

Link to repository: <https://github.com/SoftwareDevelopmentSem1/DB-Exam>

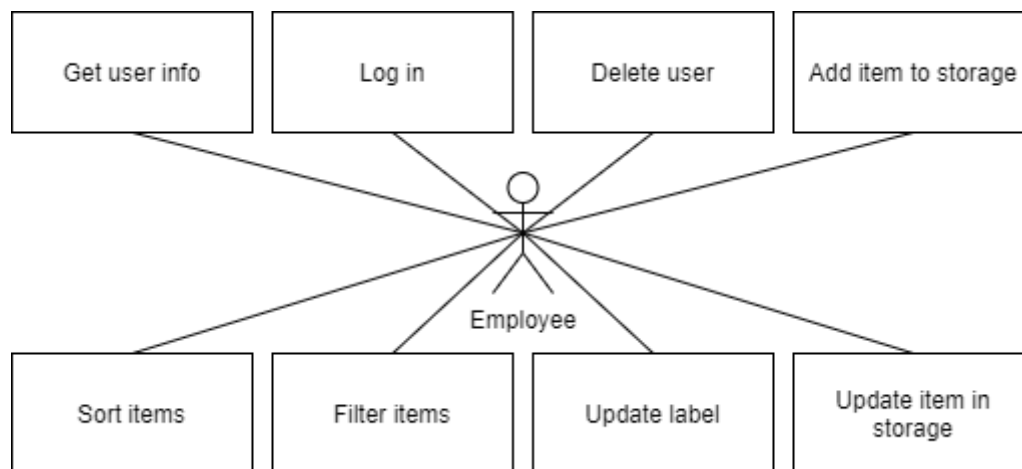
The Business

The business we have chosen for our project is an online retail store à la nemlig.com.

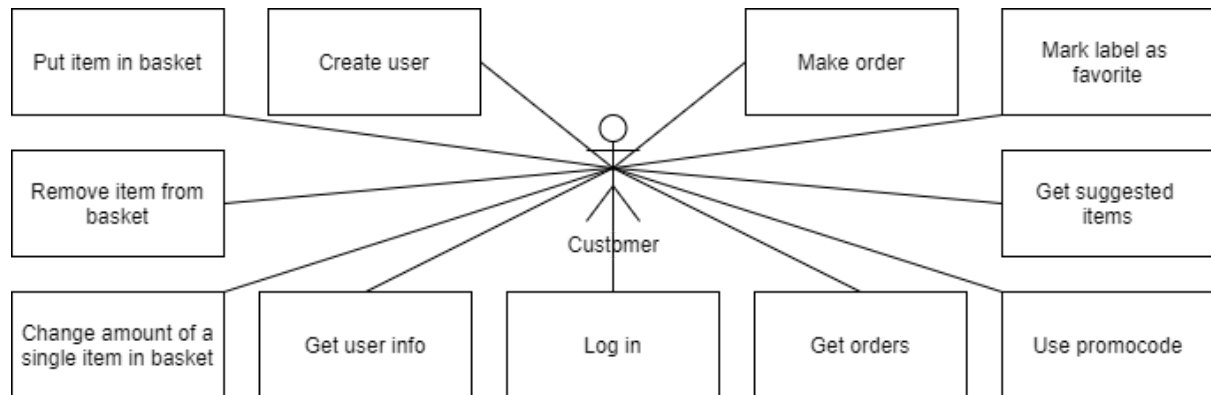
A customer should be able to create an account, login and sign out. The customer should be able to browse all the different goods and filter by price, brand, and type, in addition to being able to sort based on price. When the customer has decided upon a good they should be able to add the desired amount of that good to their basket. When the customer is done browsing goods, they should be able to order and pay for the content of their basket. If the customer has not interacted with their basket for 30 minutes the basket is cleared. The customer should be able to get a list of their previous orders. The customer should be able to mark a brand or type as favorite.

