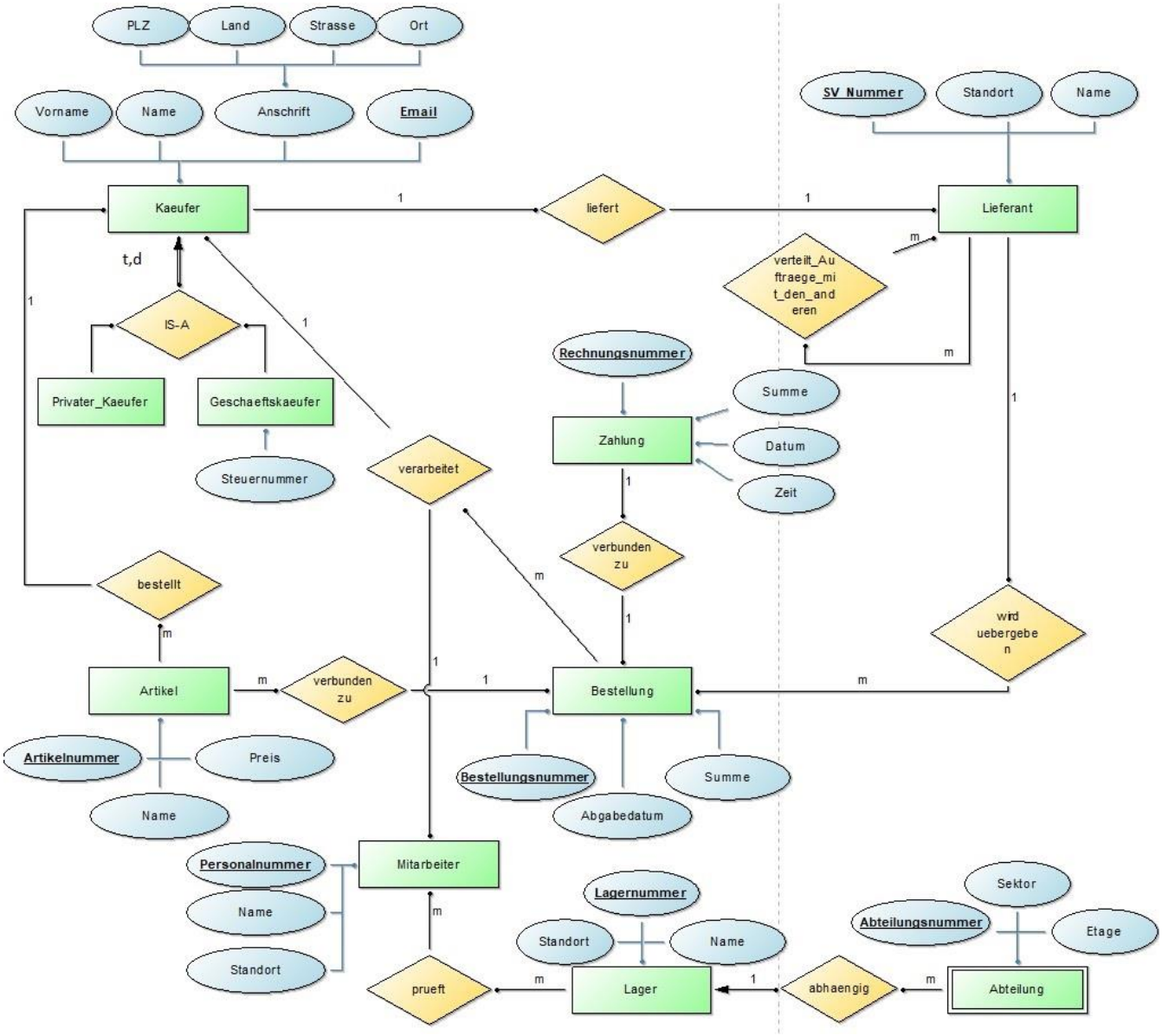
- Framework: Spring Boot for building the backend, which simplifies development and integrates smoothly with databases.
- Database: PostgreSQL for storing structured data effectively.
- NoSQL Database: MongoDB for handling flexible and unstructured data.
- Language: Java (version 11) for backend programming.

