Project Specification Databases

Authors:	André-Anan Gilbert, Annika Dackermann, Valentin Müller
Matriculation Numbers:	3465546, 5562028, 4616344
Course:	WWI20DSB
Subject:	Databases
Lecturer:	Petko Rutesic

Table of Contents

Table of Figures 3
1 Problem Specification 4
2 Database Modeling 6
2.1 ER Diagram 6
2.2 Transformation to a Relational Model 7
2.3 Normalization 8
3 Application 10
3.1 Authentication 11
3.2 API Endpoints for CRUD 12

Table of Figures

Figure 1: ER Diagram.. 6
Figure 2: Relational Model 7
Figure 3: Architecture of the Application.. 10
Figure 4: Authentication Workflow.. 11
Figure 5: Product API.. 12

Source of all figures is that they are our own illustrations made specifically for this documentation.

1 Problem Specification

The goal of this project is the development of a web application that allows an E-Commerce shop to keep track of its data. It should allow storage of typical data like orders, customers, products, suppliers and so on. The project will consist of a database implemented with PostgreSQL, a backend server created with Python and the framework FastAPI and a frontend that utilizes NextJS. SQLAlchemy serves as the object-relational mapping tool. Application and database are served in docker containers that can be started with a docker-file that is included in the project.

The database, which was built in this project, fulfils the task of a Data Management System, which keeps track of orders, suppliers, and customers of an E-Commerce. The following text gives a description of the database. To keep track of all available products, every product has a unique product ID, name, price it is being sold for, category it belongs to and a reference to the supplier for the product. Furthermore, every category of a product has an identifier, name, and description. Furthermore, every supplier for a product has a unique identifier and further information like, for example, its name, phone number, email address, country, region, postal code, street name and house number are saved on a table. When the customer orders something, an order is being created, which references to a unit of order details. Moreover, the order details reference to a product and an order, and it contains information about the quantity of the product, the price, the price of a single unit and information about a discount. Furthermore, every order is assigned a unique ID and it also refers to the customer who made the order. Additionally, the status of the order, the order date, the target county, region, postal code, street name and house number are kept track of, and the shipping service and the employee responsible for the order are referenced. Besides keeping track of the order details, the information of the customer is also kept track of, by assigning the customer a unique customer number. Their salutation, first name, last name, email address, phone number, default payment, country, region, postal code, street name and house number are also saved on a table. In addition to the customer data, the employee data is also stored on a table. The employees are assigned a unique employee ID. Their salutation, first name, last name, unique social security number, job title, department, a reference to the warehouse they are working at, phone number, email address, country, region, postal code, street name and house number are kept track of in the employee table. Every shipping service is also assigned a unique identifier and the information of its name, phone number, email address, country, region, postal code, street name and house number are saved on a table. There is also a separate table to keep track of the assignment of postal codes to the cities, towns, or villages. The invoices which belong to an order made by a customer are assigned to a unique number, reference to a customer and an order, the invoice status, invoice date and due date are also kept track of. Moreover, for every warehouse that stores the products, a unique identifier, phone number, email address, country, region, postal code, street name and house number are saved in specific warehouse table. Finally, there is also a table that lists how many of each product is currently in stock at which warehouse.

2 Database Modeling

The following chapter outlines how a database is designed that serves as the foundation of the application described in the previous chapter.