An employee should be able to get a list of the inventory, complete with the amount of stock for each item. The employee should be able to update the stock, price and tags of an item when needed.

Use Cases



As an employee we want to be able to manage different activities on the site, this includes updating information about an item, adding new items, changing the stock of a set item if new items have arrived at the warehouse. We also wanna be able to see all previous orders, and update their state, as they have been sent out.



A customer should have the options to create a profile, which is stored in the postgres database and see items. Access to the website is granted when the customer logs in, and starts a session. When a session is started the customer should have access to features like mark favorite labels, purchase an order and get suggested items based on favorites. The order is based on the items placed in the basket, where the amount can be modified, this order can be combined with a promode, which gives some amount of discount. The customer also should have access to the order history and profile info.

Requirements

Functional

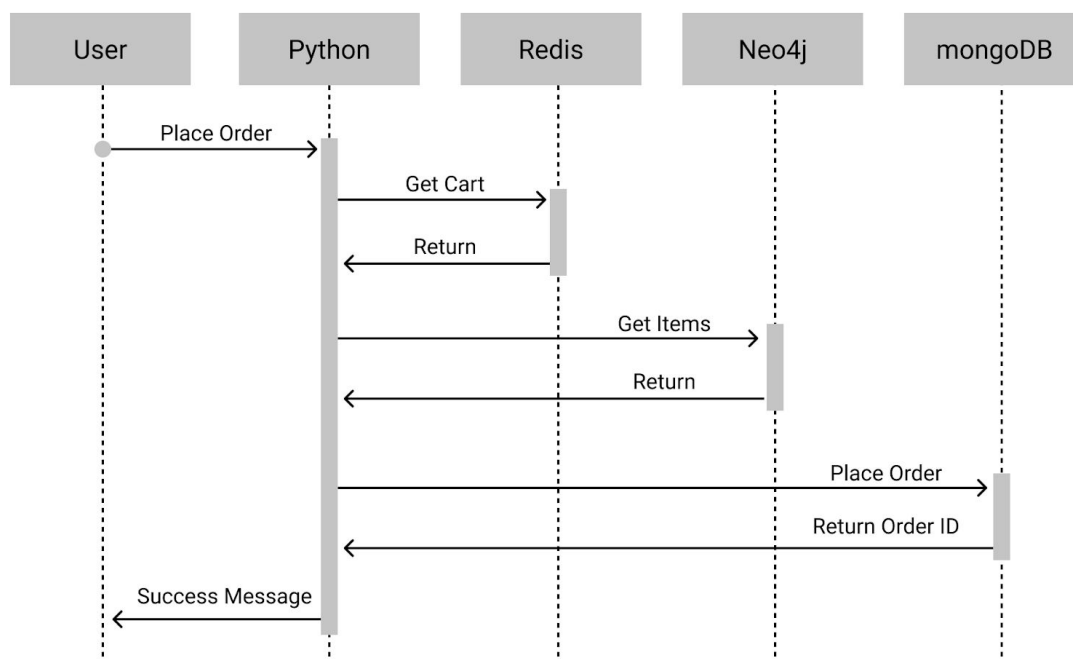
Customer	Employee
Create an account	Get a list of the inventory
Login to their account	Update inventory stock
Sign out of their account	Update inventory price
Browse the inventory	Update inventory tags
Sort inventory by price	Add new promo code
Filter inventory by price	Get a list of orders
Filter inventory by brand	
Filter inventory by type	
Add goods to their basket	
Submit a promo code	
Checkout their basket	

Get a list of their previous orders	
Mark brand as favorite	
Mark type as favorite	
Mark item as favorite	

Non-functional

- A basket should be cleared if it hasn't been interacted with for 30 minutes
- The employee actions should be locked for the customers

Implementation Scenarios



When a user places an order, the cart is requested from the redis database. This cart is forwarded to Neo4j where the items and stock are handled and returned with a price tag. The items from Neo4j are forwarded to mongoDB. MongoDB stores the order and handles the price calculation based on the item's amount and prices. After the order has been stored a success response is sent to the customer.

