



UNIVERSITÀ DI PISA

COMPUTER ENGINEERING

Large-Scale and Multi-Structured Databases

WORKGROUP PROJECT DOCUMENTATION

Design and development of “*cybuy*”:

an Application interacting with NoSQL Databases

Students

Federica Baldi
Tommaso Burlon
Tommaso Giorgi

Academic Year 2020/2021

CONTENTS

1	DESIGN.....	1
1.1	THE APPLICATION	1
1.2	REQUIREMENTS.....	1
1.2.1	Main actors.....	1
1.2.2	Functional requirements.....	2
1.2.3	Non-functional requirements.....	3
1.3	USE CASES	4
1.4	ANALYSIS CLASSES	4
1.5	DATA MODEL.....	6
1.5.1	Document DB collections	6
1.5.2	Key-Value DB namespaces and keys.....	12
1.6	DISTRIBUTED DATABASE DESIGN	13
1.6.1	Replicas	13
1.6.2	Sharding.....	15
1.7	SYSTEM ARCHITECTURE.....	15
1.7.1	Client Side.....	15
1.7.2	Server Side.....	16
1.7.3	Frameworks.....	16
2	IMPLEMENTATION.....	18
2.1	DATASET.....	18
2.2	REPOSITORY STRUCTURE	18
2.3	MAIN MODULES	19
2.3.1	Python Module	19
2.3.2	Java Modules	19
2.4	CYBUY: MAIN PACKAGES AND CLASSES.....	24
2.4.1	The gui package	25
2.4.2	The persistence package	25
2.4.3	The middleware package	26
2.5	MOST RELEVANT QUERIES	26
2.5.1	MongoDB queries	26
2.5.2	MongoDB aggregations.....	36

2.5.3	Key-Value Database queries.....	41
3	TEST.....	49
3.1	STATISTICAL ANALYSIS.....	49
3.1.1	MongoDB Indexes.....	49
3.1.2	Brief Consideration about the CAP Theorem	50
3.2	USER MANUAL.....	51
3.2.1	Navigation bar	51
3.2.2	Registration	52
3.2.3	Login.....	52
3.2.4	Browsing Products	53
3.2.5	Product Details	54
3.2.6	Account Page	55
3.2.7	Orders History.....	56
3.2.8	Standard User functionalities	58
3.2.9	Seller functionalities.....	60
3.2.10	Administrator functionalities.....	61
4	LIST OF FIGURES	62

1 DESIGN

1.1 THE APPLICATION

cybuy is an e-commerce application whose focus is on electronic products. It aims to make the purchase process easier for customers and to allow sellers to manage their products and their sales.

As previously stated, the application can be used both from customers and sellers.

Customers can use the application in order to browse the list of available products, see the details of a specific product, and decide whether to add it to their cart or to their wishlist.

In addition, customers can place orders, keep track of their previous ones and they can decide to leave a review on the products they have already bought.

On the other hand, sellers can manage their inventory – by adding new products and modifying or deleting old ones – and they can visualize the list of orders and fulfill them.

cybuy also provides an analytics section, through which sellers can visualize sales reports and reviewing performance, and take business decisions accordingly.

1.2 REQUIREMENTS

1.2.1 Main actors

The application is meant to be used by four different types of actors:

- *Anonymous Users*, who can browse products and their details, but are not allowed to buy or do anything. They can log into the application with their *username* and *password*, or they can sign up (either as *Standard Users* or *Sellers*).
- *Standard Users*, who are the customers of the e-commerce application. They can search for products (applying filters on demand), place orders and manage their cart and wishlist.
- *Sellers*, who can put products up for sale, manage their incoming orders and perform sales analytics.
- *Administrators*, who are special *Sellers* holding high-level privileges. Indeed, they can delete other *Sellers* when necessary (for instance, if they are scammers or if they have only bad reviews).

1.2.2 Functional requirements

The functional requirements of the application are listed below.

- The application must provide a registration form in order to allow new users to sign up, either as *Standard Users* or as *Sellers*¹.
- The application must handle the login process, so that *Anonymous Users* can identify themselves via a *username* and a *password*.
- The application must handle the logout process, so that *Standard Users*, *Sellers*, and *Administrators* can disconnect from the system.
- The application must also give *Standard Users*, *Sellers*, and *Administrators* the opportunity to delete their account if they no longer wish to use the offered services.
- *Anonymous Users* must be denied the chance to buy products or to sell them.
- *Anonymous Users* and *Standard Users* must be given the ability to browse the list of for-sale products, optionally using parameters².
- When selecting a product, *Anonymous Users* and *Standard Users* must be able to view its details.
- While viewing the details of a product, *Standard Users* must be given the opportunity to add the product either to their cart or to their wishlist.
- *Standard Users* must be offered the possibility to manage their cart, either by changing the quantity of a product, removing a product, or emptying the whole cart.
- When the cart contains the products to be purchased in the right quantity, *Standard Users* must be able to place an order.
- The application must provide *Standard Users* with an orders section, where they can view their order history and the details of each order (such as its status and the list of ordered products).
- If an order has not been shipped yet, *Standard Users* must be able to cancel it.
- *Standard Users* must be offered the possibility to manage their wishlist, both by removing a product and moving it to the cart.
- *Standard Users* must be able to leave reviews³ of products they have purchased and already received.
- *Sellers* must be given the opportunity to put products up for sale, specifying their details and the available quantity.

¹ Note that it is not possible to sign up as *Administrators*. In fact, application *Administrators* are already pre-registered.

² Parameters provided by the application are filter by *category* (and *subcategory*), filter by *keyword*, sort by *price* and sort by *reviews*.

³ A one-to-five-star rating of the overall product quality and seller reliability.

- The application must show *Sellers* the list of products they have put on sale. When a product is selected, the application must display its details.
- *Sellers* must be able to edit the details of products they have already listed for sale and, whenever necessary, to delete a product.
- The application must provide *Sellers* with a section related to incoming orders, by which they can view the list of ordered products, change the status of orders, and fulfill them.
- The application must provide *Sellers* with an analytics section, where they can view reports and sales statistics. In particular, *Sellers* can view:
 - their best-selling *Product*;
 - how many *Products* they have sold;
 - their total earnings;
 - the average delivery time;
 - the average number of stars received and their distribution;
 - a monthly chart showing daily sales, with some statistical information about customers (such as their gender and age).
- *Administrators* must be offered all the functionalities offered to *Sellers*.
- *Administrators* must be given the authority to delete other *Sellers'* accounts if the need arises.

1.2.3 Non-functional requirements

The following list outlines the non-functional requirements of the application.

- *High Availability*: The application must guarantee 24/7 service, so that users can use it whenever they want to.
- *Usability*: The application must be *user friendly*, that is easy to use, with a simple and intuitive user interface.
- *Performance*: The application must ensure short response time and low latency, so that its use is enjoyable.
- *Fault tolerance*: The application must be able to continue operating properly in the event of a failure.
- *Scalability*: The application must be able to handle a growing number of users and products on sale.
- *Security*: Users passwords must be protected and stored encrypted.

1.3 USE CASES

Figure 1 shows the UML use case diagram representing users' interaction with the system.

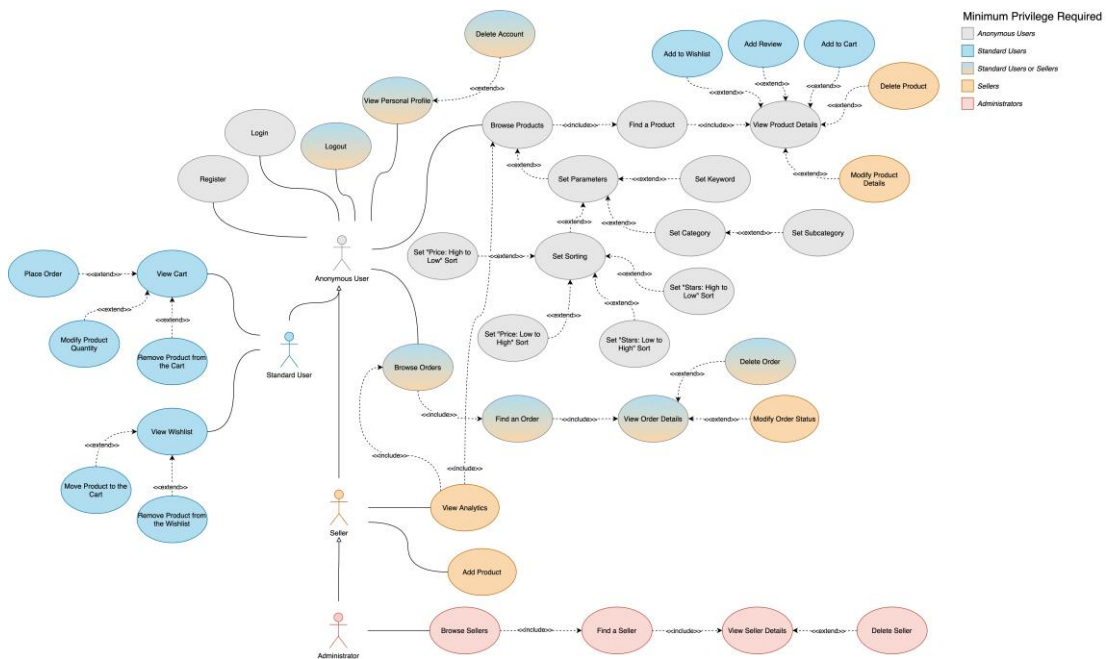


Figure 1. UML diagram of the main Use Cases

The main use cases are explained in detail below.

- **Browse Products:** The application will show a list of products, along with their description, image, and price. For *Anonymous Users* and *Standard Users*, this will be the list of all available products, while for *Sellers* it will be the list of products they have put up for sale.
- **View Product Details:** By selecting a product in the browsable list, users are given a more in-depth view of it. *Standard Users* can add the product to their cart or to their wishlist, or they can leave a review (if they have purchased and already received the product). The *Seller* of the product may change its details or decide to delete it, namely remove it from sale.
- **Browse Orders (requires Login):** The application will display the list of all placed orders. For *Standard Users*, this will be the list of outgoing orders, while for *Sellers* it will be the list of incoming ones. Both *Standard Users* and *Sellers* are given the opportunity to delete an order if it has not been shipped yet. Only *Sellers* are given the ability to update the status of the order, as it is being processed, shipped and, finally, delivered.

1.4 ANALYSIS CLASSES

Figure 2 shows the main entities of the application and the relationships between them.

Users are characterized by their *name* and *surname*, *email*, a *username* and a *password*, their *gender*, *age*, and their *location*.

Users must be either Standard Users or Sellers; Sellers may also be Administrators.

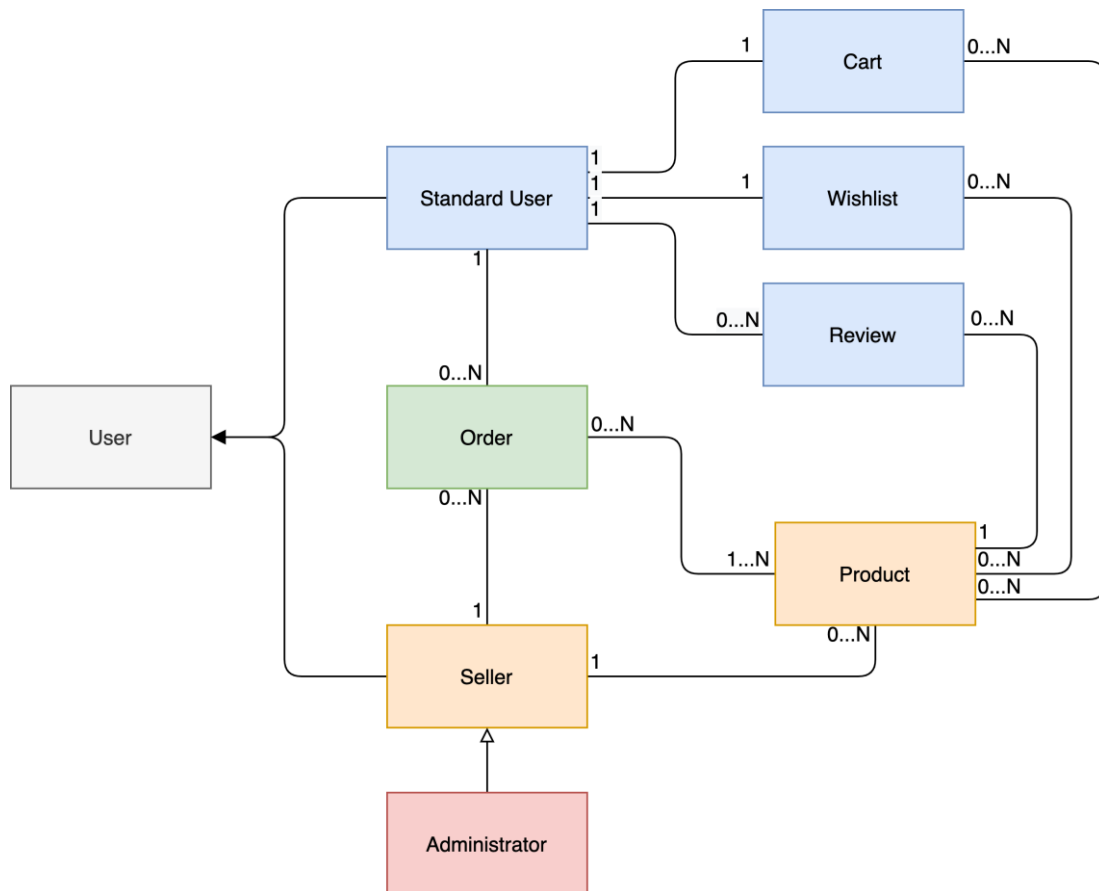


Figure 2. UML diagram of the Analysis Classes

Each *Seller* may have none or many *Products* up for sale. Each *Product* must have its own seller.

All *Products* have some common information associated with them – such as a short *description*, an *image*, and their *price* – together with some specific fields.

Indeed, the *details* associated with a product can vary, in both nature and quantity, depending on the *type of the product* (which is itself an attribute).

Standard Users are (uniquely) associated with their personal *Cart* and *Wishlist*; both may contain zero or many *Products*.

In the *Cart*, each *Product* is associated with its *quantity*, whereas in the *Wishlist*, each *Product* is associated with the *date* of its addition to the list.

Standard Users may have placed zero or many *Orders* and, similarly, *Sellers* may have zero or more incoming *Orders*.

Each *Order* contains at least one *Product*, but it may contain many, and it must be associated with the *Standard User* who placed it and the *Seller* who put the *Product(s)* up for sale.

Furthermore, *Orders* are characterized by some information such as the *date* they were placed, their *status* (e.g., “sent”, “delivered”, etc.) and their *total amount*.

Standard Users may have left none or many *Reviews* for the *Products* they purchased. Each *Product* may not have any *Reviews* yet, or it may have many.

Reviews must be associated with the *Standard User* who left them and the *Product* they refer to. Each *Review* also contains the number of *stars* assigned to the *Product* (from one to five).

1.5 DATA MODEL

1.5.1 Document DB collections

The Document Database will handle the entities of *User* (all its specializations), *Product*, *Order*, and *Review*.

Figure 3 below shows the corresponding section of the UML diagram of the *Analysis Classes* for greater clarity.

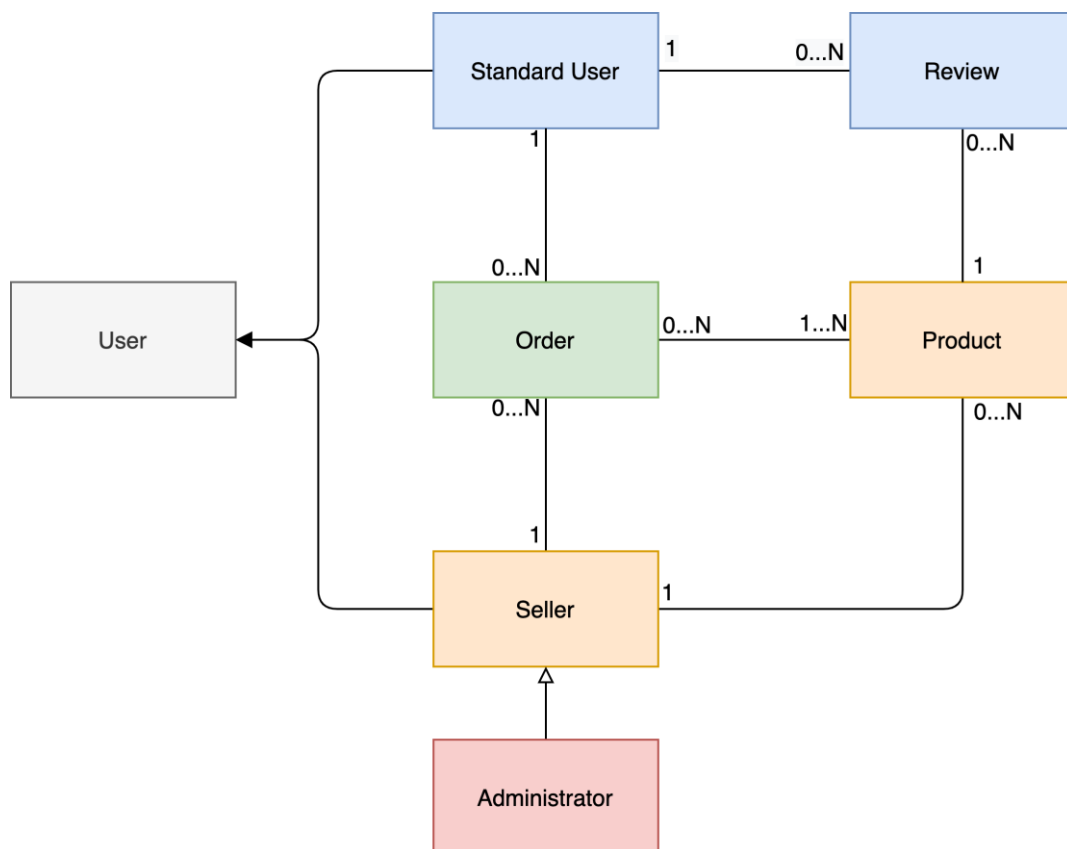


Figure 3. Section of the UML diagram of the *Analysis Classes*

Documents will be organized in four collections, namely *Users*, *Products*, *Orders* and *Analytics*.