To create sample data, I used the Java Faker library(https://github.com/DiUS/java-faker), which makes it easier to generate realistic and representative test data.

Frontend:

-Framework: Vaadin for creating the user interface, which allows to build interactive components using Java and HTML, as Vaadin supports the Object Oriented Model.

RDBMS model



I have mapped the logical database design to the physical using MySQL(see below). The M-N relations from the logical design have been changed to separate tables, which are connected with the 1-M relations in the physical one.

```
CREATE TABLE IF NOT EXISTS public.articles
(
    article_id integer NOT NULL DEFAULT nextval('articles_article_id_seq'::regclass),
    article_name character varying(255) COLLATE pg_catalog."default" NOT NULL,
    article_price double precision NOT NULL,
    article_quantity integer NOT NULL,
    customer_id integer,
    order_id integer,
    CONSTRAINT articles_pkey PRIMARY KEY (article_id),
    CONSTRAINT fk4jl2hthd3dvt7qujdaoe4g1qo FOREIGN KEY (order_id)
        REFERENCES public.orders (order_id) MATCH SIMPLE
        ON UPDATE NO ACTION
```

```
        ON DELETE NO ACTION,
    CONSTRAINT fkr1ka7d5y92sks6djphoh59xct FOREIGN KEY (customer_id)
        REFERENCES public.customers (customer_id) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
)
CREATE TABLE IF NOT EXISTS public.customers
(
    customer_id integer NOT NULL DEFAULT nextval('customers_customer_id_seq'::regclass),
    address character varying(255) COLLATE pg_catalog."default" NOT NULL,
    email character varying(255) COLLATE pg_catalog."default" NOT NULL,
    lastname character varying(255) COLLATE pg_catalog."default" NOT NULL,
    password character varying(255) COLLATE pg_catalog."default" NOT NULL,
    surname character varying(255) COLLATE pg_catalog."default" NOT NULL,
    vendor_name character varying(255) COLLATE pg_catalog."default",
    CONSTRAINT customers_pkey PRIMARY KEY (customer_id),
    CONSTRAINT fkk3oply38n67clbttu8954wuoq FOREIGN KEY (vendor_name)
        REFERENCES public.vendors (vendor_name) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
)
CREATE TABLE IF NOT EXISTS public.departments
(
    department_id integer NOT NULL DEFAULT nextval('departments_department_id_seq'::regclass),
    etage integer NOT NULL,
    sector character varying(255) COLLATE pg_catalog."default" NOT NULL,
    warehouse_warehouse_id integer,
    CONSTRAINT departments_pkey PRIMARY KEY (department_id),
    CONSTRAINT fkrmxciy7dwdmu585pqnkqypl46 FOREIGN KEY (warehouse_warehouse_id)
        REFERENCES public.warehouses (warehouse_id) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
)
CREATE TABLE IF NOT EXISTS public.employees
(
    personal_id integer NOT NULL DEFAULT nextval('employees_personal_id_seq'::regclass),
    location character varying(255) COLLATE pg_catalog."default" NOT NULL,
    name character varying(255) COLLATE pg_catalog."default" NOT NULL,
    CONSTRAINT employees_pkey PRIMARY KEY (personal_id)
)
CREATE TABLE IF NOT EXISTS public.orders
(
    order_id integer NOT NULL DEFAULT nextval('orders_order_id_seq'::regclass),
    fulfilled boolean NOT NULL,
    order_date character varying(255) COLLATE pg_catalog."default" NOT NULL,