CAP theorem

C - consistency - different nodes respond with the same data to the same request

A - availability - the system responds to a request, even if the system isn't working or its data is outdated

P - partition tolerance - the system detects, remains operational during, and heals a partition caused by the failure of one or many nodes

ACID principle

A - atomicity - either all of the operations in a transaction succeed, or none of them do

C - consistency - the database enforces rules about its fields and the relationships between fields

I - isolation - the extent to which rows that a transaction is affecting can be affected by other transactions

D - durability - the database writes its data to a permanent medium (hard drive) so that data is not lost during a power failure or other system failure

Choice of databases

Redis

Redis is an in-memory database, structured using key-value pairs.

We decided to use Redis Database as a way to store a users cart, the reason redis is especially good for this is because it lives in the computer's ram, this makes it incredibly fast. It is important to note that for this same reason the Redis isn't a very persistent database. This could in many cases be a problem, but since this data isn't for example a user's credentials etc. even if someone gets lost, it's not the end of the world.

We could have stored an accounts cart in a session on the server, however by storing it in a database it makes it possible for the user to in essence share the same cart on multiple devices as long as they are logged into the same account, this makes it so an entire family can use the same cart or a person can access their cart from multiple devices.

Redis supports clustering, this means we can spread the data out into different nodes.

This could be as follows:

- Node A contains hash slots from 0 to 5500.
- Node B contains hash slots from 5501 to 11000.
- Node C contains hash slots from 11001 to 16383.

If a node fails, we use a backup node, this is implemented using a master-slave pattern, this means that every node has a "slave" that is a copy of it, so if the master node fails the slave takes over and serves those specific slots.

PostgreSQL

For storing user data we chose to use postgres, the relational database enables us to not have duplicate data for region

PostgreSQL is an open-source relational database management system emphasizing extensibility and SQL compliance.

PostgreSQL is an advanced, enterprise class relational database that supports both SQL (relational) and JSON (non-relational) querying. It is a highly stable database management system with high levels of resilience, integrity, and correctness.

Benefits of using PostgreSQL include: its rich features and extension, its reliability and standards compliance, and its open source license.

PostgreSQL possesses features like Multi-Version Concurrency Control (MVCC), point in time recovery, granular access controls, tablespaces, asynchronous replication, nested transactions, a refined query optimizer, and write ahead logging. It supports international character sets, multi-byte character encodings, Unicode, and is locale-aware for sorting, case-sensitivity, and formatting. PostgreSQL is highly scalable both in the quantity of data it can manage and in the number of concurrent users it can accommodate.

PostgreSQL's write ahead logging makes it a highly fault tolerant database. Its large base of open source contributors lends it to a built-in community support network. PostgreSQL is ACID compliant, and has full support for foreign keys, joins, views, triggers, and stored procedures, in many different languages. It includes most data types, including INTEGER, NUMERIC, BOOLEAN, CHAR, VARCHAR, DATE, INTERVAL, and TIMESTAMP. It also supports storage of binary large objects, including pictures, sounds, or video.

PostgreSQL source code is available under an open source license, granting you the freedom to use, modify, and implement it as you see fit, at no charge. PostgreSQL's dedicated community of contributors and enthusiasts regularly find bugs and fixes, contributing to the overall security of the database system.