1.5.1.1 Users Collection

The *Users* collection will maintain information about all types of *Users* (*Standard Users*, *Sellers*, and *Administrators*).

Fields containing general personal information such as *name*, *surname*, *email*, *password*, *username*, *gender*, *age*, and *location* are present in all documents.

For security reasons, the *password* field is an embedded document containing the *User's* encrypted password and the encryption key required for authentication (*salt*).

Also, a *role* field is present to distinguish between the three types of *Users* and to offer them the different functionalities of the application accordingly.

```
{
  "_id": {
    "$oid": "5fef4603c93ea608c9ef8ead"
  },
  "country": "Italy",
  "role": "User",
  "name": "Tommaso",
  "surname": "Giorgi",
  "email": "tommy@mail.it",
  "username": "Tomawrk",
  "gender": "Male",
  "age": 22,
  "orders": [
    {
      "_id": {
        "$oid": "60095be5a701245a1000bb30"
      },
      "price": 129.99,
      "seller_username": "RassyuM",
      "state": "pending",
      "order_date": {
        "$date": "2021-01-21T10:48:05.547Z"
      },
      "orderedProduct": [
        {
          "productId": "5fe9a411845df004b4acebd5",
          "ordered_quantity": 1,
          "description": "Pritom 10 inch Android Tablet Octa-Core, 3GB RAM, 5G WiFi, GPS, Bluetooth 5.0, ...",
          "image": "https://m.media-amazon.com/images/I/71CPCbUsZ9L._AC_UY218_.jpg"
        }
      ]
    }
  ],
  "reviews": [
    {
      "productId": "5fdb8edd0e189329746b6e24",
      "stars": 3
    },
    {
      "productId": "5fe9a2d2845df004b4acbaa8",
      "stars": 5
    }
  ],
  "password_info": {
    "password": "9M7EjfxTiffu866mbt6w7YnmGqiOiouKAWiP13XvvKA=",
    "salt": "LgFEuZu9hMRyKU5h6vmKevYZwDbjJ8"
  }
}
```

Figure 4. Example of document in the *Users* collection (*Standard User*)

The *orders* field is an array of nested documents regarding *Orders*. As can be seen in Figure 4 and Figure 5, the fields of the nested documents are slightly different depending on the type of *User*.

For instance, *Order* documents for *Standard Users* will contain the username of the *Seller*, while *Order* documents for *Sellers* will contain the username of the *User*.

In addition, redundancies regarding the ordered products are also included in the *Users'* documents. In this way, some details about them will be easily visible along with the order overview.

Since letting the number of members in embedded documents increase without limit can lead to problems, once delivered, *Orders* will be moved to the *Orders* collection that will be presented in detail later.

Figure 4 shows how only documents regarding *Standard Users* contain a *reviews* field, which is an array of embedded documents. Each *Review* contains the *ID* of the *Product* it refers to and the number of *stars* assigned to it.

Documents regarding *Sellers*, instead, contain a field for *products on sale*, that is an array of *IDs* of *Products* they put up for sale (Figure 5).

```
{
  "_id": {
    "$oid": "5fdf8787a8335e0908c1e30c"
  },
  "role": "Seller",
  "products_on_sale": [
    "5fe9a411845df004b4acebd5",
    "5fe9a420845df004b4acefd1",
    "5feb41a684937b2318c556dc",
    "5feb41a784937b2318c55ad8"
  ],
  "name": "Abigail",
  "surname": "Titus",
  "email": "nokia_20008@xxxx.xxx",
  "username": "RassyuM",
  "gender": "Female",
  "country": "France",
  "age": 38,
  "orders": [
    {
      "_id": {
        "$oid": "60095be5a701245a1000bb30"
      },
      "price": 129.99,
      "user_username": "Tomawk",
      "state": "pending",
      "order_date": {
        "$date": "2021-01-21T10:48:05.547Z"
      },
      "orderedProduct": [
        {
          "productId": "5fe9a411845df004b4acebd5",
          "ordered_quantity": 1
        }
      ]
    }
  ],
  "password_info": {
    "password": "Y00kEFuJ3vcSeGcY8CbfcUnSQL15tYOPH913kHmWFTY=",
    "salt": "OuTZdn4EDEV0x4yCBK4YRIIzxLfRJR"
  }
}
```

Figure 5. Example of document in the *Users* collection (*Seller*)

Although document linking is somewhat unnatural for document databases, it seemed the most appropriate choice given the nature of the queries the application must answer.

In fact, it is important for documents regarding *Products* to be stored in a separate collection, in order to make the search and purchase processes more straightforward.

Also, since a lot of information must be maintained for each *Product*, it would have been unfeasible to embed *Products* documents into the document of their *Seller*.

1.5.1.2 Products Collection

The *Products* collection will maintain information about all types of *Products*.

The fields *description*, *image*, *product type*, *price*, and *quantity available* are present in all documents and must be specified by the *Seller* when inserting a new *Product*.

The *details* field contains a document whose size and content vary depending on the type of *Product*. For example, since the *Product* whose document is shown in Figure 6 is a tablet, information about the operating system, storage capacity, and screen size are listed.

The *seller* field contains the username of the *Seller* who put the *Product* up for sale. The application will take care of keeping this reference consistent with the *products on sale* field in the document of the *Seller*.

All other fields, namely *quantity sold*, *total reviews* and *average review*, will be automatically added and kept updated by the application.

These fields are redundant, because their value could be derived each time from other data present in the database. However, since they do not require large amounts of memory, it is more convenient – performance wise – to store them than to compute them on-demand.

```
{
  "_id": {
    "$oid": "5fe9a410845df004b4aceb85"
  },
  "description": "VANKYO MatrixPad S8 Tablet 8 inch, Android 9.0 Pie, 2 GB RAM, ...",
  "image": "https://m.media-amazon.com/images/I/71zBBUXrnSL._AC_UY218_.jpg",
  "product_type": "tablet",
  "details": {
    "Display_Size": "8 Inches",
    "Screen_Resolution": "1280x800",
    "Brand": "Vankyo",
    "Model_Number": "S8",
    "Operating_System": "Android 9 Pie",
    "Weight": "11.2 ounces",
    "Product_Dimensions": "8.21 x 4.89 x 0.35 inches",
    "Color": "Black",
    "Storage": "32 GB"
  },
  "total_reviews": 2,
  "quantity_sold": 3,
  "quantity_available": 44,
  "price": 84.99,
  "average_review": 4.5,
  "seller_username": "nsfw_1324"
}
```

Figure 6. Example of document in the *Products* collection

1.5.1.3 Orders Collection

The *Orders* collection will maintain information about *Orders* that have already been received. As anticipated earlier, once an *Order* is delivered, its data within *Users'* documents are deleted and moved to this collection.

This solution has several advantages. First, it eliminates the duplication of information regarding the same *Order* in multiple documents.

Moreover, in some way it puts a limit to the size of *Users'* documents. In fact, supposedly a single *Standard User* will not place many *Orders* together in a short period of time and *Sellers* will try their best to fulfill *Orders* as soon as possible.

The main drawback is that retrieving past *Orders* will be definitely slower than retrieving current ones.

However, *Standard Users* will most likely be interested in viewing in-progress *Orders* and their status updates, and less interested in viewing those regarding items they have already received.

Similarly, *Sellers* will be more interested in viewing the details of *Orders* they have yet to ship, rather than those they have already fulfilled.

As shown in Figure 7, each document in the *Orders* collection contains fields about *price*, *seller username*, *user*, *dates of delivery* and *order*, and a field for *ordered products*.

This last field is an array of embedded documents, each of which contains the ID of the *Product* ordered and in what *quantity*.

The *user* field is also an embedded document, containing the *username* of the *User* who placed the *Order* and some of his/her personal information that will be useful for computing analytics.

```
{
  "_id": {
    "$oid": "600424b6b4b83e62f962d1b6"
  },
  "price": 249.99,
  "seller_username": "Jer456556412",
  "user": {
    "username": "Tillandz",
    "age": 32,
    "gender": "Female"
  },
  "delivery_date": {
    "$date": "2021-01-03T11:19:54.818Z"
  },
  "order_date": {
    "$date": "2021-01-01T11:19:54.818Z"
  },
  "orderedProduct": [
    {
      "productId": "5fe9a3ff845df004b4ace8ad",
      "ordered_quantity": 1
    },
    {
      "productId": "5fdb8edc0e189329746b645e",
      "ordered_quantity": 1
    }
  ]
}
```

Figure 7. Example of document in the *Orders* collection

1.5.1.4 Analytics Collection

The *Analytics* collection will contain all documents related to the analytics of the *Sellers*.

Since performing the various aggregations needed is computationally intensive, analytics cannot be computed on-demand because this would significantly slow down the application performance.

Therefore, analytics for each *Seller* are calculated once a day on the server side and then stored in the corresponding document.

Obviously, one of the disadvantages of this choice is that *Sellers* may view outdated information. However, this is acceptable because reports only need to be a guide for business decisions, and they do not have to be real-time.

Each document in the *Analytics* collection (Figure 8) contains the *username* of the *Seller* and the *month* it refers to, along with the computed analytics.

Specifically, it contains: the *total number of reviews*, the *average review* and the *distribution of stars* obtained, the *average delivery time*, the *total earnings*, the *total sales*, the *best-selling product*, and *daily information* regarding sales.

When analytics are calculated for a *Seller*, if a document already exists for that *Seller* and for that month, the document is updated, otherwise a new document is created.

In order to prevent the document from growing beyond the space allocated to it and having to be moved to another location, at the time of creation the document already contains all the required fields. In particular, the *sales by day* field already contains the embedded documents for each day of the month, with all fields set to 0.

```
{
  "_id": {
    "$oid": "60033fec64c0176144520a3d"
  },
  "seller_username": "jason8001",
  "total_reviews": 57,
  "avg_reviews": 3.5789473684210527,
  "avg_delivery": 2,
  "total_earnings": 29831.36,
  "total_sales": 95,
  "month": 1,
  "best_selling_product": {
    "productId": "5fe9a29b845df004b4acb4e3",
    "description": "iRecadata L41 Portable SSD Drive, External Solid State Drive,USB 3.0,..."
  },
  "star_distribution": {
    "1": 5,
    "2": 6,
    "3": 16,
    "4": 11,
    "5": 18
  },
  "sales_by_day": [
    {
      "day": 1,
      "all_sales": 0,
      "male_sales": 0,
      "female_sales": 0,
      "young_sales": 0,
      "old_sales": 0
    },
    {
      "day": 2,
      "all_sales": 1,
      "male_sales": 1,
      "female_sales": 0,
      "young_sales": 0,
      "old_sales": 1
    },
    {
      "day": "..."
    }
  ]
}
```

Figure 8. Example of document in the *Analytics* collection

1.5.2 Key-Value DB namespaces and keys

The Key-Value Database will be in charge of handling the entities of *Standard User*, *Cart*, *Wishlist* and *Product*.

For greater clarity, the relative section of the UML diagram of the *Analysis Classes* is displayed in Figure 9.

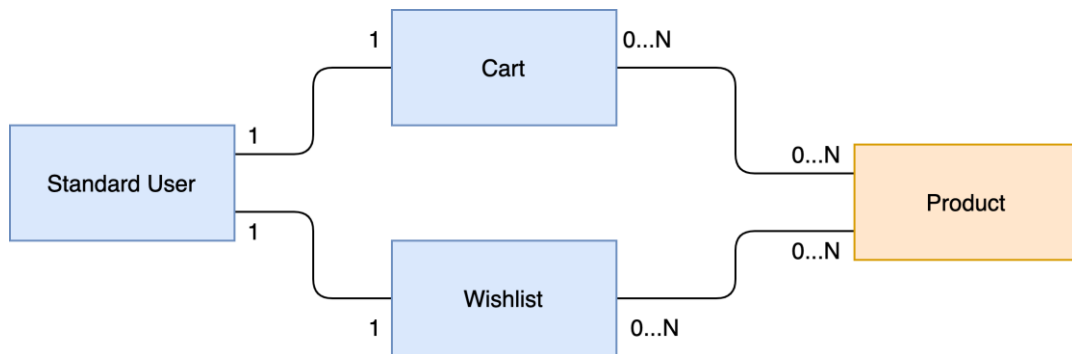


Figure 9. Section of the UML diagram of the *Analysis Classes*

Both *Cart* and *Wishlist* are involved on the one hand in a one-to-one relationship with *Standard User*, and on the other hand in a many-to-many relationship with *Product*.

With the aim of avoiding potential key-naming conflicts, two namespaces have been defined, one for the *Cart* and the other for the *Wishlist*.

This results in the addition of a prefix to the keys, namely `cart` and `wishlist`, respectively.

The final key-value configuration for the *Cart* is the following:

```
cart:user:$user_id4:product:$product_id5:quantity = $quantity_value
```

Whereas here is the final key-value configuration for the *Wishlist*:

```
wishlist:user:$user_id:product:$product_id:date = $date_value
```

The Key-Value Database will also act as a cache to make quickly accessible some information regarding *Products* placed in a *Cart* or in a *Wishlist*.

Not all details regarding a *Product* need to be visible from the *Cart* and/or in the *Wishlist*, but it would be beneficial to at least have the basic information.

For instance, in this Database redundancies will be introduced on the *Product description*, *image*, and *price*.

The resulting key-value configuration is the following:

```
product:$product_id:$attribute = $attribute_value
```

The application will be responsible for keeping the redundant information consistent with the Document Database.

⁴ Note that the `user_id` is the same one used in the Document Database to uniquely identify a *User*.

⁵ Note that the `product_id` is the same one used in the Document Database to uniquely identify a *Product*.

Also, since the Key-Value Database already tends to be memory-intensive, an algorithm that can free up some of the allocated cache memory when needed must be implemented.

To do this, a *usage* attribute is also stored for each *Product*; its value indicates how many *Carts* and *Wishlists* currently contain that *Product*.

```
product:$product_id:usage = $usage_value
```

1.6 DISTRIBUTED DATABASE DESIGN

The entire database will be deployed on a cluster of servers. At the moment, the cluster will be composed of three servers, but if in the future the load on the system increases, it will be easily possible to add servers to the cluster and scale horizontally.

1.6.1 Replicas

Since high availability and fault tolerance are two of the non-functional requirements of the application, it may be beneficial to save multiple copies of data in the cluster.

However, since performance is also among the non-functional requirements, it is important to choose a number of replicas that is not so large.

A trade-off between the two requirements could be to set to three the number of copies of each data item that the database will maintain.

1.6.1.1 Document Database Replicas

With respect to Document Database replicas, the servers will be configured in a master-slave replication model, with one primary node and two secondary nodes.

All read operations will be directed to the primary node; however, in order to ensure high availability, if the primary node is down, operations will read from secondary nodes⁶.

All write operations will be sent to the primary node, but the number of copies of the data item that must be written before the write can complete will depend on the collection (as shown in Table 1).

Collection Name	W	R
Users	3	1
Products	2	2
Orders	2	2
Analytics	3	1

W = number of copies that must be written before the write can complete

R = number of copies that the application will access when reading

Table 1. Read/Write Concerns for each Collection

The decision to distinguish read and write concerns by collection comes from the fact that different collections contain different information, with different importance and update frequency.

We want the login process to be as fast as possible and user's current orders to be quickly retrievable as well, so read operations to the *Users* collection must be fast.

⁶In MongoDB, Read Preference is set to primaryPreferred.

Writes, on the other hand, can be slower because these only occur during the registration process and order creation, times when data consistency is important, and the user may be willing to wait a little while.

Both reads and writes to the *Products* and *Orders* collections need to be reasonably fast, but still consistent. In fact, we do not want a user to see outdated information about a product or to buy a product that is actually no longer available.

Furthermore, writes, but especially reads towards the *Orders* collection will not happen as often. This is because, as we have already mentioned, users will not be very interested in viewing their past orders and, server-side, analytics will be calculated on the *Orders* collection only once a day.

As for the *Analytics* collection, write operations are sent only once a day and from the server side, so there is no need for them to be as fast as possible. Also, we want the computed information not to be lost and to be quickly accessible for users to read.

1.6.1.2 Key-Value Database Replicas

As for the Key-Value Database replicas, the servers will be configured in a masterless replication model and set up in a ring structure, as shown in Figure 10.