    total_amount double precision NOT NULL,
    customer_id integer,
    employee_id integer,
    payment_id integer,
    vendor_id character varying(255) COLLATE pg_catalog."default",
    CONSTRAINT orders_pkey PRIMARY KEY (order_id),
    CONSTRAINT fk8aol9f99s97mtyhij0tvfj41f FOREIGN KEY (payment_id)
        REFERENCES public.payments (payment_id) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION,
    CONSTRAINT fkb1443sk0bxprtkqree3o2qk90 FOREIGN KEY (vendor_id)
```

```
        REFERENCES public.vendors (vendor_name) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION,
    CONSTRAINT fkfhl8bv0xn3sj33q2f3scf1bq6 FOREIGN KEY (employee_id)
        REFERENCES public.employees (personal_id) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION,
    CONSTRAINT fkpxtb8awmi0dk6smoh2vp1litg FOREIGN KEY (customer_id)
        REFERENCES public.customers (customer_id) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
)
CREATE TABLE IF NOT EXISTS public.payments
(
    payment_id integer NOT NULL DEFAULT nextval('payments_payment_id_seq'::regclass),
    payment_date character varying(255) COLLATE pg_catalog."default" NOT NULL,
    price double precision NOT NULL,
    "time" character varying(255) COLLATE pg_catalog."default" NOT NULL,
    CONSTRAINT payments_pkey PRIMARY KEY (payment_id)
)
CREATE TABLE IF NOT EXISTS public.vendor_relationship
(
    vendor_id character varying(255) COLLATE pg_catalog."default" NOT NULL,
    related_vendor_id character varying(255) COLLATE pg_catalog."default" NOT NULL,
    CONSTRAINT fke86qi25ut8e0uul4lr25hd4ex FOREIGN KEY (vendor_id)
        REFERENCES public.vendors (vendor_name) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION,
    CONSTRAINT fkkdoqis1kjcj1jvc3eqwftt7yh FOREIGN KEY (related_vendor_id)
        REFERENCES public.vendors (vendor_name) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
)
CREATE TABLE IF NOT EXISTS public.vendors
(
    vendor_name character varying(255) COLLATE pg_catalog."default" NOT NULL,
    vendor_address character varying(255) COLLATE pg_catalog."default" NOT NULL,
    vendor_svnr character varying(255) COLLATE pg_catalog."default" NOT NULL,
    CONSTRAINT vendors_pkey PRIMARY KEY (vendor_name)
)
CREATE TABLE IF NOT EXISTS public.warehouse_employee
(
    warehouse_id integer NOT NULL,
    employee_id integer NOT NULL,
    CONSTRAINT fkdnksnomecyoteavkph3pq3md0 FOREIGN KEY (employee_id)
        REFERENCES public.employees (personal_id) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION,
    CONSTRAINT fkjce8naeoxgxhjq174be824et FOREIGN KEY (warehouse_id)
        REFERENCES public.warehouses (warehouse_id) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
)
CREATE TABLE IF NOT EXISTS public.vendor_relationship
(
    vendor_id character varying(255) COLLATE pg_catalog."default" NOT NULL,
```

```
    related_vendor_id character varying(255) COLLATE pg_catalog."default" NOT NULL,
    CONSTRAINT fke86qi25ut8e0uul4lr25hd4ex FOREIGN KEY (vendor_id)
        REFERENCES public.vendors (vendor_name) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION,
    CONSTRAINT fkkdoqis1kjcj1jvc3eqwftt7yh FOREIGN KEY (related_vendor_id)
        REFERENCES public.vendors (vendor_name) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
)
```