Neo4j

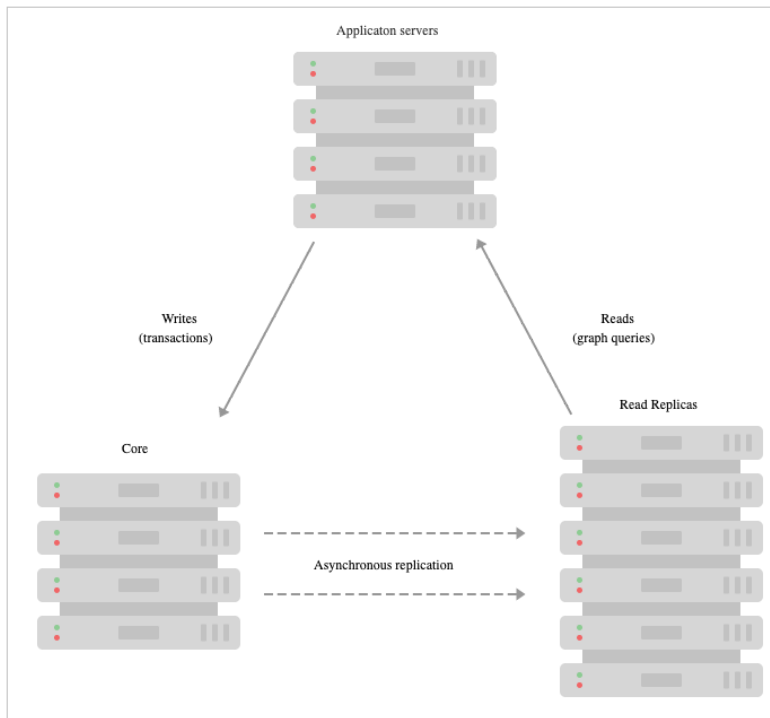
We selected this database to contain the stock of our goods, with nodes called Item, Brands and Labels. The main reason was to create a connection between the relations, so an item easily could reference another similar item. This is obtained because every Item node has a relation to a Brand node, and one or more Label nodes. It's easy to get similar Item nodes from a specific item, by looking at the items connected to the same labels.

With a relational database this had been obtained with a lot of joins which has a low performance.

Neo4j uses transactions explicitly on every query using ACID. This means a query can either be fully successful or rolled back if an issue should occur. The Neo4j python driver gives the possibility to roll the transaction back, if there should appear an issue. We are using this

feature when we are adding an item to the database. If the item already exists we will roll the transaction back, instead of overriding it.

Neo4j supports clustering, called Causal Clustering, where multiple read replicas of the core server can be spread to gain a more comprehensive access. In our case, there is a lot of reading due to all the get queries on this database which makes this type of cluster a great load optimizer. Writing occurs less often, and can therefore be called directly on the core server.



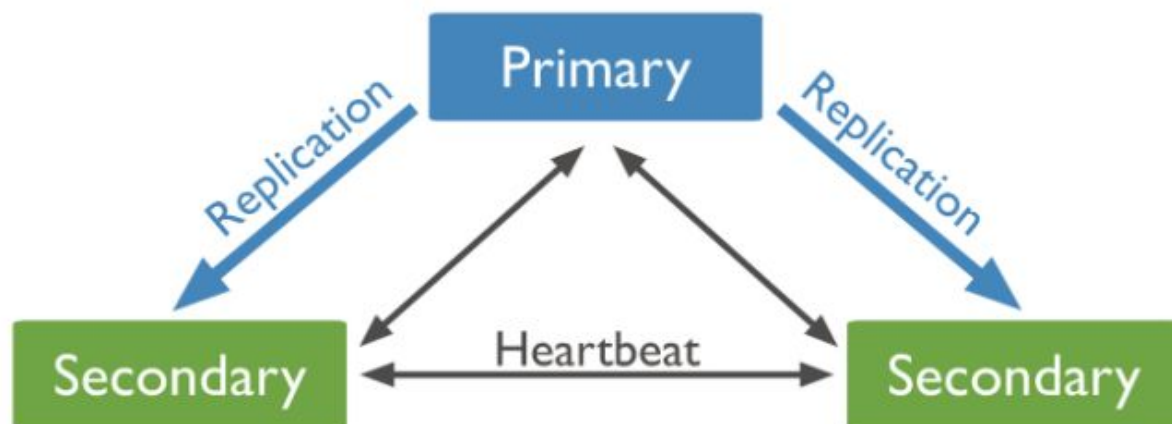
<https://neo4j.com/docs/operations-manual/current/clustering/introduction/>

MongoDB

We chose to use MongoDB to store the orders and the promo codes. We chose Mongo for the orders because we would be able to store the orders as documents with the items embedded in a list.