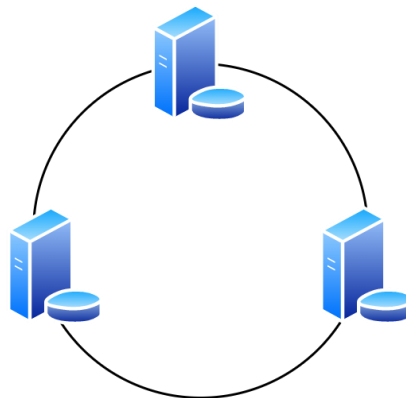


Figure 10. Ring structure

This means that there is not a single server that has the master copy of updated data, instead servers in the cluster must collaborate to keep the replicas consistent with each other.

Each server can accept both read and write requests. Whenever there is a write operation to one of the servers, the updated piece of data is also written to the two servers linked to the original one, namely its neighbors.

A write request returns once the first copy is written, the other two will happen later. Similarly, a read request reads a single version only.

Obviously, with this configuration the risk is that data will be lost if a node fails before the second write or that the data read may not be the last version. However, speed of operations and high availability must be prioritized over consistency.

Within the same session, all reads and all writes from a client will always be sent to the same server, unless the server is down. In this way, an attempt is made to ensure session consistency.

If the server fails, the client will automatically try to contact another server. Once again, availability is favored over consistency; users can accept their cart or wishlist to be inconsistent once in a while, but could not bear the application to stop working.

1.6.2 Sharding

In a scenario where the number of users grows rapidly, the size of the database will also grow accordingly, and it may become unfeasible to maintain an entire copy of the database on a single server.

Moreover, as introduced earlier, if the load of the system increases, it would be convenient to add servers to the cluster.

In this case, an appropriate solution would be sharding the data and choosing a partition scheme to distribute the workload as evenly as possible.

Since there are neither document fields nor keys that by themselves naturally distribute workloads evenly, a hash function would be appropriate to generate the shard key.

The hash function would be applied to the unique document IDs of all documents in the Document Database and to keys in the Key-Value Database.

A possible solution would be then to divide the hash value by the number of servers and to use the remainder to locate the node where the shard should be stored.

However, the application could catch on the international market and the database could grow further, requiring the addition of more servers. Or it may unfortunately prove to be a failure and some servers could be removed from the cluster.

In any case, the best solution for the application needs is to apply consistent hashing.

First, we would need to calculate the hash value of the IP addresses of all the servers in the cluster. Then, we would calculate the hash value of each document ID and each key and assign the relative object to its successor server in the address space.

The replicas for each shard would still be three and, taking advantage of the fact that the servers are already logically placed in a circle, every object would also be copied to its predecessor server and to the successor server of its successor server in the address space.

1.7 SYSTEM ARCHITECTURE

The architectural pattern used for the design of the overall system is the common *two-tier client-server* one (Figure 11). Clients, for instance the users of the application, will request to query the database to servers in the cluster.

1.7.1 Client Side

On the client side, the *Presentation Layer* and the *Logic Layer* run.

The *Presentation Layer* displays information related to the services offered by the application, such as browsing products, purchasing, and shopping cart contents. It consists of a Graphical User Interface that users can access directly.

Its main task is to translate the actions performed by users into requests for the underlying layer and, once the results are received, to translate these into something users can understand.

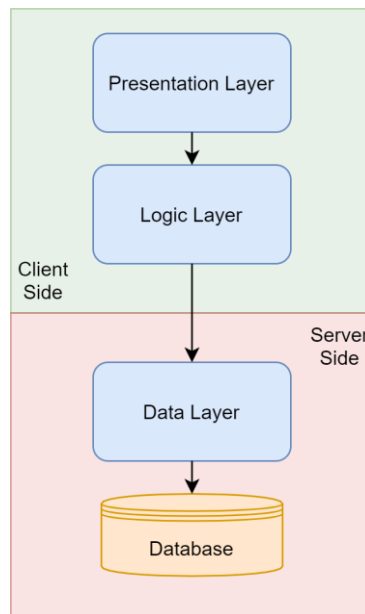


Figure 11. Two-tier Client-Server Architecture

The *Logic Layer* acts as an interface between the *Presentation Layer* and the *Data Layer*, located on the server side.

It is in charge of processing requests coming from the *Presentation Layer*, computing all the right calculations, and passing them to the *Data Layer* (via a specific communication protocol). Once the data is received from the database, the *Logic Layer* is responsible for manipulating them and sending them back to the *Presentation Layer*.

1.7.2 Server Side

On the server side, a copy of the database (both the Document Database and the Key-Value Database) is maintained, and the *Data Layer* runs.

The *Data Layer* function is to receive requests from the *Logic Layer* and to perform the necessary operation into the database.

Demands from the *Logic Layer* could be either queries to retrieve data or requests to modify the database (insertions, updates, deletions). In any case, once the demanded operation is performed, information is passed back to the *Logic Layer*, and then eventually back to the user (*Presentation Layer*).

1.7.3 Frameworks

As shown in Figure 12, the application code will be written completely in Java, with the addition of JavaFX for the GUI.

Concerning databases, MongoDB⁷ will be used for the Document Database, and LevelDB⁸ will be used for the Key-Value Database.

Since LevelDB does not natively support the creation and management of replicas, a specific service will be implemented in Java.

Apache Maven⁹ will be used as a tool for building and managing the whole application, while for version control Git (in particular, GitHub¹⁰) will be used.

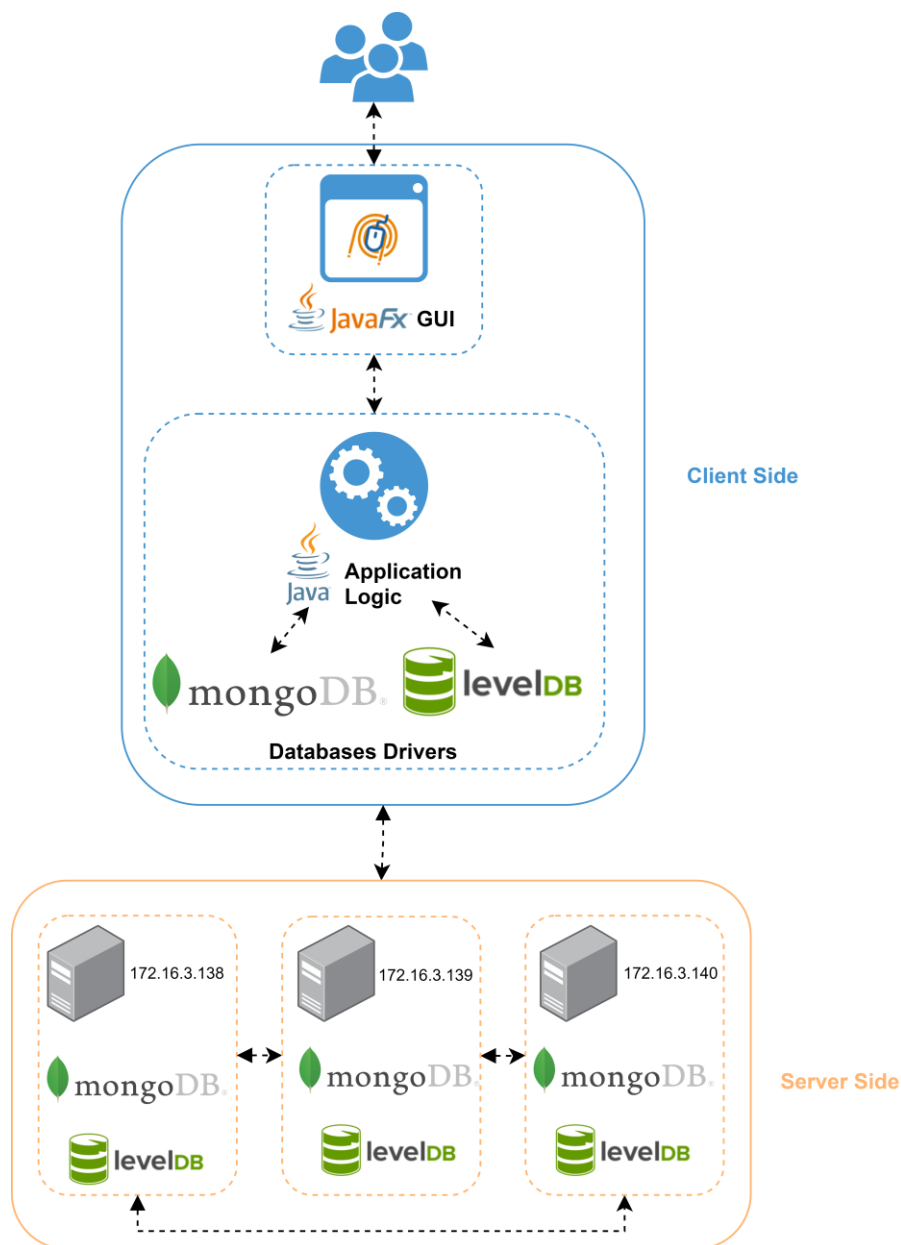


Figure 12. System architecture

⁷ <https://www.mongodb.com/>

⁸ <https://github.com/google/leveldb>

⁹ <https://maven.apache.org/>

¹⁰ <https://github.com/>

2 IMPLEMENTATION¹¹

2.1 DATASET

At the beginning of the implementation process, a real dataset that met the needs of the application had to be retrieved.

Since *cybuy* is an e-commerce application whose focus is on electronic products, there were mainly to find information about two entities: *products* and *users*.

The dataset for products is the result of scraping algorithms performed on different websites. Finding information regarding the products was not difficult, but it was necessary to take some measures to ensure that each generated document contained the fields previously indicated in the *Data Model* section.

On the other hand, finding a dataset containing real users was not possible due to privacy policies, so it was decided to combine different sources. In particular, a dataset containing usernames, a dataset containing names and another one containing emails, password, and other personal information¹², were merged to form a single dataset.

The two datasets thus obtained were inserted into the Document Database and further adjustments were then performed on them.

First, each user was assigned a role in a pseudo-random way (either *Standard User*, *Seller* or *Administrator*). Then, each product was associated with its own seller.

Finally, other entities such as orders and reviews were generated once again by some pseudo-random algorithms, in an attempt to mimic human behavior.

2.2 REPOSITORY STRUCTURE

The project repository, whose structure is displayed in Figure 13, is organized as follows:

- *AddOn*. This folder contains all the resources used in the *cybuy* module. In turn, it is organized into subfolders, containing the CSS, fonts, images and FXML needed to load the GUI.
- *cybuy*. This folder contains the code of the application to be run on the client side.
- *daemon*. This folder contains the code for a program that will run on the primary MongoDB server to update the analytics documents.
- *serverLevelDB*. This folder contains the code that will run on the three servers to manage queries and replicas on LevelDB.
- *Web Scraping*. This folder will contain some of the scripts written for web scraping.
- *Database Dump*. This folder will contain a dump of the Document Database.

¹¹ All the source code for the application and a dump of the database can be found on GitHub at the following link: <https://github.com/Tomawk/cybuy-group17>

¹² Dataset containing reddit usernames, dataset containing most popular names, dataset containing bank customers details

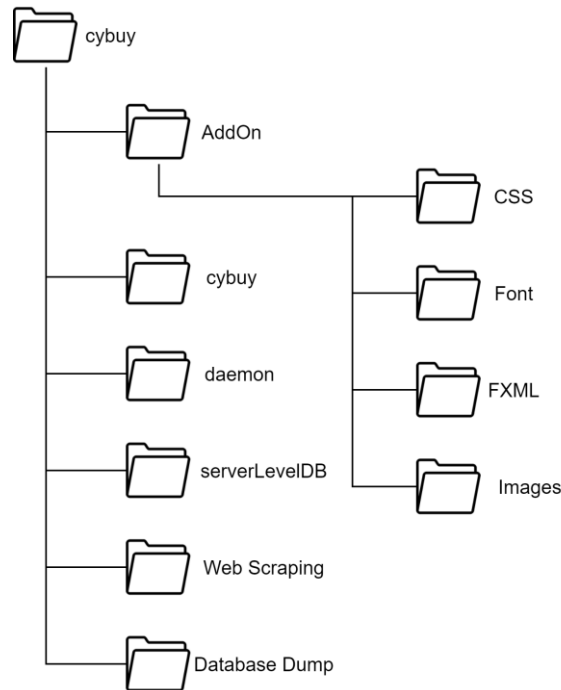


Figure 13. Directory structure of the project repository

2.3 MAIN MODULES

For the purpose of developing the application, four modules were written, three of them in Java and one in Python.

2.3.1 Python Module

The module written in Python was used in the first phase to populate the database. In fact, it consisted of some scraping scripts able to retrieve useful data from various web sites.

In particular, multiple scripts were written in order to scrape information about electronic products mainly from Amazon¹³, eBay¹⁴ and GAME¹⁵.

Some of the scripts are included among the source files on GitHub to serve as an example, but we will not discuss their implementation in detail because it is beyond the focus of this documentation.

2.3.2 Java Modules

Of the three modules written in Java, one will run client-side and the other two will run server-side.

The client module is called *cybuy* and contains all the code needed to run the application. Details about its implementation will be discussed in the next section.

The two server modules are *daemon* and *serverLevelDB*.

¹³ <https://www.amazon.com/>

¹⁴ <https://www.ebay.com/>

¹⁵ <https://www.game.co.uk/>

2.3.2.1 daemon

As its name suggests, this module contains the code for a daemon that will run on the primary MongoDB server every day at 00:01 (as shown in Figure 14).

```
workgroup-17@ProfileLARGESCALE57:~$ crontab -l
# Calculate analytics at 00.01 every day
1 0 * * * java -jar /home/workgroup-17/daemon/daemon.jar > /home/workgroup-17/daemon/log.txt
```

Figure 14. Command `crontab -l` executed on the primary server (IP: 172.16.3.138)

The purpose of this program is to calculate the analytics of all sellers and update the corresponding documents within the *Analytics* collection.

In the *main()* function, *updateAnalytics()* is called. As shown in Figure 15, this function, in turn, calls all the functions to calculate analytics for every seller present in the *Users* collection.

```
1. public static void updateAnalytics() {
2.     Date currentDate = new Date();
3.     Calendar cal = Calendar.getInstance();
4.     cal.setTime(currentDate);
5.     cal.add(Calendar.DATE, -1); // The analytics will refer to yesterday
6.     int currentMonth = cal.get(Calendar.MONTH) + 1;
7.     int analyticsDay = cal.get(Calendar.DAY_OF_MONTH);
8.     int analyticsYear = cal.get(Calendar.YEAR);
9.
10.    FindIterable<Document> documents = usersCollection.find(or(eq("role", "Admin"), eq("role", "Seller")));
11.    for (Document document : documents) {
12.        String username = document.getString("username");
13.
14.        // Calculate all the analytics
15.        User seller = User.UserBuilder().setUsername(username);
16.        getReviewsAnalytics(seller);
17.        int totalReviews = seller.getTotalReviews();
18.        double avgReviews = seller.getAverageReview();
19.        int averageDeliveryTime = getAverageDeliveryTime(seller);
20.        double totalEarnings = getTotalEarnings(seller);
21.        int totalSales = getTotalSales(seller);
22.        Product bestSellingProduct = getBestSellingProduct(seller);
23.        int[] numberOfStars = getNumberOfStars(seller);
24.        DailyAnalytics dailyAnalytics = getDailyAnalytics(seller, analyticsDay, currentMonth, analyticsYear);
25.
26.        // Search if the analytics document for this user is already present
27.        FindIterable<Document> analyticsDocument = analyticsCollection.find(and(eq("seller_username", username), eq("month",
currentMonth))));
```

Figure 15. Function *updateAnalytics()* code – PART I

The aggregations computed on MongoDB to retrieve analytics for each seller will be explained in detail in the concerning section *MongoDB aggregations*.

After performing the necessary calculations for each seller, the function checks whether or not a document for the current month already exists regarding that seller in the *Analytics* collection.

