UNIVERSIDADE D COIMBRA
FACULDADE DE CIÊNCIAS E TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

Digital Bubble

Decision Summary

Databases project, 3rd year, 2nd semester

André Filipe Costa Colaço, 2020220301, uc2020220301@student.uc.pt

Sancho Amaral Simões, 2019217590, uc2019217590@student.uc.pt

20 de maio de 2022

This document serves the purpose of shortly enumerating and explaining some of decisions/methodologies implemented in the scope of the *DigitalBubble* project.

Database

- The database structure outputted by the tool *Onda* was found more complex than needed. Some simplifications at the database level were performed such as the removal and grouping of tables. Even though some redundancies may be present in the built database, its management and querying became much easier.
- Inheritance was used within the user and product entities in order to avoid the excess number of redundant columns in each table.
- Indexes were created in *PKs* (by default) and in *FKs* in order to speed future mass *join queries*.
- The notification system was implemented via the usage of *triggers* and *functions.*
- A database user was specifically created for the maintenance of the *DigitalBubble* database.

API

- **AUTOCOMMIT** always off.
- In order to improve the code modularity/organization, some classes and primitives were created (as in *DTOs*) to hold the incoming *HTTP* payloads. This improved the validation of the mandatory fields as well as the validation of fields' datatypes.
- The Python library *jwt* was used in order to provide and manage *JWT* tokens for the purpose of authentication and authorization.
- A *Python* decorator (annotation) was created so that it could be placed above each endpoint method and therefore specify which roles could access it.
- The data correspondent to the requesting user is accessible through the provided *JWT* token (~ session behaviour).
- In order to prevent concurrency conflicts when placing an order for a specific set of products, before initiating the correspondent transaction, its isolation level is set to *serializable* so that every overlapping set of identical transactions is scheduled as if they were to be executed in a serial way.
- For security reasons, the user's password is not put directly in the database: it is first subject to the *SHA-512* hashing algorithm.
- The *SQL* code that corresponds to the retrieval of the data of a certain product was a bit difficult to think of so that it could fit in only one query (only one access to the database). The group came to the conclusion that, even though the data looked like

incompatible to put in only one *SELECT* statement, it could fit in separate *SELECT* statements, wired through *UNION*'s and the usage of *typecasts* and some *NULL* columns (the *UNION* statement requires that all rows have the same number of columns and of the same type).

(Future) Implementation suggestions (not executed due to the time shortage)

- Creation of a table that stores every exception/error for the purpose of maintenance and auditing.
- Due to the usage of serializable transactions, connections pools may be useful in order to prevent locks/congestions of the platform.
- Addition of history behaviour for every strong entity, such as done with the product, in order to improve the platform auditability.
- Creation of more indexes based on the platform needs (most common queries).
- Separation of each entity *API* methods in controllers (following the *MVC* design pattern).
- Further/better usage and implementation of *DTO*-like entities.
- Benchmarking of the database by feeding mass queries to it.
- Implementation of the *webclient* (*aka frontend*).
- Documentation of the *Python* code.