MongoDB uses the CAP theorem and is CP, Consistency and Partition Tolerance, it resolves network partitions by maintaining consistency while compromising on availability. Starting from MongoDB 4.0 it has added support for multi-document transactions, making it the first database to combine the speed, flexibility, and power of the document model with the data integrity ACID guarantees.

MongoDB is a single-master system, each replica set can have only one primary node that receives all the write operations. All the other nodes in a replica set are secondary nodes, that replicate the operation log of the primary node and apply it to their own data set.



When the primary node becomes unavailable, the secondary node with the most recent operation log will be elected as the primary node. Once all the other secondary nodes catch up with the new master, the cluster becomes available again.

Database Structure

Redis

The structure of the cart looks like this:

```
email: {  
  item_id:"amount|price|name",  
  item_id:"amount|price|name",  
  item_id:"amount|price|name"  
}
```

Email is the cart's 'key' and is used for access to a specific users cart. The cart itself is the value, and is in itself another hashtable. In the cart we store information about the selected item, amount, price and name. We keep this values in one string separated by a pipe (|)

PostgreSQL

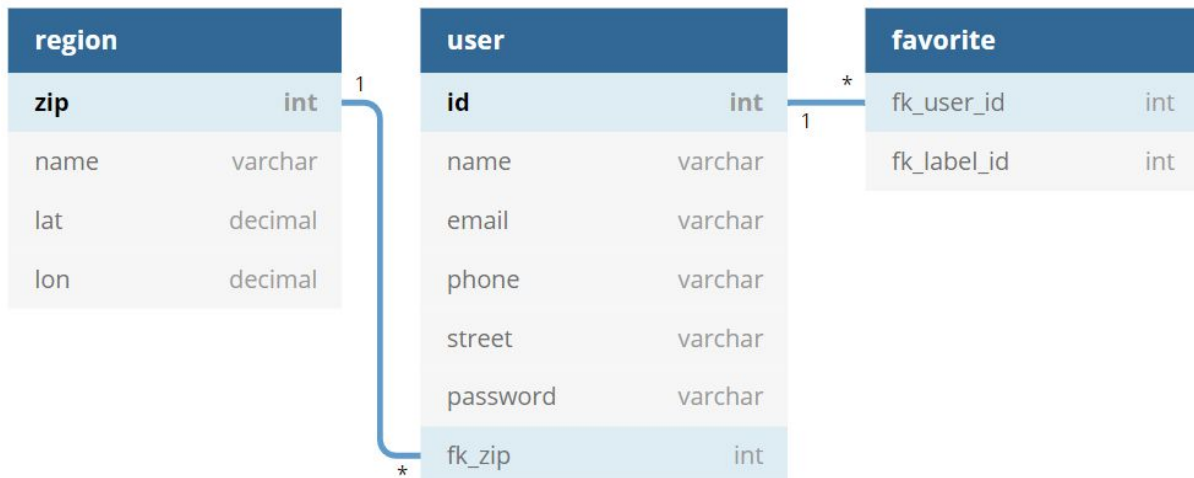
Normalization

UN: Primary keys (no duplicate tuples), no repeating groups

1NF: Atomic columns (cells have a single value)

2NF: No partial dependencies (values depend on the whole of every Candidate key)

3NF: No transitive dependencies (values depend only on Candidate keys)



We moved regions to a separate table to not break 3rd normal form, because name, lat and lon is dependent on zip.

We moved favorites out of the user table as having a list of values in a single cell would break 1st normal form, we did this by creating a 1-Many relation between user and favorite.