2.1 ER Diagram

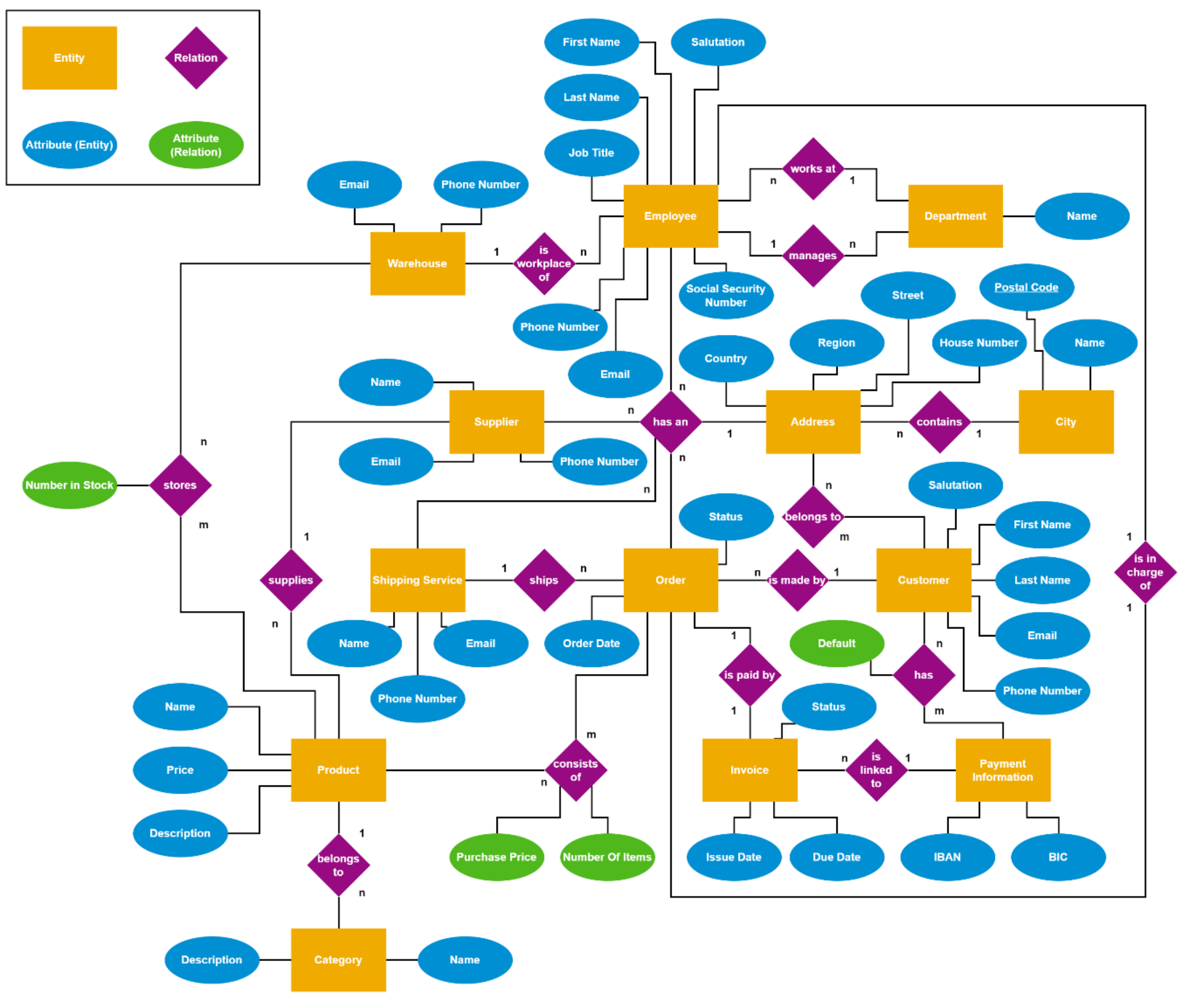


Figure 1: ER Diagram

The first step in the implementation of a database is the creation of an ER Diagram, which describes the relation of objects in the application or problem-specific domain. The ER Model of this project is shown in Figure 1. It shows different types of relationships that are characterized by the cardinalities. The most abundant case is the case of a many-to-one, but also several many-to-many relationships are included, for example between products and warehouses or customers and orders. Order and invoice are associated with a one-to-one relationship, the entity invoice acts as a weak entity, because an invoice can only exist in connection with an order.

The picture of the ER Diagram is included in the project files with full resolution.