If the document already exists, it is updated with the newly calculated values (Figure 17). Otherwise, a new empty document is created, already containing all the documents concerning each day of the month, as anticipated in the *Data Model* section (Figure 16).

```

31.         if (analyticsDocument.first() == null) {
32.
33.             ArrayList<DailyAnalytics> salesByDay = new ArrayList<>();
34.             for(int i = 1; i<32; i++) {
35.                 DailyAnalytics empty = DailyAnalytics.DailyAnalyticsBuilder()
36.                     .setDay_num(i).setAll_sales(0).setFemale_sales(0)
37.                     .setMale_sales(0).setOld_sales(0).setYoung_sales(0).build();
38.                 salesByDay.add(empty);
39.             }
40.
41.             salesByDay.set(analyticsDay-1, dailyAnalytics);
42.
43.             analytics = Analytics.AnalyticsBuilder()
44.                 .setMonth(currentMonth)
45.                 .setSellerUsername(username)
46.                 .setTotalReviews(totalReviews)
47.                 .setAvgReviews(avgReviews)
48.                 .setAvgDelivery(averageDeliveryTime)
49.                 .setTotalEarnings(totalEarnings)
50.                 .setTotalSales(totalSales)
51.                 .setBestSellingProduct(bestSellingProduct)
52.                 .setStarsDistribution(numberOfStars)
53.                 .setSalesByDay(salesByDay).build();
54.
55.             Document newDoc = convertAnalyticsToDocument(analytics);
56.             System.out.println("New document: inserted analytics for seller " + username);
57.             analyticsCollection.insertOne(newDoc);

```

Figure 16. Function *updateAnalytics()* code – PART II

```

59.     } else {
60.         analytics = convertToAnalytics(analyticsDocument.first());
61.         ArrayList<DailyAnalytics> salesByDay = analytics.getSalesByDay();
62.         salesByDay.set(analyticsDay-1, dailyAnalytics);
63.
64.         analytics.setTotalReviews(totalReviews)
65.             .setAvgReviews(avgReviews)
66.             .setAvgDelivery(averageDeliveryTime)
67.             .setTotalEarnings(totalEarnings)
68.             .setTotalSales(totalSales)
69.             .setBestSellingProduct(bestSellingProduct)
70.             .setStarsDistribution(numberOfStars)
71.             .setSalesByDay(salesByDay).build();
72.
73.         Document newDoc = convertAnalyticsToDocument(analytics);
74.
75.         System.out.println("Document already present: updated analytics for seller " + username);
76.         analyticsCollection.replaceOne(eq("_id", new ObjectId(analytics.getObjectId())), newDoc);
77.     }
78. }
79. }

```

Figure 17. Function *updateAnalytics()* code – PART III

2.3.2.2 *serverLevelDB*

LevelDB does not natively provide support for creating and managing replicas, nor for multi-process access to the same database file. For this, the *serverLevelDB* module was written.

As previously mentioned, as far as it concerns the Key-Value Database, the three servers within the cluster have been organized in a masterless replication model. In order to do so, servers will need to run the code contained in this module.

The code is the same for each of the three servers, except for a few minor changes. In fact, the address and port variables for the first and second servers will be set accordingly on each server (Figure 18).

```
public class Server {
    private static LevelDB db;
    private static final String firstServerAddress = "172.16.3.138";
    private static final int firstServerPort = 1234;
    private static final String secondServerAddress = "172.16.3.139";
    private static final int secondServerPort = 1234;
```

Figure 18. Class *Server* variables

Each server is a multithreaded one; whenever a client sends a request, a thread is generated to process it. In order to accept multiple requests from multiple clients at the same time, multiple threads are generated.

By implementing a multithreaded server, the waiting time for the client decreases. Indeed, while in a single-threaded server other users have to wait until the running process gets completed, in multithreaded servers all users can get a response at a single time, so no user has to wait for other processes to finish.

Moreover, although a LevelDB database can only be opened by one process at a time, inter-thread concurrency is not an issue, as stated in the official documentation¹⁶.

“Within a single process, the same `leveldb : DB` object may be safely shared by multiple concurrent threads. I.e., different threads may write into or fetch iterators or call `Get` on the same database without any external synchronization (the `leveldb` implementation will automatically do the required synchronization).”

The *Server* class has the following tasks, as shown in Figure 19:

- *Establishing the Connection*: Server socket object is initialized and inside a while loop a socket object continuously accepts an incoming connection.
- *Obtaining the Streams*: The `InputStream` object and `OutputStream` object is extracted from the current requests' socket object.

¹⁶ <https://github.com/google/leveldb/blob/master/doc/index.md>

- *Creating a handler object*: After obtaining the streams and port number (client Socket object), a new *ClientHandler* object (the class explained below) is created with these parameters.
- *Invoking the start() method*: The start() method is invoked on this newly created *Thread* object.

```

7.         try {
8.             // server is listening on port 1234
9.             server = new ServerSocket(1234);
10.            server.setReuseAddress(true);
11.            System.out.println("SERVER STARTED ON PORT " + server.getLocalPort() + " ..."
12.            + "\nReplicas: " + firstServerAddress + " " + secondServerAddress);
13.
14.            // running infinite loop for getting
15.            // client request
16.            while (true) {
17.
18.                // socket object to receive incoming client
19.                // requests
20.                Socket client = server.accept();
21.
22.                // create a new thread object
23.                ClientHandler clientSock
24.                    = new ClientHandler(client);
25.
26.                // This thread will handle the client
27.                // separately
28.                new Thread(clientSock).start();
29.            }
30.        }
31.        catch (IOException e) {

```

Figure 19. Part of function *main()* in *Server* class

The *ClientHandler* class implements *Runnable* interface and thus it can be passed as a target while creating a new *Thread*.

Inside the *run()* method of this class, the client message is read, and a handler function is called according to the type of the message.

For instance, an *add_cart* message will trigger the insertion of a product in the cart of the user, and a *get_cart* message will retrieve all the products in the cart of the user.

As mentioned above, all *ClientHandlers* share the same database instance. In particular, one of the variables of the *Server* class is an instance of the *LevelDB* class, containing the database object and all the queries that can be performed on it.

So basically, what a *ClientHandler* does upon receiving a message from a client is to call the corresponding query within the *LevelDB* class. The implementation of all queries on the database will be shown in detail in the related *Key-Value Database queries* section.

The *ClientHandler*, after handling the client request, also has the task of handling the replicas. In fact, if a write operation has been performed on the database, the *ClientHandler* must send messages to the replicas to update them.

To do this, similarly to what the *Server* does, the *ClientHandler* creates two new *ReplicaHandler* threads, one per replica.

ReplicaHandler is another class implementing *Runnable* whose sole purpose is to communicate with replicas.

Each *ReplicaHandler* sends an update message to one of the replicas and waits for an ACK. If the ACK is received, the connection with the replica is closed. Otherwise, another attempt is made, up to a maximum of three attempts.

For operations performed on the wishlist, the *ReplicaHandler* can simply forward to the replica the message the *ClientHandler* received. In fact, a product is either in a user's wishlist or not; this means that if the user were to notice any inconsistency, all he/she would have to do is request again the insertion or removal of a particular product.

As for the cart instead, since there is the *quantity* attribute, if some insertion/removal message were to be lost, the inconsistencies would no longer be repaired. Therefore, in this case the *ReplicaHandler* always send a message containing the updated quantity for the product.

2.4 CYBUY: MAIN PACKAGES AND CLASSES

The packages organization follows a hybrid approach: packages are first organized by layers, and then, within every layer, they are divided by feature.

As can be seen from Figure 20, the root package contains five children: *config*, *gui*, *middleware*, *model*, and *persistence*.

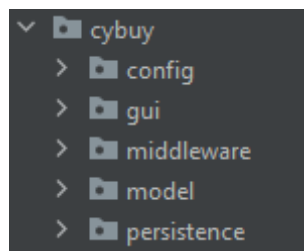


Figure 20. Package Structure

The *config* package contains the functions to load and read the configuration of the application (IPs of the servers).

The *model* package contains every class needed to represent in an OOP style every entity of the database: *User*, *Order*, *Product*, *Review*, *Analytics* and *DailyAnalytics*.

These classes contain only getter and setter methods to load and read data and all of them, except for the *Review* Object, are implemented following the *Builder Pattern*, since they have a lot of members that can or cannot be set at the initialization.

The other packages will be explained in detail below.

2.4.1 The gui package

The *gui* package contains the classes to handle the Graphical User Interface.

The implementation of the User Interface complies with the following logic. Every Interface is an implementation of an abstract class called *prototypes.GraphicInterface* which contains three methods:

- *initialize()*: a method that starts to load every graphical object and puts it into the scene tree.
- *getRoot()*: a getter method to retrieve the root of the scene tree.
- *setScene()*: a setter method that place the current Interface into the window.

Interfaces exchange information about the state of the application using another class called *Session*. This class contains the method to move from an Interface to another, and getter and setter methods for the information that is going to be shared between interfaces, such as the user logged. Every interface needs to be initialized giving this class as a parameter.

The implementation of the *GraphicInterface* follows two different approaches. Indeed, some interfaces have been implemented using the FXML approach, so every graphical data is contained by an XML-like file and the Interface simply loads the FXML file and its controller and then sets the parameters of the Interface using its controller.

Other interfaces, instead, have been implemented using a hard coded approach so the initialize function builds every object and moves it into its position.

We have chosen to use two different approaches since we have found some interfaces easier to implement with one method and other interfaces with the other method.

However, the project remains modular since every Interface is an implementation of an abstract Interface.

2.4.2 The persistence package

The *persistence* package contains the driver for MongoDB and the client side of the LevelDB functionalities.

The *mongoDBDriver* class contains as a member every query that has been implemented towards the MongoDB database. Analytics, as it was written before, are executed by a daemon on server side.

LevelDBClient is a class that offers a simple way to communicate with the servers that handle every operation performed on the LevelDB database.

The server and the client communicate using the *Message* class. This is a *Serializable* class with three attributes:

- the *type*, containing the name of the operation that needs to be executed;
- the *ID* of the *user* who is making the request;
- an *ArrayList* of *Products*.

These attributes are all the possible parameters needed by the LevelDB server to execute any query.

2.4.3 The middleware package

The *middleware* package contains the classes that are needed to connect the *Presentation Layer* (front end) with the drivers. So, it will connect the packages from *gui* to the ones inside the *persistence* folder.

The *EventHandler* class is a *Singleton* that is used to communicate between the GUI and the database drivers. It has a lot of methods that simply call the methods of the driver changing its parameters. It therefore works like an *Adapter* pattern.

The classes *Utilities* and *PasswordUtils* contain utility functions used by the GUI and the drivers, such as the control of images URLs and the hashing for the passwords.

2.5 MOST RELEVANT QUERIES

2.5.1 MongoDB queries

2.5.1.1 Insert a New Product

This function (Figure 21) is used to insert a new product into the database.

```

1. public String insertProduct(Product newProduct) {
2.     Document productDocument = ProductToDocument(newProduct);
3.     ClientSession clientSession = mongoClient.startSession();
4.     TransactionBody<String> txnFunc = () -> {
5.         BsonValue id = productsCollection.insertOne(clientSession, productDocument).getInsertedId();
6.         if (id == null) {
7.             return "ERROR inserting the product";
8.         }
9.         String productId = id.asObjectId().getValue().toHexString();
10.        newProduct.setObjectId(productId);
11.        long modifiedCount = usersCollection.updateOne(clientSession, eq("username", newProduct.getSellerUsername()),
12.            addToSet("products_on_sale", newProduct.getObjectId())).getModifiedCount();
13.        if (modifiedCount == 0) {
14.            return "ERROR inserting the product in the seller's document";
15.        }
16.        return "OK";
17.    };
18.    if (executeTransaction(clientSession, txnFunc)) {
19.        return newProduct.getObjectId();
20.    }
21.    return "ERROR";
22. }

```

Figure 21. Add Product Query

As already mentioned, the *Product* class stores all the details about a product, such as *description*, *price*, *image URL* and other useful information.

A *Product* instance is passed to the function, which then converts it into a MongoDB *Document* and simply inserts this product into the *Products* collection.

This new product is inserted by a *Seller*, so it is important to insert it in the seller's document in the database as well.

In order to do so, we simply update the *product_on_sale* array in the seller's document within the *Users* collection.

2.5.1.2 Modify an existing Product

```

1.  public boolean modifyProduct(String productId, Product newProduct){
2.      Bson query = eq("_id", new ObjectId(productId));
3.      Bson update = Updates.combine(
4.          Updates.set("image", newProduct.getImage()),
5.          Updates.set("description", newProduct.getDescription()),
6.          Updates.set("quantity_available", newProduct.getQuantityAvailable()),
7.          Updates.set("price", newProduct.getPrice()),
8.          Updates.set("details", newProduct.getDetails())
9.      );
10.
11.     UpdateResult response = productsCollection.updateOne(query, update);
12.     return response.getModifiedCount() == 1;
13. }

```

Figure 22. Modify Product Query

The function shown in Figure 22 is used to modify an existing product, by changing some of the features the seller had previously added.

The *Seller* can change this information:

- *description*,
- *price*,
- *image URL*,
- features about the product (*details*).

The function searches for the product document in the *Products* collection using the product ID and then changes all the fields that the seller specified in the application.

Note that this information was stored into the *newProduct* instance passed as argument in the function.

2.5.1.3 Delete a Product

Figure 23 shows the function that deletes a seller's product. This function takes as parameters the *Product* instance of the product to be deleted and the *User* instance of the seller that owns it.

In order to delete the product, we must delete its document from the *Products* collection, and its ID from the *products_on_sales* array of the seller's document in the *Users* collection.

Since the function needs to access multiple documents, all operations are included in a MongoDB Transaction, in order to ensure atomicity.

```

1. public boolean deleteProduct(User user, Product product) {
2.     if (product.getObjectId() == null ||
3.         !user.getUsername().equals(product.getSellerUsername())) return false;
4.     ClientSession clientSession = mongoClient.startSession();
5.     TransactionBody<String> txnFunc = () -> {
6.         long deletedCount = productsCollection.deleteOne(clientSession, eq("_id",
7.             new ObjectId(product.getObjectId()))).getDeletedCount();
8.         if (deletedCount == 0) {
9.             return "ERROR: There is no such product";
10.        }
11.        long modifiedCount = usersCollection.updateOne(clientSession, eq("username", product.getSellerUsername()),
12.            pull("products_on_sale", product.getObjectId())).getModifiedCount();
13.        if (modifiedCount == 0) {
14.            return "ERROR deleting the product from seller's document";
15.        }
16.        return "OK";
17.    };
18.    return executeTransaction(clientSession, txnFunc);
19. }

```

Figure 23. Delete Product Query

2.5.1.4 Get a list of Products based on specific parameters

```

1. public ArrayList<Product> getProductsList(String keyword, String productType, String platform, int skip, int limit, int sort, String
2.     seller_username) {
3.     final ArrayList<Product> productList = new ArrayList<>();
4.
5.     FindIterable<Document> productsDocuments;
6.
7.     Bson findByDescription = text("\"+keyword+"");
8.
9.     Bson query;
10.    if (keyword != null && !keyword.equals("")) {
11.        if (productType != null) {
12.            if (platform != null) {
13.                if (seller_username != null) {
14.                    query = and(eq("product_type", productType), eq("seller_username", seller_username), eq("platform", platform),
15.                        findByDescription);
16.                } else {
17.                    query = (and(ne("quantity_available", 0), eq("product_type", productType), eq("platform", platform),
18.                        findByDescription));
19.                }
20.            } else {
21.                if (seller_username != null) {
22.                    query = (and(eq("product_type", productType), eq("seller_username", seller_username), findByDescription));
23.                } else {
24.                    query = (and(ne("quantity_available", 0), eq("product_type", productType), findByDescription));
25.                }
26.            }
27.        }
28.    }
29. }

```

Figure 24. Get Product List Query¹⁷

This function (Figure 24) is used to retrieve a list of products from the database based on certain parameters.

The application will use this function to display a list of products, filtered and/or sorted according to the user's choices.

The function takes these arguments:

- *keyword*: only the products with a description matching the keyword will be retrieved.

¹⁷ The full code is displayed on the GitHub Repository or at this link below:
<https://pastebin.com/8ksxaCz9>

- *productType*: only the products belonging to this specific category will be retrieved.
- *platform*: the subcategory is specified too.
- *skip*: the first N items will be skipped and not retrieved.
- *limit*: only N items will be retrieved (usually only 12 products are displayed in a page).
- *sort*: an integer that corresponds to a specific sort based on price or review.
- *seller_username*: only the products of a target seller will be retrieved.

The *keyword* parameter is set if the user uses the search bar to look for a specific product. The *productType* parameter is set only if a category was previously clicked (the same happen also for the subcategory *platform*).

At last, the *sort* parameter is set if the user selects it from a menu in the application. Indeed, the user can sort the list of products by Price from High to Low (integer 1), by Price from Low to High (integer 2), by Stars from High to Low (integer 3) and by Stars from Low to High (integer 4).

The skip and limit integers are automatically set by the application in order to retrieve the right list of products to be displayed in a specific page.

2.5.1.5 Get the number of Products based on specific parameters

The function in Figure 25 is used to retrieve the number of products in the application that match specific criteria. The arguments specified in the function are very similar to the *getProductsList* query.

Indeed, it is possible to retrieve the number of products that belong to a specific category or subcategory, the number of products for sale from a specific seller, or the number of products whose description match the *keyword*.

```

1. public long getNumberOfProducts(String keyword, String productType, String platform, String seller_username) {
2.     long numberOfProducts;
3.
4.     Bson findByDescription = text("\\" + keyword + "\\");
5.     if(keyword != null)
6.         findByDescription = regex("description", keyword, "i");
7.
8.     if (keyword != null) {
9.         if (productType != null) {
10.            if (platform != null) {
11.                if (seller_username != null) {
12.                    numberOfProducts = productsCollection.countDocuments(and(eq("product_type", productType), eq("seller_username",
13.                        seller_username), findByDescription, eq("platform", platform)));
14.                } else {
15.                    numberOfProducts = productsCollection.countDocuments(and(eq("product_type", productType), findByDescription,
16.                        eq("platform", platform)));
17.                }
18.            } else {
19.                if (seller_username != null) {
20.                    numberOfProducts = productsCollection.countDocuments(and(eq("product_type", productType), eq("seller_username",
21.                        seller_username), findByDescription));
22.                } else {
23.                    numberOfProducts = productsCollection.countDocuments(and(eq("product_type", productType), findByDescription));
24.                }
25.            }
26.        } else {
27.            if (platform != null) {
28.                if (seller_username != null) {
29.                    numberOfProducts = productsCollection.countDocuments(and(findByDescription, eq("seller_username",
30.                        seller_username), eq("platform", platform)));

```

Figure 25. Get Number of Products Query¹⁸

¹⁸ The full code can be found on the GitHub Repository of the project

2.5.1.6 Get an instance of a Java Product of the from a product ID

Figure 26 shows a function that simply returns a *Product* instance for the product whose *ObjectId* is the one specified as an argument.