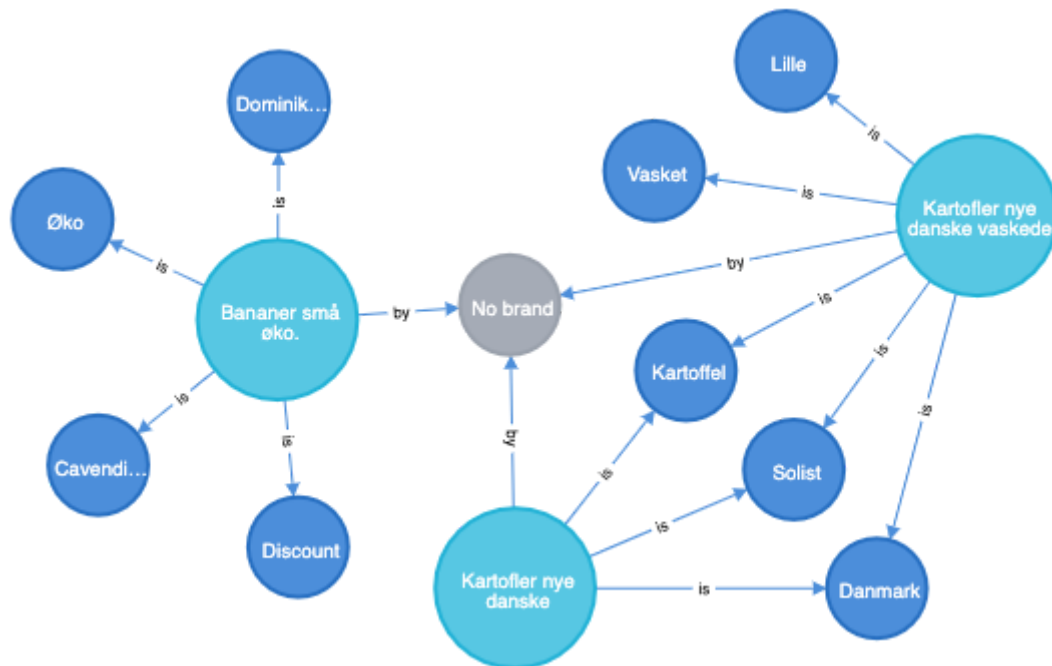
We decided to denormalize 'street', because it fits our use case better.

We have a query where we need to return a joined table of favorite, user and region, for this we decided to make a view to make the select query.

We decided to make all of our queries as stored procedures or functions to keep a layer of separation between the database queries and the backend code from our flask application.

Neo4j

This is the structure of our nodes.



Neo4j is a graph database, each node is represented as a list. Each node has pointers to other nodes that it is related to, this could be an item. Having an edge that points to each of its associated labels, as well as an edge to the brand that produces it.


Because it's a graph it makes relations between items in the database very easy and fast to access. We could for example find all items that have the same category as a given item by first following an edge to a brand. In our case above we could be going from "Kartofler nye danske vaskede" to "No brand", we can then from here see that all edges going out from this node are all the items of this specific label.

MongoDB

We decided to store the orders as documents with an auto-generated `_id`, the date the order was placed, the id of the user that placed the order, the discount in percent, and an embedded list of items, where each item has an id, a name, a price, and an amount.

```
 _id: ObjectId("5ece4da60b96e35e5ced974a")  
date: 1589862419  
user_id: 2055  
items: Array  
  0: Object  
    item_id: 31  
    name: "Multikernesandwich"  
    price: 1995  
    amount: 3  
  1: Object  
  2: Object  
  3: Object  
  4: Object  
  5: Object  
discount: null
```

Promo codes are stored as a document with an auto-generated `_id`, the code to use (id), the discount percentage, when it will expire, and a boolean representing if it has been used.

```
 _id: ObjectId("5ece7ccdd6ad723d40eca062")  
id: "qmkeqtob"  
percentage: 15  
used: false  
expires: 1594816829
```

Conclusion

Every database has the potential to solve every problem, though not equally efficient. When deciding the different databases for this project, we had to look at which databases could solve our different use cases most convenient and efficiently.