2.2 Transformation to a Relational Model

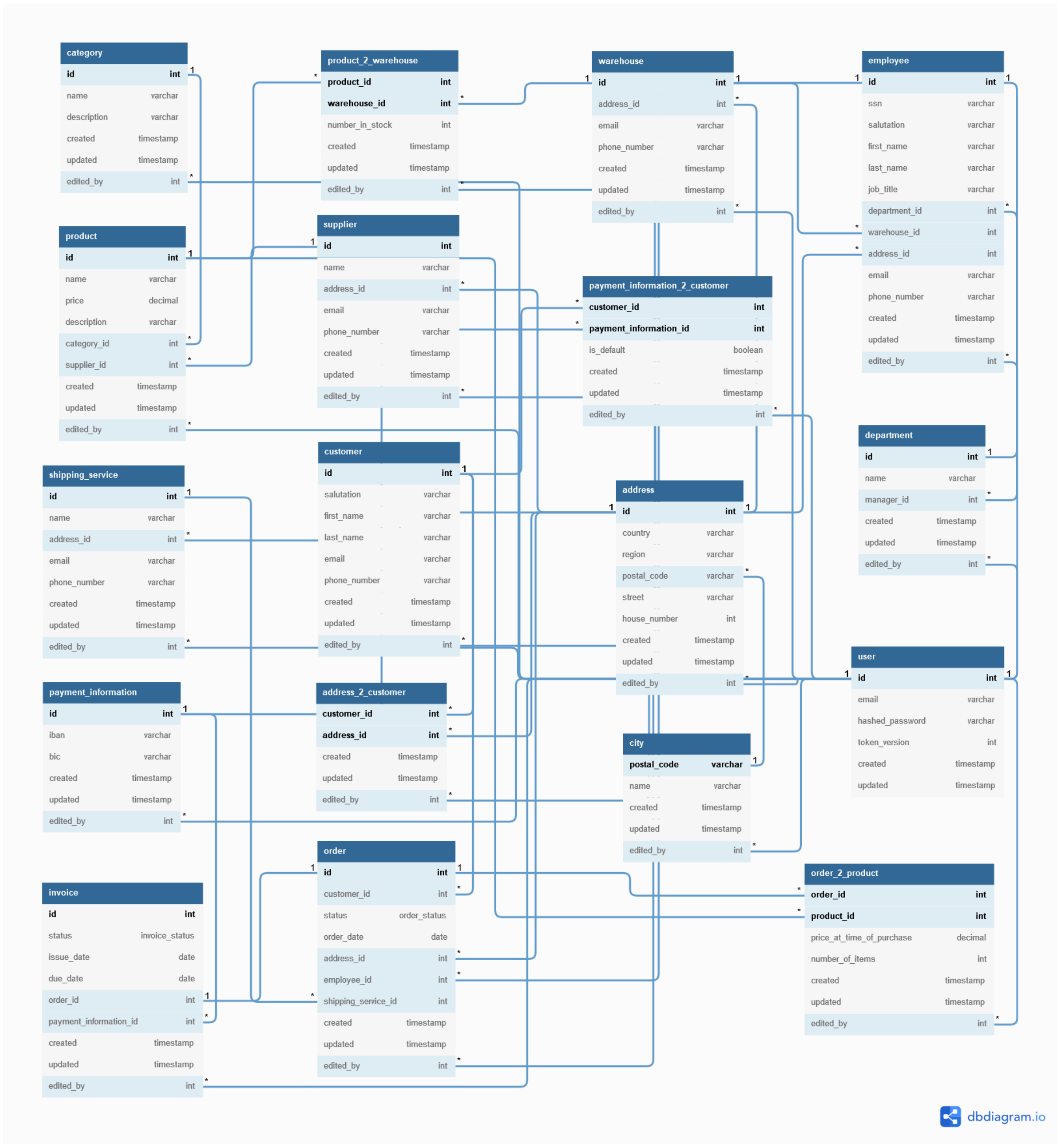


Figure 2: Relational Model

The design of the actual database is derived from the ER Model shown in Figure 1. The Relational Model in Figure 2 represents the design of the database. Custom ids to serve as primary keys have been added to all tables except the city table, which possesses a “natural” identifier in the postal code. A user table has been added, since it is a purely technical entity that is not related to the problem-domain. The attributes of other tables have been extended with the timestamps for creation and last update, as well as logging of the user that conducted the change.