The function simply finds the product with that ID in the *Products* collection and then it converts the MongoDB Document into the Java *Product* class.

```
1. public Product getProductFromId(String id) {
2.     FindIterable<Document> productsDocuments = productsCollection.find(eq("_id", new ObjectId(id)));
3.     if (productsDocuments.first() == null) return null;
4.     return documentToProduct(productsDocuments.first());
5. }
```

Figure 26. Get Product from Id Query

2.5.1.7 Insert a new User

This function (Figure 26Figure 27) inserts a new user into the *Users* collection of MongoDB.

It takes an instance of a Java *User* with all the information regarding the user to be inserted and simply appends all the information into a MongoDB Document that will be added into the collection.

```
1. public String insertUser(User newUser) {
2.     Document user = new Document("name", newUser.getName())
3.         .append("surname", newUser.getSurname())
4.         .append("username", newUser.getUsername())
5.         .append("email", newUser.getEmail())
6.         .append("gender", newUser.getGender())
7.         .append("country", newUser.getCountry())
8.         .append("age", newUser.getAge())
9.         .append("role", newUser.getRole());
10.    Document password_doc = new Document();
11.    password_doc.append("password", newUser.getPassword());
12.    password_doc.append("salt", newUser.getSalt());
13.    user.append("password_info", password_doc);
14.    BsonValue id = usersCollection.insertOne(user).getInsertedId();
15.    if (id == null) return null;
16.    return id.asObjectId().getValue().toHexString();
17. }
```

Figure 27. Insert User Query

2.5.1.8 Delete an existing User

The function displayed in Figure 28 deletes an existing User from the application.

If the User to be deleted is a *Standard User*, the function simply deletes its document from the *Users* collection because there is no need to delete any active or past orders.

Otherwise, if the User is a *Seller* or an *Administrator*, in addition to deleting the user's document from the *Users* collection, all of the user's products for sale must be deleted, all

active orders must be cancelled, and documents regarding the user withing the *Analytics* collection must be deleted.

Once again, all these operations are included in a MongoDB Transaction to ensure atomicity.

```

1. public boolean deleteUser(User user) {
2.     boolean success;
3.
4.     ClientSession clientSession = mongoClient.startSession();
5.
6.     if(user.getRole().equals("User")){
7.         success = usersCollection.deleteOne(eq("_id", new ObjectId(user.getObjectId()))).wasAcknowledged();
8.     } else { //is a Seller or an Admin
9.         ArrayList<String> seller_products = user.getProductsOnSale();
10.        ArrayList<Order> seller_orders = user.getOrders();
11.        TransactionBody<String> txnFunc = () -> {
12.            for (String seller_product : seller_products) { //delete all the products of a seller
13.                Product product = Product.ProductBuilder().setObjectId(seller_product)
14.                    .setSellerUsername(user.getUsername()).build();
15.                boolean deleted_product = deleteProduct(user, product);
16.                if(!deleted_product) {
17.                    return "Error deleting the product";
18.                }
19.            }
20.            for (Order seller_order : seller_orders) { //delete all active orders
21.                boolean deleted_order = deleteOrder(seller_order, true);
22.                if(!deleted_order){
23.                    return "Error deleting the order";
24.                }
25.            }
26.            //Delete analytics document
27.            boolean deleted_analytics = analyticsCollection.deleteMany(eq("seller_username",
28.                user.getUsername())).wasAcknowledged();
29.            if(!deleted_analytics){
30.                return "Error deleting the analytics of the user";
31.            }
32.            boolean deleted_user = usersCollection.deleteOne(eq("_id", new ObjectId(user.getObjectId()))).wasAcknowledged();
33.            if(!deleted_user){
34.                return "Error deleting the user";
35.            }
36.            return "OK";
37.        };
38.        success = executeTransaction(clientSession, txnFunc);
39.    }
40.    return success;
41. }

```

Figure 28. Delete User Query

2.5.1.9 Find a User

Figure 29 displays the function that finds the user with a specific *username* or with a specific *username* and *password* pair.

If we want to retrieve a document with only the username, the password field must be null, otherwise both the arguments of the function must be specified.

The password provided as an argument to the function is the one inserted by the user during the login phase, so it is not encrypted.

In order to verify if this password was the one that he inserted during the registration phase, we must encrypt it using the same encryption key of the registration phase (that was previously stored in the user's document) and check if there is a match between the current encrypted password and the one stored previously in the document.

```

1. public User findUser(String username, String password) {
2.     Document userDocument;
3.     if (username == null) { //we are not interested in finding all user with a specific password
4.         return null;
5.     } else { //Username not null
6.         if (password == null) { //we are interested in finding an user with this username
7.             userDocument = usersCollection.find(eq("username", username)).first();
8.             if(userDocument == null) return null;
9.             else return documentToUser(userDocument);
10.        } else { //login (i want to find exactly the document with the user and the password specified in the fields
11.            userDocument = usersCollection.find(eq("username", username)).first();
12.            if(userDocument == null) return null;
13.            else{
14.                Document password_doc = (Document) userDocument.get("password_info");
15.                String salt = password_doc.getString("salt");
16.                String encoded_password = password_doc.getString("password");
17.                boolean passwordMatch = PasswordUtils.verifyUserPassword(password, encoded_password, salt);
18.                if(!passwordMatch) return null;
19.                else return documentToUser(userDocument);
20.            }
21.        }
22.    }
23. }

```

Figure 29. Find User Query

```

1. public boolean insertReview(User user, Review review) {
2.     if (review.getProduct() == null || user == null ||
3.         !user.getRole().equals("User")) return false;
4.     String productId = review.getProduct();
5.     ObjectId userId = new ObjectId(user.getObjectId());
6.     ClientSession clientSession = mongoClient.startSession();
7.     TransactionBody<String> txnFunc = () -> {
8.         Document userDoc = usersCollection.find(clientSession, and(eq("_id", userId), eq("reviews.productId",
9.             productId))).projection(include("reviews.$")).first();
10.        if (userDoc == null) {
11.            return "ERROR: User cannot place the review, the product was not previously ordered";
12.        }
13.        ArrayList<Document> reviews = (ArrayList<Document>) userDoc.get("reviews");
14.        int previousStars = reviews.get(0).getInteger("stars");
15.        if (previousStars != -1) {
16.            return "ERROR: User cannot place the review, the product has already been reviewed";
17.        }
18.        long updateCount = usersCollection.updateOne(clientSession, and(eq("_id", userId), eq("reviews.productId", productId)),
19.            set("reviews.$stars", review.getStars())).getModifiedCount();
20.        if (updateCount == 0) {
21.            return "ERROR: no modification";
22.        }
23.        Bson query = eq("_id", new ObjectId(productId));
24.        FindIterable<Document> productsDocuments = productsCollection.find(clientSession, query);
25.        Document productDoc = productsDocuments.first();
26.        if (productDoc == null) return "ERROR: Non-existing product";
27.        int totalReviews = productDoc.getInteger("total_reviews");
28.        double averageReview = productDoc.getDouble("average_review");
29.        int newTotal = totalReviews + 1;
30.        double newAverage = ((averageReview * totalReviews) + review.getStars()) / newTotal;
31.        long modifiedCount = productsCollection.updateOne(clientSession, query, set("total_reviews", newTotal)).getModifiedCount();
32.        if (modifiedCount == 0)
33.            return "ERROR updating total reviews";
34.        boolean modified = productsCollection.updateOne(clientSession, query, set("average_review", newAverage)).wasAcknowledged();
35.        if (!modified)
36.            return "ERROR updating average review";
37.        return "OK";
38.    };
39.    return executeTransaction(clientSession, txnFunc);
40. }

```

Figure 30. Insert Review Query

2.5.1.10 Insert a Review

This function (Figure 30) is used to insert a review from a specific user to a product that he previously purchased.

When a *standard user* has bought a product that has been successfully delivered to him/her, the user can place a review 1-to-5-star review for that product.

The function checks that the user has already bought the product; in this case in the reviews array of his/her document there is already a document for the said product.

If the number of stars is set to -1 the user can place a review and his mark will replace it, otherwise if the integer is not -1 it means that the user has already placed a review to that product before.

All the operations are once again included in a MongoDB Transaction to ensure their atomicity.

2.5.1.11 Insert an Order

```

1. public boolean insertOrder(Order order) {
2.     Pair<Bson, Bson> insertUser, insertSeller;
3.     order.setObjectId(ObjectId.get().toString());
4.     //retrieve the query for the user
5.     insertUser = insertOrderUser(order);
6.     //retrieve the query for the seller
7.     insertSeller = insertOrderSeller(order);
8.     ClientSession clientSession = mongoClient.startSession();
9.     //Transaction should guarantee ACID properties (in this case it's necessary)
10.    TransactionBody<String> txnFunc = () -> {
11.        //insert order into the user and the seller
12.        UpdateResult response;
13.        response = usersCollection.updateOne(clientSession, insertUser.getKey(), insertUser.getValue());
14.        if (response.getModifiedCount() == 0) {
15.            clientSession.abortTransaction();
16.            return "ERROR there is no such client";
17.        }
18.        response = usersCollection.updateOne(clientSession, insertSeller.getKey(), insertSeller.getValue());
19.        if (response.getModifiedCount() == 0) {
20.            clientSession.abortTransaction();
21.            return "ERROR there is no such seller";
22.        }
23.        for (Order.ProductOrder product : order.getProductOrders()) {
24.            Bson query = eq("_id", new ObjectId(product.getProductID()));
25.            //get product document to control availability
26.            Document productDocument = productsCollection.find(clientSession, query).first();
27.            //update the product quantity sold member
28.            increment = new Document()
29.                .append("quantity_sold", product.getOrdered_quantity())

```

Figure 31. Insert Order Query¹⁹

This function (Figure 31) inserts an order into the database; at the moment of the insertion, it is an active order.

¹⁹ The full function can be found in the GitHub Repository of the project

More specifically, the function will insert the order into the document of the user who placed it and also into the seller's document within the *Users* collection.

All the operations are included in a MongoDB Transaction.

It is important to notice that when users checkout their cart, the application will take care of splitting the entire order into smaller orders, each regarding a single seller.

2.5.1.12 Delete an Order

Figure 32 displays the function that deletes an active order from both the document of the user who placed it and the document of seller.

This function takes the Java *Order* instance and a boolean called *restoreAvailability* that, if set as true, will restore the availability of that product as if the product had never been purchased before.

```

1. public boolean deleteOrder(Order order, boolean restoreAvailability) {
2.     //retrieve IDs of user seller and order
3.     ObjectId orderId;
4.     String user_username = order.getUser_username();
5.     String seller_username = order.getSeller_username();
6.     orderId = new ObjectId(order.getObjectId());
7.     ClientSession clientSession = mongoClient.startSession();
8.     //this transaction should be ACID
9.     TransactionBody<String> txn = () -> {
10.         Bson query, update;
11.         update = Updates.pull("orders", new Document("_id", orderId));
12.         query = or(
13.             eq("username", user_username),
14.             eq("username", seller_username)
15.         );
16.         UpdateResult response = usersCollection.updateMany(clientSession, query, update);
17.         if (response.getModifiedCount() != 2) {
18.             clientSession.abortTransaction();
19.             return "ERROR";
20.         }
21.         return "OK";
22.     };
23.     //Launch transaction and wait for response
24.     String transactionOK;
25.     try {
26.         transactionOK = clientSession.withTransaction(txn);
27.     } catch (Exception e) {
28.         System.out.println("Error");

```

Figure 32. Delete Order Query²⁰

²⁰ The full function can be found on the GitHub repository of the project

2.5.1.13 Change the order state of an existing Order

This function (Figure 33) is used to change the order state of a specific order.

If the order is changed to “Arrived”, it must be moved to the Orders collection and it must be removed from the documents of the seller and of the user that placed it.

The order state can be *pending* (it is the state of a new order), *sent*, *delivered*, *arrived*, or *cancelled*.

```

1. public boolean changeOrderState(Order order, OrderState newState) {
2.     //change order state
3.     changeOrderStateKernel(order, newState);
4.     //check if the order needs to be inserted into the order collection
5.     if (newState == OrderState.arrived) {
6.         deleteOrder(order, false);
7.         order.setDeliveryDate(new Date());
8.         insertOrderCollection(order);
9.         //time to check if exists an user review for that product if not create a review object
10.        //check for every product
11.        String user_username = order.getUser_username();
12.        for (Order.ProductOrder product : order.getProductOrders()) {
13.            Document response;
14.            Bson query = and(
15.                eq("username", user_username),
16.                eq("reviews.productId", product.getProductID())
17.            );
18.            response = usersCollection.find(query).first();
19.            //if review not exist create a new empty review
20.            if (response == null) {
21.                Document emptyReview = new Document()
22.                    .append("productId", product.getProductID())
23.                    .append("stars", -1);
24.                usersCollection.updateOne(eq("username", user_username), Updates.addToSet("reviews", emptyReview));
25.            }
26.        }
27.    }
28.    return true;
29. }
30. }
```

Figure 33. Change Order State Query

```

1. public Analytics getSellerAnalytics(User user) {
2.     Document analyticsDocument;
3.     Analytics newAnalytics = Analytics.AnalyticsBuilder();
4.     if (user == null || user.getRole().equals("User") || user.getUsername() == null)
5.         return newAnalytics;
6.     String seller_username = user.getUsername();
7.     analyticsDocument = analyticsCollection.find(eq("seller_username", seller_username)).first();
8.     if (analyticsDocument != null) {
9.         newAnalytics = convertToAnalytics(analyticsDocument);
10.    }
11.    return newAnalytics;
12. }
13. }
```

Figure 34. Get Seller Analytics Query

2.5.1.14 Get the analytics of a target Seller

The function shown in Figure 34 retrieves the analytics of the specific seller passed as an argument.

It simply finds the analytics document into the *Analytics* collection of that specific user and then it uses a conversion function to convert the document to an instance of the Java *Analytics* class.

2.5.1.15 Get a list of Sellers based on their Ranking

Figure 35 shows the function that returns an ArrayList of users based on specific parameters. The function takes a *username* String and an integer that corresponds to a particular sort.

If the *username* String is specified, only users whose username matches the string will be inserted into the list. Otherwise, all the users will be retrieved.

If the *sort* integer is set to 0, the users will be retrieved in descending order of average review score. If it is set to 1, instead, the users will be retrieved in ascending order.

```

1. public ArrayList<User> getSellerRanking(String username, int sorting) {
2.     ArrayList<User> sellers = new ArrayList<>();
3.     FindIterable<Document> sellersDocs;
4.     if(username != null) {
5.         sellersDocs = analyticsCollection.find(regex("seller_username", username));
6.     } else {
7.         sellersDocs = analyticsCollection.find();
8.     }
9.     if(sorting == 1) {
10.        sellersDocs = sellersDocs.sort(ascending("avg_reviews"));
11.    } else {
12.        sellersDocs = sellersDocs.sort(descending("avg_reviews"));
13.    }
14.    sellersDocs.limit(10);
15.    for(Document doc : sellersDocs) {
16.        String seller_username = doc.getString("seller_username");
17.        double avg_reviews = doc.getDouble("avg_reviews");
18.        int total_reviews = doc.getInteger("total_reviews");
19.        User seller = User.UserBuilder().setUsername(seller_username)
20.                                         .setAverageReview(avg_reviews)
21.                                         .setTotalReviews(total_reviews).build();
22.        sellers.add(seller);
23.    }
24.    return sellers;
25. }

```

Figure 35. Get Seller List By Ranking Query

2.5.2 MongoDB aggregations

This section will show all the aggregations created for MongoDB. For each aggregation the pipeline code will be displayed; to see how we exported each pipeline to Java language please refer to the code of the *daemon* module.

2.5.2.1 Total Reviews and Average Review

In order to retrieve the total number of reviews and the average review for a seller an aggregation is performed on the *Products* collection.

This aggregation takes advantage of the fact that redundancies regarding the total number of reviews and the average review for a product have been included in the document of that product.

The pipeline for this aggregation is shown in Figure 36, where the username "Lee4an" is shown as an example. Obviously, in the Java application the seller username is a parameter for the function that will perform the aggregation.