Set-up

mongo-shell

```
mongoimport --db db_exam --collection orders --file data/orders.json --jsonArray
```

```
mongoimport --db db_exam --collection promo_codes --file data/promo_codes.json --jsonArray
```

Mongodb roles

```
db.createRole(
  {
    role: "userrole",
    privileges: [
      { resource: { db: "dbexam", collection: "orders" }, actions:
[ "find", "insert" ] },
      { resource: { db: "dbexam", collection: "promos" }, actions:
[ "find", "insert", "update","remove" ] },
    ],
    roles: []
  }
)

db.createUser(
  {
    user: "user",
    pwd: "1234",
    roles: [
      { role: "userrole", db: "dbexam"}
    ]
  }
)
```

PostgreSQL

```
CREATE TABLE "user" (
  "id" SERIAL PRIMARY KEY,
  "name" varchar NOT NULL,
  "email" varchar UNIQUE NOT NULL,
```

```
"phone" varchar UNIQUE NOT NULL,  
"street" varchar NOT NULL,  
"password" varchar NOT NULL,  
"fk_zip" int NOT NULL  
);
```

```
CREATE TABLE "region" (  
  "zip" int PRIMARY KEY,  
  "name" varchar NOT NULL,  
  "lat" decimal NOT NULL,  
  "lon" decimal NOT NULL  
);
```

```
CREATE TABLE "favorite" (  
  "fk_user_id" int NOT NULL,  
  "fk_label_id" int NOT NULL,  
  PRIMARY KEY (fk_user_id, fk_label_id)  
);
```

```
ALTER TABLE "user" ADD FOREIGN KEY ("fk_zip") REFERENCES "region" ("zip");
```

```
ALTER TABLE "favorite" ADD FOREIGN KEY ("fk_user_id") REFERENCES "user" ("id") ON  
DELETE CASCADE;
```

1. Create database and run above code to create tables
2. Add Data to tables:
DB-Exam/dbex/regions.csv
DB-Exam/dbex/users.csv
DB-Exam/dbex/favorites.csv
3. Run all scripts from:
DB-Exam/SQL-scripts/...
4. Add Database info to: "DB-Exam\python\db_access.py" file

Neo4j scripts

// load

```
LOAD CSV WITH HEADERS FROM 'file:///foodDone.csv' AS r  
MERGE (item: Item {name:r.name, price: toInteger(r.price), stock: toInteger(r.stock), img:  
r.link})  
MERGE (brand: Brand{brand: r.brand})  
  
FOREACH(label IN split(r.labels, ";")|MERGE (l:Label {name: label}))
```

// item label relation

```
LOAD CSV WITH HEADERS FROM 'file:///foodDone.csv' AS r
MATCH (item: Item {name: r.name})
```

```
UNWIND split(r.labels, ",") as l
MATCH (label: Label {name: l})
```

```
MERGE (item)-[:is]-(label)
```

// item brand relation

```
LOAD CSV WITH HEADERS FROM 'file:///foodDone.csv' AS r
MATCH (item: Item {name: r.name})
MATCH (brand: Brand {brand:r.brand})
```

```
MERGE (item)-[:by]-(brand)
```

// Filtre efter label

```
match (:Label{name:"Discount"})--(item:Item)
return item.name
```

// Filtre efter fav

```
UNWIND ["Frugt", "Gin"] as fav
MATCH (l:Label {name: fav})--(i:Item)
RETURN i
```

// order price

```
MATCH (i:Item)
RETURN i
ORDER BY i.price
```

// filter price

```
MATCH (i:Item)
WHERE i.price > 1000 and i.price < 2000
RETURN i
```