Relationships are implemented with the addition of the primary key of one table as a foreign key into the other. In the case of one-to-many, the foreign key originates from the table that is the “one-partner” in the relationship and is added to the “many-partner”. For each many-to-many relationship pictured in the ER Model an association table is created, that possesses a primary key consisting of the primary keys of both entities as foreign keys. The one-to-one relationship is implemented similarly to the one-to-many but utilizes a unique constraint in the table that contains the foreign key to ensure that it is implemented correctly.

The Relational Model pictured in Figure 2 is included in the project files in full resolution, since it may be hard to read in this document.

2.3 Normalization

The following subchapter will explain why the built database application satisfies the required third normal form standard. A database should satisfy the third normal form to prevent anomalies and redundancies in the data. It is important to prevent this, as it will otherwise lead to inconsistencies when a query is executed. As shown in figure 2, all attributes in each table are functionally dependent on the primary key, like for example the product ID or the customer ID. All other keys in each table are references, such called foreign keys, to other tables, for example, department ID, warehouse ID and address ID in the employee table point to different tables. The department ID references the department, the warehouse ID references the warehouse and the address ID to the address table. Another reason this database application satisfies the third normal form, is that the attributes are all atomic, which means that the attributes are not assigned to multiple values of the same format. This can be shown in the Employee/Customer table as a person's name is split into first name and last name. The third normal form is also clearly recognizable by the outsourcing of the postal code in the address table to a sub-table city, in which the postal codes are assigned to its city, town or village. It is an important part of the third normal form, so that the attributes within the address table are only dependent on one key and do not have a transitive dependency on another key.

3 Application

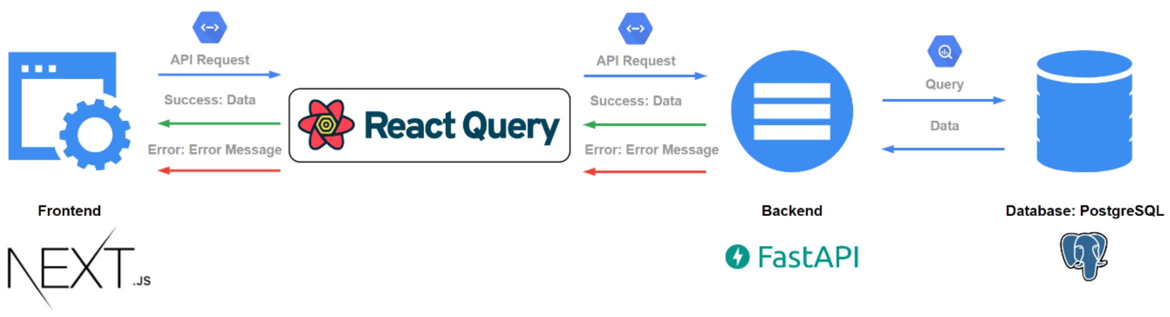


Figure 3: Architecture of the Application

The architecture of the complete application is detailed in Figure 3. It follows the three-layer architecture that is typical for web applications and consists of a frontend and backend servers as well as a database.

The whole application is wrapped within a Docker-container. This allows for building, configuration and deployment of the whole application based on a Docker image on any common operation system.

The frontend server runs on node.js and is implemented with the framework NextJS, which is based on the library React, but additionally utilizes server-side rendering for the normally client-side rendered React components, as well as static site generation. The latter is used for all views, only the API calls are executed on the client side.

The API is based on the standard REST (REpresentational State Transfer) and provided by the backend server. The backend is implemented with the Python-Framework FastAPI. This framework not only allows for intuitive and fast development of APIs that are fully compatible with open standards like JSON and OpenAPI, but also by default includes SwaggerUI, which not only provides automatic documentation of the API but also allows for testing of the endpoints without the use of additional tools like Postman. The data synchronization between front- and backend is handled by React Query an opinionated data-fetching library that handles fetching, caching, and updating server state.

The database uses PostgreSQL as its relational database management system. The backend uses queries to read and write information from and to the database. Those are implemented with the help of the object relational mapper SQLAlchemy. The initial creation of the database is handled with Alembic. On application startup, a check is conducted whether the database is already populated with data, if not the initial data is inserted with the help of SQLAlchemy.

The application also includes an instance of PGAdmin to allow for easy access and management of the database. An overview of all parts of the application (along with the respective link to access each part) is included in the README-file.