Special care had to be taken when calculating the average review. In fact, without any check, the calculation of this analytics on a user who has not yet received any review would have generated a division by 0.

```

1  [
2  {
3      $match: {
4          seller_username: "Lee4an"
5      }
6  },
7  {
8      $set: {
9          sum_of_reviews: {
10             $multiply: [
11                 "$total_reviews",
12                 "$average_review"
13             ]
14         }
15     }
16 },
17 {
18     $group: {
19         _id: "$seller_username",
20         total_seller_reviews: {
21             $sum: "$total_reviews"
22         },
23         sum_of_seller_reviews: {
24             $sum: "$sum_of_reviews"
25         }
26     }
27 },
28 {
29     $set: {
30         average_seller_review: {
31             $cond: [
32                 {
33                     $eq: [
34                         "$total_seller_reviews",
35                         0
36                     ]
37                 },
38                 0,
39                 {
40                     $divide: [
41                         "$sum_of_seller_reviews",
42                         "$total_seller_reviews"
43                     ]
44                 }
45             ]
46         }
47     }
48 }
49 ]

```

Figure 36. Aggregation that retrieves the total number of reviews and the average review of a seller

2.5.2.2 Stars distribution

In order to retrieve the distribution of stars (reviews) for every product on sale for a specific seller an analytic is performed on the *Order* Collection (see Figure 37).

This job could also be accomplished by adding more attributes to the products collection, but this could also increase the size of the collection and not permitting a rapid change of paradigm in case of a possible update of the system (poor flexibility).

The analytic starts by searching in the *Orders* collection and filters every document that does not involve the seller, then group every client that has bought something.

A lookup is needed in order to retrieve the reviews of the clients. The lookup is very fast because it takes advantage of the username index.

Then some cleaning of the result document is mandatory and then group up every review that the seller has taken by the quantity of stars and count them to obtain the distribution.

```

1  [
2    {
3      $match: {
4        seller_username: "Lee4an"
5      }
6    },
7    {
8      $group: {
9        _id: "$user.username",
10       orderedProduct: {
11         $push: "$orderedProduct"
12       }
13     }
14   },
15   {
16     $lookup: {
17       from: "users",
18       localField: "_id",
19       foreignField: "username",
20       as: "user"
21     }
22   },
23   {
24     $project: {
25       reviews : {$arrayElemAt : ["$user.reviews", 0]},
26       orderedProduct : "$orderedProduct"
27     }
28   },
29   {
30     $unwind: {
31       path: "$reviews"
32     }
33   },
34   {
35     $unwind: {
36       path: "$orderedProduct"
37     }
38   },
39   {
40     $unwind: {
41       path: "$orderedProduct"
42     }
43   },
44   {
45     $project: {
46       reviews: 1,
47       orderedProduct: 1,
48       compare: {
49         $cmp: ["$reviews.productId", "$orderedProduct.productId"]
50       }
51     }
52   },
53   {
54     $match: {
55       compare : 0
56     }
57   },
58   {
59     $group: {
60       _id: "$reviews.stars",
61       fieldN: {
62         $sum: 1
63       }
64     }
65   }
66 ]

```

Figure 37. The analytic to retrieve the distribution of the Reviews

2.5.2.3 Average Delivery Time

In order to obtain the Average Delivery Time of a seller an aggregation is performed on the *Orders* collection.

This is a simple aggregation that calculate the average of the difference between the *delivery_date* and the *order_date* of a document and take the average.

The aggregation for the user "Lee4an" is shown as an example in Figure 38.

```

1  [
2    {
3      $match: {
4        seller_username: "Lee4an"
5      }
6    },
7    {
8      $project: {
9        seller_username: 1,
10       delivery_duration: {
11         $subtract: ["$delivery_date", "$order_date"]
12       }
13     }
14   },
15   {
16     $group: {
17       id: "$seller_username",
18       average_duration: {
19         $avg: "$delivery_duration"
20       }
21     }
22   }
23 ]

```

Figure 38. The analytic to calculate the average delivery duration

2.5.2.4 Total Earnings

In order to retrieve the total earnings of a seller an aggregation is performed on the *Products* collection.

This aggregation exploits the fact that redundancies regarding the number of units sold have been included in product documents.

The pipeline for this aggregation is shown in Figure 39, where the username "Lee4an" is shown as an example. Once again, in the Java application the seller username is a parameter for the function that will perform the aggregation.

```

1  [
2    {
3      $match: {
4        seller_username: "Lee4an"
5      }
6    },
7    {
8      $set: {
9        product_earnings: {
10         $multiply: [
11           "$quantity_sold",
12           "$price"
13         ]
14       }
15     }
16   },
17   {
18     $group: {
19       id: "$seller_username",
20       total_earnings: {
21         $sum: "$product_earnings"
22       }
23     }
24   }
25 ]

```

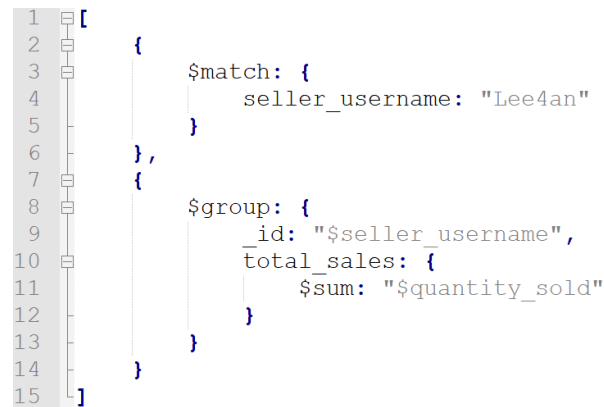
Figure 39. Aggregation that retrieves the total earnings of a seller

2.5.2.5 Total Sales

In order to retrieve the Total Sales of a seller an aggregation on the *Products* collection is needed.

We simply do a match on the `seller_username` specifying the username of the seller that we want to retrieve the total sales for and then we group all the documents summing all the `quantity_sold` fields.

This pipeline is shown in Figure 40, "Lee4an" is shown as an example. In Java, the `seller_username` is a parameter for the function that will perform this aggregation.



```

1  [
2    {
3      '$match': {
4        'seller_username': 'Lee4an'
5      },
6    },
7    {
8      '$group': {
9        '_id': '$seller_username',
10       'total_sales': {
11         '$sum': '$quantity_sold'
12       }
13     }
14   }
15 ]

```

Figure 40. Aggregation that retrieves the total number of sales for a seller

2.5.2.6 Daily Sales Analytics

As previously mentioned, whenever an order arrives at the customer, the document containing the order overview is placed in the *Orders* collection.

The documents in this collection will also contain personal information regarding customers, such as their age and gender.

It can be important for a seller to be able to view the daily performance of their sales, not only in a general way but also by dividing customers into groups: *males* and *females*, *young* and *old*²¹.

For this purpose, an aggregation, whose pipeline is shown in Figure 41, is performed on the *Orders* collection.

In the Figure, the username "Lee4an" and the date 22/01/2021 are shown as examples, but these are parameters in the function of the Java application that will perform the aggregation.

The result obtained will be a triplet containing the number of total sales, the number of total purchases made by male customers, and the number of total purchases made by old customers, for that day.

²¹ In our analysis, customers fall into the *young* category if their age is less than or equal to 35.

Note that the number of total purchases made by female customers and the number of total purchases made by young customers can be easily calculated from the previous result; this calculation will be done in the Java application function that will perform this aggregation.

```

1  [
2  {
3      $match: {
4          seller_username: "Lee4an"
5      }
6  },
7  {
8      $project: {
9          seller_username: 1,
10         user: 1,
11         year: {
12             $year: "$delivery_date"
13         },
14         month: {
15             $month: "$delivery_date"
16         },
17         day: {
18             $dayOfMonth: "$delivery_date"
19         }
20     }
21 },
22 {
23     $match: {
24         day: 22,
25         month: 1,
26         year: 2021
27     }
28 },
29 {
30     $group: {
31         _id: "$seller_username",
32         male_sales: {
33             $sum: {
34                 $cond: [
35                     {
36                         $eq: [
37                             "$user.gender",
38                             "Male"
39                         ],
40                         1,
41                         0
42                     }
43                 ]
44             },
45             old_sales: {
46                 $sum: {
47                     $cond: [
48                         {
49                             $gt: [
50                                 "$user.age",
51                                 35
52                             ],
53                             1,
54                             0
55                         }
56                     ]
57                 }
58             }
59         }
60     }
61 ]

```

Figure 41. Aggregation that retrieves the daily analytics for a seller

2.5.3 Key-Value Database queries

2.5.3.1 Insert Product in Cart

The function shown in Figure 42 inserts the product passed as parameter in the cart of the user whose ID is passed as parameter.

First of all, the function checks if the item is already present in the cart. If so, it increases the attribute value and it returns the updated quantity. Otherwise, it inserts the product with a quantity of 1 and calls *insertProductInfos()*.

This last function, whose implementation we will see later, deals with redundancies on product description, price, and image.

```

1. public int addProductToCart(String user_id, Product product){
2.     //open db
3.     if(!dbOpen) openDB();
4.
5.     //If the product is already present in the cart, its quantity needs to be increased
6.     String product_id = product.getObjectId();
7.
8.     String key = "cart:user:" + user_id + ":product:" + product_id + ":quantity";
9.     String quantity = getValue(key);
10.
11.    if(quantity == null){
12.        putValue(key,"1"); //item is not in the cart -> added the product with quantity = 1
13.        // I also need to handle the redundancies
14.        if(insertProductInfos(product))
15.            return 1;
16.        return 0;
17.    } else { // item already in the cart, update its quantity
18.        int quantity_int = Integer.parseInt(quantity);
19.        quantity_int += 1;
20.        String new_quantity_string = "" + quantity_int;
21.        putValue(key,new_quantity_string);
22.        return quantity_int;
23.    }
24. }

```

Figure 42. Insert Product in Cart

2.5.3.2 Remove Product from Cart

The function shown in Figure 43 removes the product passed as parameter from the cart of the user whose ID is passed as parameter.

First of all, the function checks if the item is already present in the cart. If so, it decreases the attribute value and it returns the updated quantity.

If the updated quantity is 0, the entire record is removed from the database and function *decrementUsage()* is called.

This function decrements the value of the usage attribute for the product passed as parameter. As discussed earlier in the *Data Model* section, the usage attribute is used to handle redundancies. In fact, it indicates how many carts and wishlists currently need the information about a particular product and, when it reaches 0, it means that redundancies can be eliminated.

```

1.     public int removeProductCart(String user_id, Product product){
2.         //open db
3.         if(!dbOpen) openDB();
4.
5.         //Check if the product is already present in the cart -> i need to decrease the quantity
6.         String product_id = product.getObjectId();
7.         String key = "cart:user:" + user_id + ":product:" + product_id +":quantity";
8.         String quantity = getValue(key);
9.
10.        if(quantity == null){
11.            //nothing happen there is no such items (maybe something is wrong?)
12.            return -1;
13.        }else{
14.            int quantity_int = Integer.parseInt(quantity);
15.            quantity_int--;
16.
17.            //insert the updated value iff the quantity is greater than 0
18.            if(quantity_int > 0)
19.                putValue(key, String.valueOf(quantity_int));
20.            else {
21.                decrementUsage(product_id);
22.
23.                //remove the previous value
24.                deleteValue(key);
25.            }
26.            return quantity_int;
27.        }
28.    }

```

Figure 43. Remove Product from Cart

```

1.     public ArrayList<Product> getProductsInCart(String user_id) throws IOException {
2.         ArrayList<Product> products = new ArrayList<>();
3.
4.         //open db
5.         if(!dbOpen) openDB();
6.
7.         try (DBIterator iterator = db.iterator()) {
8.             for (iterator.seekToFirst(); iterator.hasNext(); iterator.next()) {
9.                 String key = asString(iterator.peekNext().getKey());
10.
11.                // check if it is a cart record for the user passed as a parameter
12.                if (key.startsWith("cart:")) {
13.                    String[] parts = key.split(":");
14.                    if (parts[2].equals(user_id)) {
15.                        Product newProduct = getProductInfo(parts[4]);
16.
17.                        // Since the quantityAvailable parameter of Product is unused, I can use it as a parameter for the quantity
18.                        newProduct.setQuantityAvailable(Integer.parseInt(getValue(key)));
19.                        products.add(newProduct);
20.                    }
21.                }
22.            }
23.        }
24.
25.        return products;
26.    }

```

Figure 44. Retrieve all Products in Cart

2.5.3.3 Retrieve all Products in Cart

Figure 44 displays the function that retrieves all the products present in the shopping cart of the user passed as parameter.

The function must scan the entire database looking for records regarding the shopping cart of that particular user. When a record is found, the redundancies for the product are retrieved and the said product is inserted into an ArrayList.

Finally, the ArrayList containing all the products is returned.

2.5.3.4 Add Product to Wishlist

The function shown in Figure 45 inserts the product passed as parameter in the wishlist of the user whose ID is passed as parameter.

First of all, the function checks if the item is already present in the wishlist. If so, no action should be performed. Otherwise, it inserts the product with the current date as attribute and calls *insertProductInfos()*.

```

1.      public boolean addProductWishlist(String user_id, Product product){
2.          DateTimeFormatter dtf = DateTimeFormatter.ofPattern("dd/MM/yyyy");
3.          LocalDateTime now = LocalDateTime.now();
4.          String current_date = dtf.format(now); //format dd/MM/yyyy
5.
6.          //open db
7.          if(!dbOpen) openDB();
8.
9.          //check if the product is already present in the wishlist
10.         String product_id = product.getObjectId();
11.
12.         String key = "wishlist:user:" + user_id + ":product:" + product_id + ":date";
13.         String date = getValue(key);
14.
15.         if(date == null){ //item is not in the wishlist -> add a product with the current date
16.             putValue(key, current_date);
17.             // I also need to handle the redundancies
18.             insertProductInfos(product);
19.         } else { //item already in the wishlist
20.             return true;
21.         }
22.         return true;
23.     }

```

Figure 45. Add Product to Wishlist

2.5.3.5 Remove Product from Wishlist

Figure 46 displays the function that removes the product passed as parameter from the wishlist of the user whose ID is passed as parameter.

First of all, the function checks if the item is already present in the wishlist. If so, it deletes the corresponding record and it calls *decrementUsage()* function.

```

1.     public boolean removeProductWishlist(String user_id, Product product){
2.
3.         //open db
4.         if(!dbOpen) openDB();
5.
6.         String product_id = product.getObjectId();
7.         String key = "wishlist:user:" + user_id + ":product:" + product_id + ":date";
8.
9.         String date = getValue(key);
10.
11.        if(date != null) {
12.            decrementUsage(product_id);
13.            deleteValue(key);
14.        }
15.
16.        return true;
17.    }

```

Figure 46. Remove Product from Wishlist

```

1.     public ArrayList<Product> getProductsInWishlist(String user_id) throws IOException {
2.         ArrayList<Product> products = new ArrayList<>();
3.
4.         //open db
5.         if(!dbOpen) openDB();
6.
7.         try (DBIterator iterator = db.iterator()) {
8.             for (iterator.seekToFirst(); iterator.hasNext(); iterator.next()) {
9.                 String key = asString(iterator.peekNext().getKey());
10.
11.                 // check if it is a wishlist record for the user passed as a parameter
12.                 if (key.startsWith("wishlist:")) {
13.                     String[] parts = key.split(":");
14.                     if (parts[2].equals(user_id)) {
15.                         Product newProduct = getProductInfo(parts[4]);
16.                         products.add(newProduct);
17.                     }
18.                 }
19.             }
20.         }
21.         return products;
22.     }

```

Figure 47. Retrieve all Products in Wishlist

2.5.3.6 Retrieve all Products in Wishlist

The function shown in Figure 47 that retrieves all the products present in the wishlist of the user passed as parameter.

Similarly, to *getProductsInCart()*, the function must scan the entire database looking for records regarding the wishlist of that particular user. When a record is found, the redundancies for the product are retrieved and the said product is inserted into an ArrayList.

Finally, the ArrayList containing all the products is returned.

```

1.  public boolean insertProductInfos(Product product) {
2.      if(!dbOpen) openDB();
3.
4.      String id = product.getObjectId();
5.
6.      String usageKey = "product:" + id + ":usage";
7.      String usageValue = getValue(usageKey);
8.
9.
10.     if(usageValue == null) {    // The redundancy is not present yet
11.         String descriptionKey = "product:" + id + ":description";
12.         String priceKey = "product:" + id + ":price";
13.         String imageKey = "product:" + id + ":image";
14.
15.         String description = product.getDescription();
16.         String price = String.valueOf(product.getPrice());
17.         String image = product.getImage();
18.
19.         try (WriteBatch batch = db.createWriteBatch()) {
20.             batch.put(bytes(descriptionKey), bytes(description));
21.             batch.put(bytes(priceKey), bytes(price));
22.             batch.put(bytes(imageKey), bytes(image));
23.             batch.put(bytes(usageKey), bytes("1"));
24.             db.write(batch);
25.         } catch (IOException e) {
26.             e.printStackTrace();
27.             return false;
28.         }
29.     } else {    // The redundancy is already present
30.         incrementUsage(id);
31.     }
32.     return true;
33. }

```

Figure 48. Insert Product Redundancies

2.5.3.7 Insert Product Redundancies

This function, whose code is shown in Figure 48, takes care of inserting redundancies on description, image and price of the product passed as parameter.

First, it checks if redundancies are already present for the said product by checking the value of the usage attribute.

If the redundancies are not present, it proceeds with their insertion and sets the value of the usage attribute to 0. Otherwise, it simply increments the usage value by one by calling the function *incrementUsage()*.

2.5.3.8 Handling Product Redundancies

As already mentioned, the management of redundancies on product information takes place thanks to the usage attribute. The value of this attribute is rightly updated by calling the *incrementUsage()* and *decrementUsage()* functions.

The first one, whose code is shown in Figure 49, has the simple task of increasing by one the value of the usage for the product whose ID is passed as a parameter.