Data Import

The database filling is possible from the website. You can find the "**Fill database**" button in the top left corner.
In order to perform and display the main use cases, you first need to fill the database, wait until it proceeded successfully and then migrate the data by pressing the corresponding button("**Migrate database**"). After finishing the migration you can perform the main use case.

Implementation of a Web system

I have implemented a main use case and a corresponding report to it:

-Main Use-Case: Order Fulfillment
-Report: Top 5 Artikel, die am häufigsten ausverkauft sind, wenn ein Käufer sie kaufen möchte.

NoSQL Design

```
{
   "customers": [
      {
         "_id": ObjectId,
         "address": String,
         "email": String,
         "lastname": String,
         "password": String,
         "surname": String,
         "vendor_name": String,
         "orders": [
            {
               "_id": ObjectId,
               "fulfilled": Boolean,
               "order_date": Date,
               "total_amount": Number,
               "employee_id": ObjectId,
               "payment_id": ObjectId,
               "vendor_id": String,
               "articles": [
                  {
```

```
                    "_id": ObjectId,
                    "article_name": String,
                    "article_price": Number,
                    "article_quantity": Number
                }
            ]
        }
    ]
  }
  ],
  "departments": [
    {
      "_id": ObjectId,
      "etage": Number,
      "sector": String,
      "warehouse_id": ObjectId
    }
  ],
  "employees": [
    {
      "_id": ObjectId,
      "location": String,
      "name": String,
      "warehouses": [
        ObjectId
      ]
    }
  ],
  "payments": [
    {
      "_id": ObjectId,
      "payment_date": Date,
      "price": Number,
      "time": Date
    }
  ],
  "vendors": [
    {
      "_id": ObjectId,
      "vendor_name": String,
      "vendor_address": String,
      "vendor_svnr": String,
      "related_vendors": [
        ObjectId
      ]
    }
  ],
  "warehouses": [
    {
      "_id": ObjectId,
      "employees": [
        ObjectId
      ]
    }
  ]
}
```

This schema design embeds the orders and articles within the customers collection, and the employees within the warehouses collection. This is a common practice in NoSQL databases to reduce the need for joins, which are not as efficient in NoSQL databases as they are in SQL databases. The related_vendors in the vendors collection and the employees in the warehouses collection are represented as arrays of ObjectIds, which are references to other documents.

NoSQL Indexing

-Index on **Users.customer_id**: This will improve performance when querying specific users based on their customer_id.

-Index on **Articles.article_id**: It would allow faster lookups when searching for specific articles based on their article_id.

-Index on **Orders.order_id**: If frequent queries are made to find specific orders, an index on order_id could enhance performance.

-Index on **Employees.personal_id**: Similar to the above, it will improve performance when querying specific employees based on their personal_id.

-Compound Index on **Orders.customer_id** and **Orders.order_date**: If there are frequent queries to get the orders of a specific customer within a date range, this index could speed up these queries.

Data Migration

The migration of the data is quite similar to the data import - once the button “Migrate data” in the frontend is clicked, the data will be migrated **from the PostgreSQL to NoSQL** and the URLs will be updated in order to use the NoSQL paths for the further interactions in the frontend.

Comparison of use-cases and reports

Main Use-Case*:* Order Fulfillment

-PostgreSQL:

```
SELECT a.articleName, COUNT(*) AS totalOutOfStock
FROM Order o
JOIN Article a ON o.orderId = a.orderId
WHERE a.articleQuantity = 0
GROUP BY a.articleName
ORDER BY totalOutOfStock DESC
LIMIT 5;
```

Data is queried by joining the Order and Article tables based on the orderId column. Only articles with a quantity of 0 (articleQuantity = 0) are considered. The count of out-of-stock occurrences per article is calculated. Results are grouped by article name and sorted in descending order. Only the top 5 articles with the highest count are returned.

-NoSQL:

```
@Aggregation(pipeline = {
        "{ $match: { _class: { $eq: \"com.example.application.data.nosql.entities.OrderNoSQL\" }, }
}",
        "{ $unwind: \"$articles\"}",
```

```
        "{ $match: { \"articles.articleQuantity\": { $eq: 0 } } }",
        "{ $group: { _id: \"$articles.articleName\", totalOutOfStock: { $sum: 1 } } }",
        "{ $sort: {\"totalOutOfStock\": -1}}",
        "{ $limit: 5 }"
  })
```

Data is aggregated on the OrderNoSQL collection. The articles field is unwound to access individual articles. Only articles with a quantity of 0 (articleQuantity = 0) are considered. The count of out-of-stock occurrences per article is calculated. Results are grouped by article name and sorted in descending order. Only the top 5 articles with the highest count are returned.