3.1 Authentication

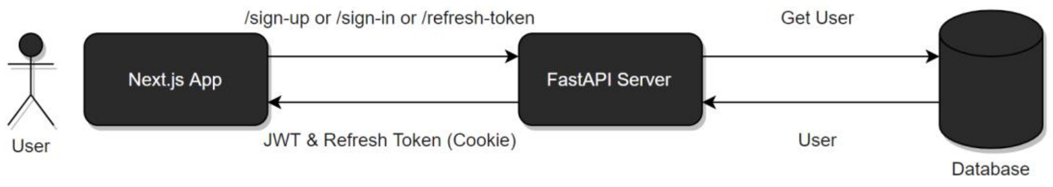

/sign-up or /sign-in or /refresh-token
Get User
Next.js App
FastAPI Server
User
JWT & Refresh Token (Cookie)
User
Database

Figure 4: Authentication Workflow

To secure communication between client and server, we have implemented cookie-based authentication, which is used to recognize user identity against credential information such as username or passwords. There are 3 endpoints which handle the entire authentication logic:

/user/sign-up

This endpoint allows the user to create an account with email and password. After the user has signed up, the backend will send an access token that will be stored in a cookie on the frontend and an HTTP only cookie containing the refresh token.

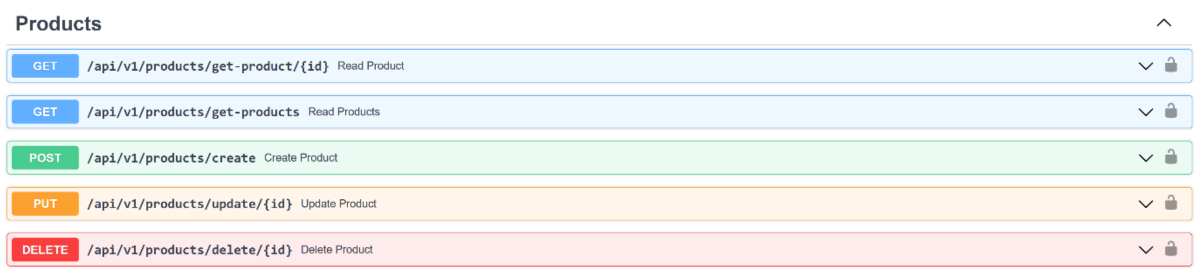
/user/sign-in

This endpoint allows the user to login (email, password) into the application. The server will send an access token and a refresh token after the server has validated if the user exists in the database.

/user/refresh-token

This endpoint is used to generate a new access token once the old access token has expired after 15 minutes. It uses the HTTP only cookie which improves protection against any Cross-site scripting (XSS) attacks since makes them impossible to read on the client-side to validate the user.

3.2 API Endpoints for CRUD



Products

GET /api/v1/products/get-product/{id} Read Product

GET /api/v1/products/get-products Read Products

POST /api/v1/products/create Create Product

PUT /api/v1/products/update/{id} Update Product

DELETE /api/v1/products/delete/{id} Delete Product

Figure 5: Product API

All other CRUD operations (that are not related to the user) can be conducted via the REST-API and are implemented in an exemplary fashion for the products. The Endpoints can be seen in Figure 5: Product APIFigure 5. For each database entity, the options create, read (single and multiple objects), update and delete should be implemented. All entity-specific operations require an authenticated user with a valid session.

Each endpoint takes parameters specified by an entity-specific schema in Python in form of a JSON object. The schema ensures that optional and mandatory parameters for the specified CRUD-operations are correctly assigned. This is required for example for create operations, where many parameters are mandatory, while most are optional when conducting an update operation. This ensures that invalid operations are identified before attempting to persist the changed in the database and causing an error.

After checking the data received from the frontend, a CRUD-repository based on the SQLAlchemy models handles the persisting of the changes. On create and update the timestamp and user log discussed in chapter 2.2 are automatically set.

Due to time constraints, the API and frontend are only implemented for selected entities. However, the database implementation is complete, as is the backend regarding repositories, schemes and SQLAlchemy models for all entities.