Figure 50 shows the code for the *decrementUsage()* function. This function must decrement by one the value of the usage attribute, but it also has the task of checking when that value reaches 0.

In fact, if the usage reaches 0, it means that that redundancy is no longer useful and can therefore be eliminated by calling the *cleanRedundancy()* function.

The latter is responsible for deleting the three records, the one for the description, the one for the image and the one for the price, for the product whose ID is passed as a parameter (Figure 51).

```

1.     private void incrementUsage(String product_id){
2.         String usage_key = "product:" + product_id + ":usage";
3.         int usage = Integer.parseInt(getValue(usage_key));
4.         usage += 1;
5.         putValue(usage_key,String.valueOf(usage));
6.     }
7.

```

Figure 49. Incrementing the *usage* value

```

1.     private void decrementUsage(String product_id){
2.         String usage_key = "product:" + product_id + ":usage";
3.         String usages = getValue(usage_key);
4.         int usage = 0;
5.
6.         if(usages != null) {
7.             usage = Integer.parseInt(usages);
8.             usage -= 1;
9.         }
10.
11.        if(usage > 0){
12.            putValue(usage_key,String.valueOf(usage));
13.        } else { // If the usage is 0 or below we can remove the redundancy
14.            cleanRedundancy(product_id);
15.        }
16.    }

```

Figure 50. Decrementing the *usage* value

```

1.     private void cleanRedundancy(String product_id) {
2.         String usageKey = "product:" + product_id + ":usage";
3.         String descriptionKey = "product:" + product_id + ":description";
4.         String priceKey = "product:" + product_id + ":price";
5.         String imageKey = "product:" + product_id + ":image";
6.
7.         ArrayList<String> keys = new ArrayList<>();
8.         keys.add(usageKey);
9.         keys.add(descriptionKey);
10.        keys.add(priceKey);
11.        keys.add(imageKey);
12.
13.        multipleDelete(keys);
14.    }

```

Figure 51. Cleaning Redundancies on a Product

3 TEST

3.1 STATISTICAL ANALYSIS

3.1.1 MongoDB Indexes

In order to enhance the execution of queries in MongoDB, *cybuy* makes use of indexes.

Without indexes, most queries in MongoDB should scan all documents within the collection to find the ones (or the one) that meet the query criteria.

This can significantly slow down the application's performance, especially considering a large and growing number of documents.

Indexes limit the number of documents to be inspected, but at the cost of memory usage and having to keep them up to date.

Given these considerations, our choice to introduce indexes was accompanied by a statistical analysis in terms of gain on performance.

The most important queries that we have chosen to improve in terms of performance are the queries that users will use the most:

- For a *standard user* the most important queries are to find a product by a substring, filter the products by their type or logging in the application.
- For a *seller* the most important queries are find every product that the user put up for sale, watch the analytics and logging in the application.

The number one priority in our application is the comfort for our users.

In order to speed up the log in, the most convenient thing to do is to create an index on the *username* attribute. As we can see from Figure 52 this index introduces an enormous gain in term of performance.

A Text index on the *description* of a product and an index on its *product_type* have been introduced in order to improve the performance of the filter query and research of the products by their description.

It is important to notice that a Text index requires more memory than an ordinary index and it is important to evaluate its size because the size of the indexes must be smaller than the RAM of the machine where the database works on.

In our case the Text index occupies about 5.4 Mb (at least 2 orders of magnitude smaller than the size of a RAM). The speed up of the Text index can be seen in Figure 52.

To improve the search of a product that a seller put up on sale it is required to create an index on the *seller_username* attribute on the *Products* collection.

This query is not important as the ones above, but we have noticed that this index also improves the performance of a lot of analytics. This is not mandatory because the analytics run only one time a day, but it is still appreciated.

As we can see from Figure 53 the analytics take advantage of this index a lot.

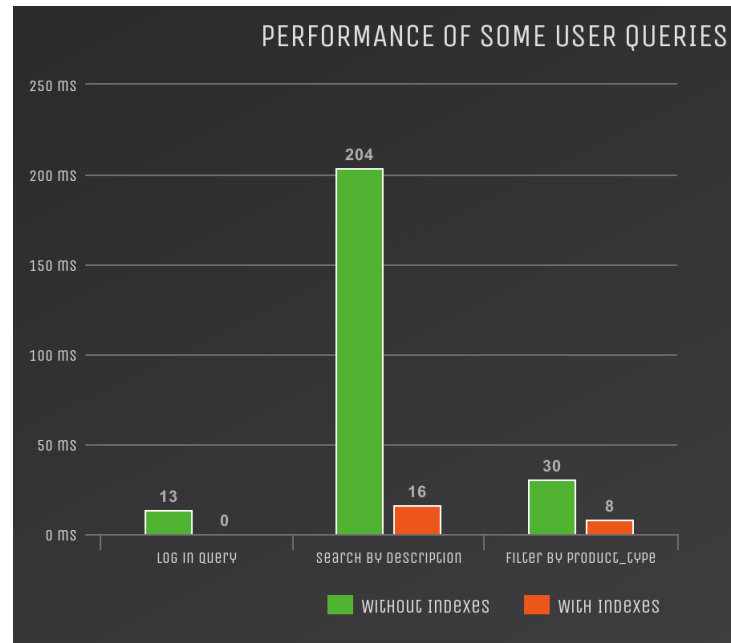


Figure 52. Performance improvement introduced by the indexes

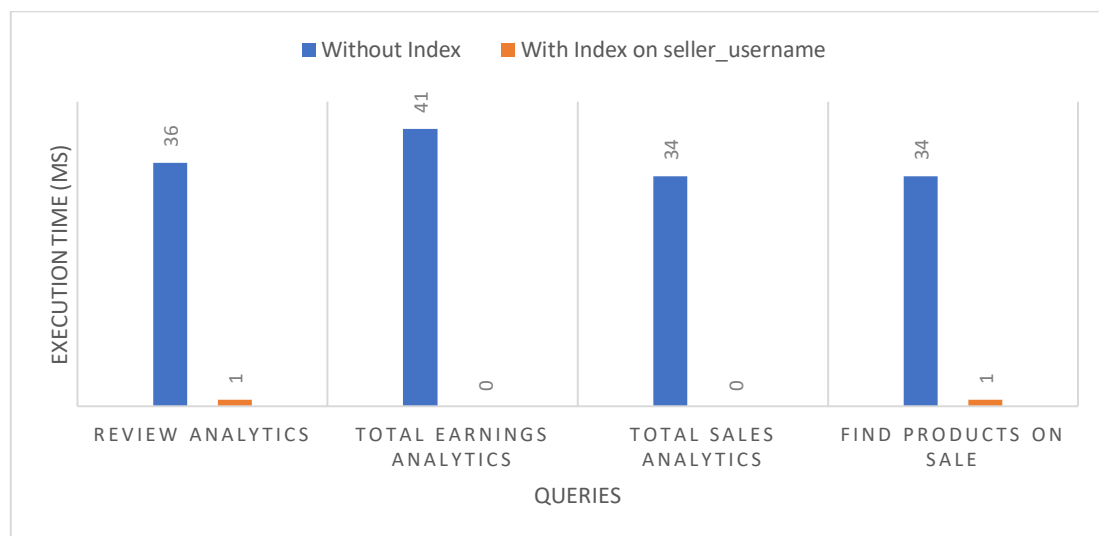


Figure 53. Queries performance with and without the index on seller_username in the *Products* collection

3.1.2 Brief Consideration about the CAP Theorem

As should be clear by this point in the documentation, all of the choices that were made during the design and implementation phases were aimed at fulfilling the functional and non-functional requirements.

Specifically, from the outset it was stated that the application had to provide high availability and fault tolerance.

As is well-known from the CAP theorem, distributed databases cannot have consistency, availability, and partition protection at the same time. Thus, in order to meet the non-

functional requirements, we have always tried to prioritize availability and partition protection over consistency.

This is particularly evident in the implementation of the servers that manage the Key-Value Database and its replicas, but it can still be seen within the entire project.

Of course, consistency is still an important requirement, and we try to maintain it whenever possible, especially when dealing with important data. However, in the event of a failure, we would rather show the user inaccurate data than stop the application from working.

As an example, let us consider how the client behaves when communicating with the server to perform an operation on the Key-Value Database.

When the application opens, one of the three possible servers is chosen randomly; from then on, all requests will be sent to that server.

However, if while sending a request the client notices that the server has disconnected for some reason, the client automatically selects another server and starts sending requests to it.

Obviously, if the first server failed before completing the writes to the replicas, the client will retrieve inconsistent data. In any case, the user will be happy to continue using the application and will take care of fixing any inconsistencies himself/herself.

So, this is an example of how availability and partition tolerance were chosen at the expense of consistency.

3.2 USER MANUAL

3.2.1 Navigation bar

The navigation bar (Figure 54) is included in every page. It is composed of an image of the application logo, a search bar and some clickable labels like “Home”, “Login”, etc.



Figure 54. Navigation bar

The *search bar* is used to search a product in the application, by specifying a keyword in the field. When the keyword is inserted, users can either click on the “Enter” button on their keyboard or click on the magnifying glass button to trigger the search.

They will then be redirected to the Browse Product page and all the products matching the keyword will be displayed.

Labels are used to navigate through the application. For instance, the “Home” button will redirect the user to the Browse Product page, the “Login” to the Login page, the “Register” to the registration page, and so on and so forth.

If the user has already logged in, the Login label will be replaced with his/her username (Figure 55). Logged users can access their personal page with all their details by clicking the *username label*.



Figure 55. Navigation bar for a logged standard user

Also, if the user is a *standard user*, a new button will appear in the navigation bar: the cart button. The cart button will not be displayed to *sellers* and *administrators*, because they cannot place orders.

3.2.2 Registration

The registration page (Figure 56) can be accessed by every type of users; even already logged users can register a new account.

Figure 56. Registration form

Users must fill all the fields: *Name*, *Surname*, *Password* and *Confirm Password*, *Username*, *Country*, and *Age*.

If a field is empty or the *Password* and *Confirm Password* are not the same, an error will be triggered, and the user will not be registered.

The username must be unique, so if there is already a user with that username in the database, the user will be forced to change it in the registration phase.

If all the fields are filled correctly, a message will be displayed, and the user will be logged in automatically.

3.2.3 Login

The login page (Figure 57) can be accessed only if the user is not already logged in, otherwise this page is inaccessible.

Users must fill both fields, specifying their *Username* and *Password*. If a user is already registered in the database with that username and password, the login will be completed; otherwise, an error message will be displayed.

The logged user will be redirected to the main page of the application, the Browse Page.

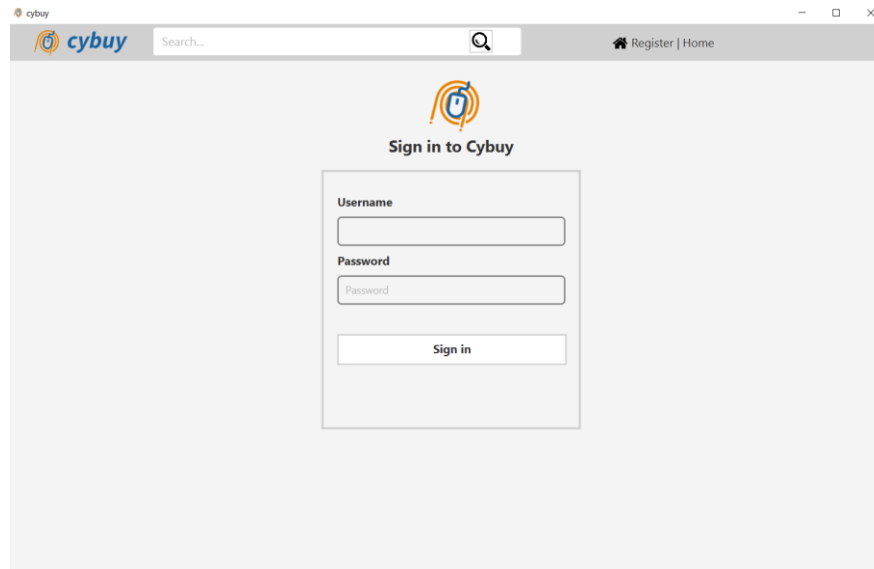


Figure 57. Login page

3.2.4 Browsing Products

The Browse Page (Figure 58) is the main page of the application. When a user starts the application, this page will be displayed at first.

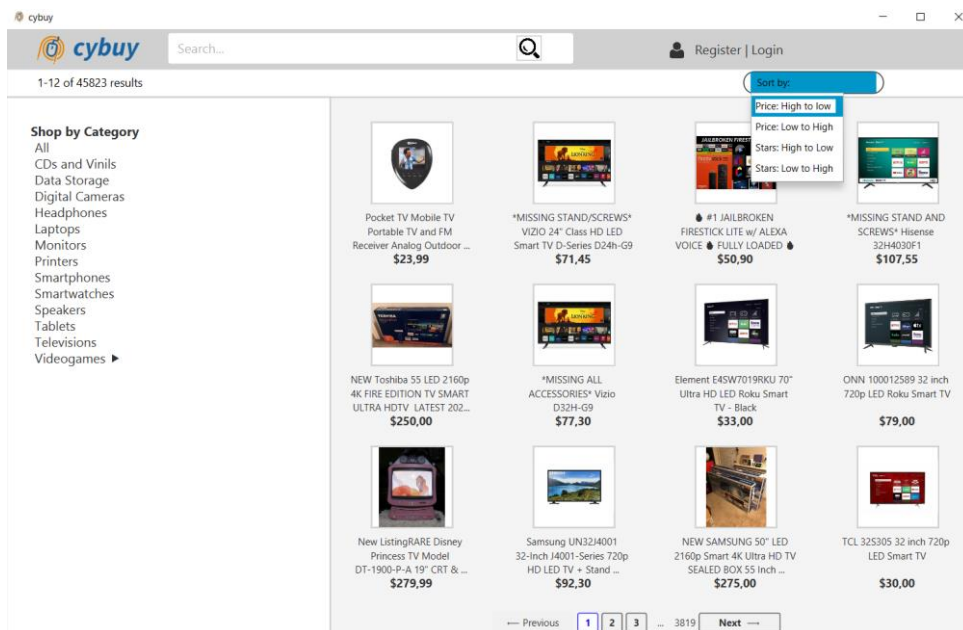


Figure 58. Browse Page

Only twelve products are displayed in one page; the user can navigate through different pages selecting the buttons at the bottom of the page.

If no category is selected, the application will display the most purchased products from all categories. Otherwise, the most bought products from the selected category will be shown.

The categories (and subcategories) are selected from the sidebar on the left and only one category can be selected at a time.

Below the navigation bar there is a bar. This bar will display the total number of products belonging to the selected category or, if the category is not selected, the total number of products for sale.

On the right we have a menu, by which the user can select a *sort* filter to display the products in a specific order, sorting them by price or star reviews.

Note that, if the user has logged in as a *seller* or an *administrator*, this page will display only the products that the user is currently selling.

3.2.5 Product Details

This page (Figure 59) can be accessed by clicking on a specific product in the main page. Also not logged users can visit this page.

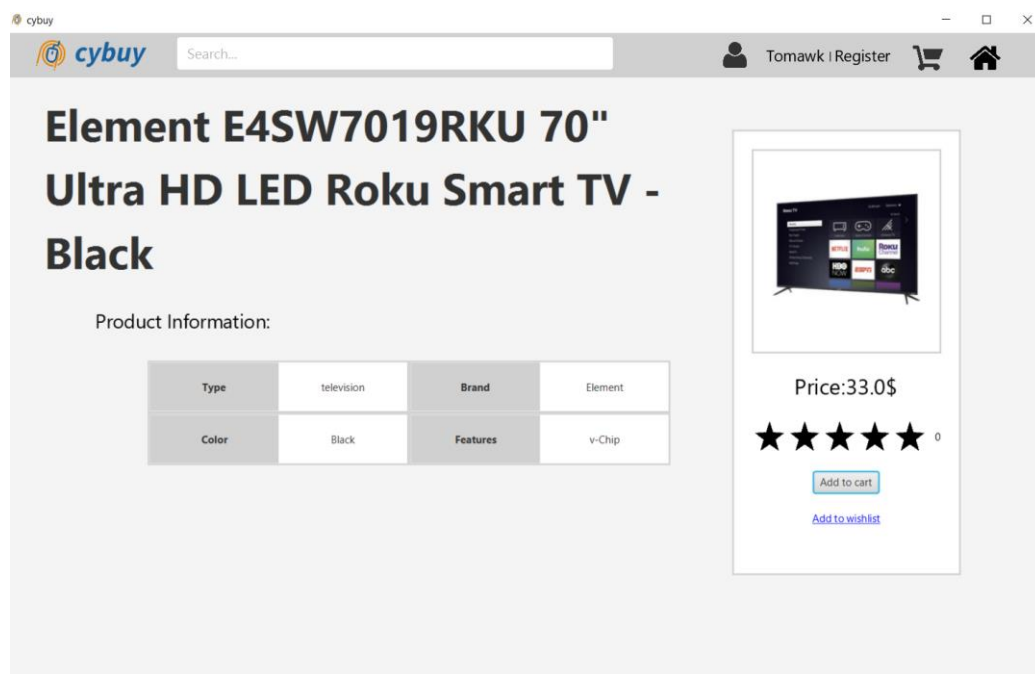


Figure 59. Product Details Page (Users Only)

This page will display all the details regarding a specific product. That is, its *description*, *image*, its *features* (such as the color, brand, etc.), *price* and *star reviews*.

In this page logged users can insert the product in their cart or in their wishlist clicking on the corresponding button. Users can also place a review on the product if they have already bought it by clicking on one of the stars.

If the user is an *administrator* or a *seller*, this page will have additional functionalities in order to let him/her modify all the information about the product (Figure 60). In this case, a user can, for example, change the *description*, the *price*, the *image* and add/remove *features* about the product. Finally, users can apply all the changes made to the product by clicking on the corresponding button.

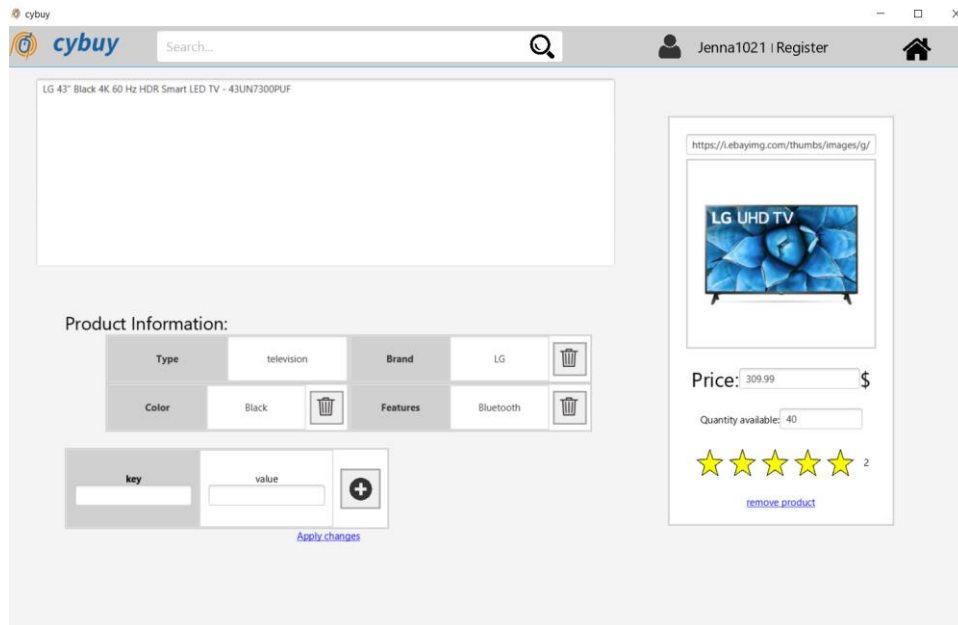


Figure 60. Products Details Page (Seller or Admin only)

3.2.6 Account Page

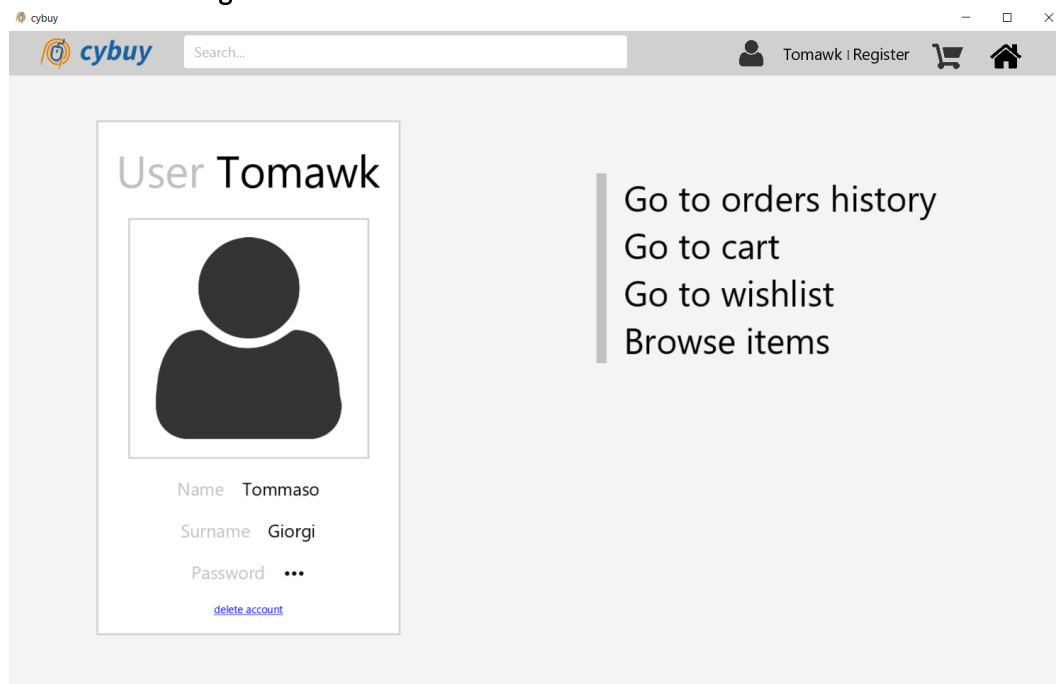


Figure 61. Account Page

The Account Page (Figure 61) is accessible only from a logged user. In this page users can see all the details about their profile, including the *Name*, *Surname*, and *Password* they inserted in the registration phase. They can also delete their profile clicking on the *delete account* button.

From this page, all users can go to their order history, *standard users* can go to their cart and to their wishlist, *sellers* can access their Analytics Page or the Add Product page, and *administrators* can access the Administration Panel.

3.2.7 Orders History

The Order Page can be accessed by any logged user from the Account Page. If the user is a standard logged user, this page will show all the active orders that the user has placed (Figure 62).



Figure 62. Active Orders Page (Normal Logged Users)

The user can see all the details and also the state of the order. If the state is pending the user can delete the order clicking on the trash bin icon. If the user clicks on the “View Past Orders” button, active orders will be replaced by past orders (Figure 63). Information about the product like the description, price and the image are not displayed in the past orders.

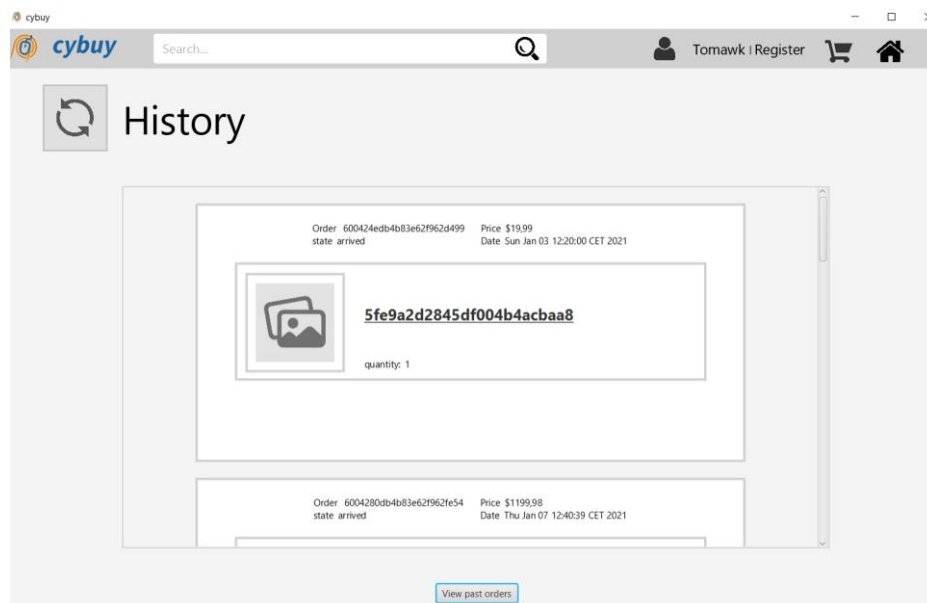


Figure 63. Past Orders Page (Normal Logged Users)

If the user is a seller, he/she has the possibility to change the state of the order and to visualize all the details of the active (Figure 64) and past orders (Figure 65).

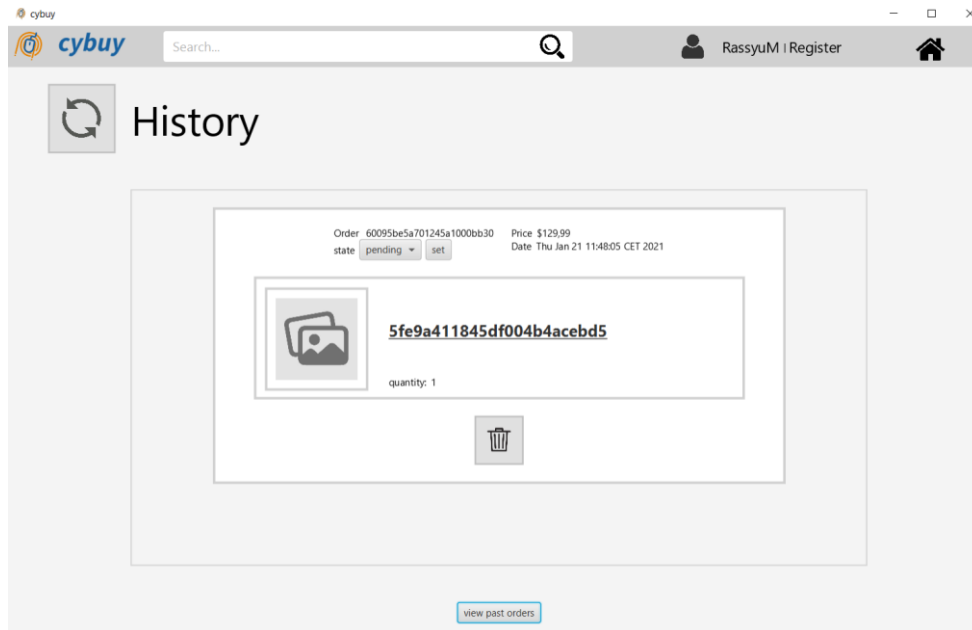


Figure 64. Active Orders Page (Sellers/Admins)

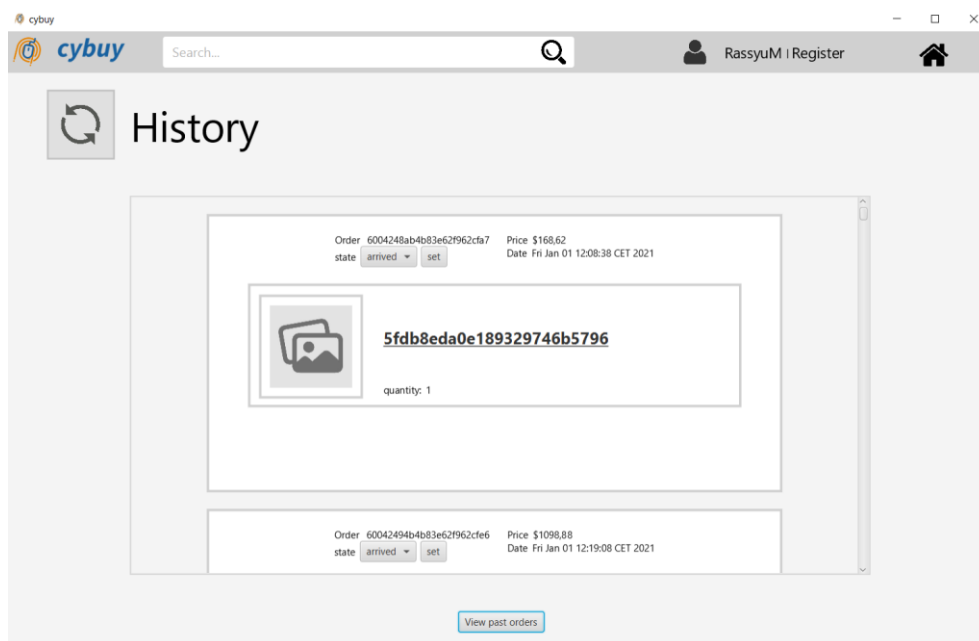


Figure 65. Past Orders Page (Sellers/Admins)

3.2.8 Standard User functionalities

3.2.8.1 Cart

The Cart Page can be accessed only from standard logged users, in fact unregistered users, sellers and admins cannot place any order.

When there aren't products in the cart, an image of an empty cart will be displayed, and a new button will appear (Figure 66).

By clicking it, the user will be redirected to the main page in order to search for some products to add to the cart. Also, the recap window on the right will be empty and the "checkout" button will be unclickable.

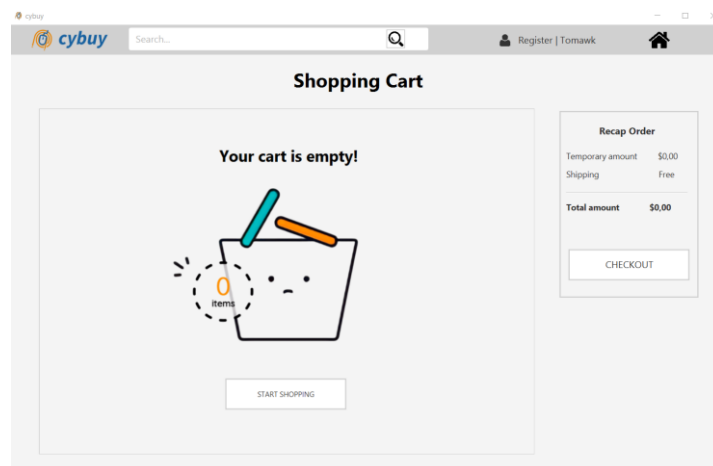


Figure 66. Empty Cart Page

Otherwise, if the cart is not empty, it will display all the products including their own description and their own price (Figure 67).

Also, the recap order will be filled with the sum of all the prices of every product. The user has the possibility to remove an item from the cart clicking on the trash bin button or increase/decrease his quantity in the cart clicking on the corresponding buttons.

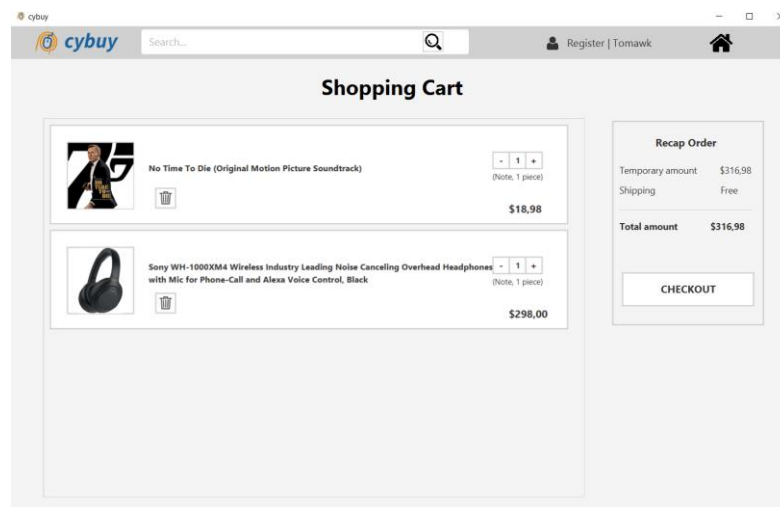


Figure 67. Cart Page

When all the products are inserted properly in the cart, the user can finally place the order clicking on the checkout button.

3.2.8.2 Wishlist

Like the cart, the wishlist is only reachable from a logged standard user. If its empty it will display this image and a button that will redirect the user to the main page in order to search for some products to add to his wishlist (Figure 68).

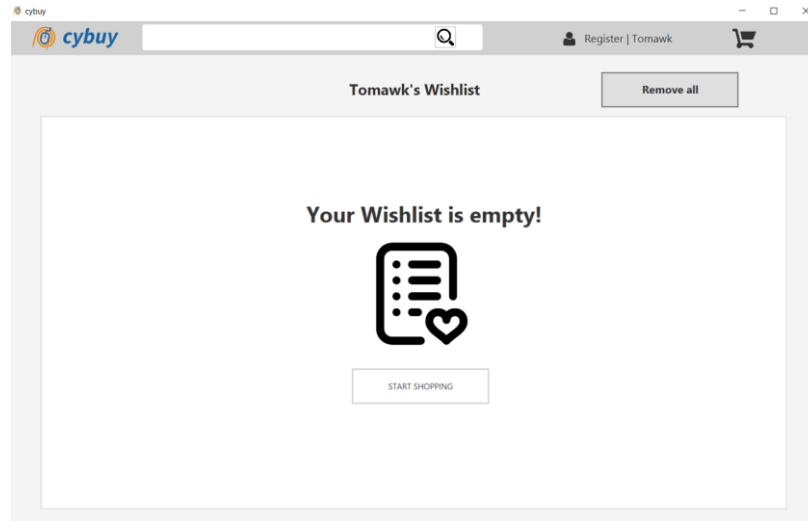


Figure 68. Empty Wishlist Page

Otherwise, if the wishlist is not empty (Figure 69), it will display all the products adding the possibility for the user to move the product from the wishlist to the cart clicking on the “add to cart” button.

Every product has also a trash bin button, if the user clicks on this button the item will be removed from the wishlist. On the wishlist we have also the “remove all” button, that is very useful if the user wants to remove all the products in the wishlist clicking just one button.

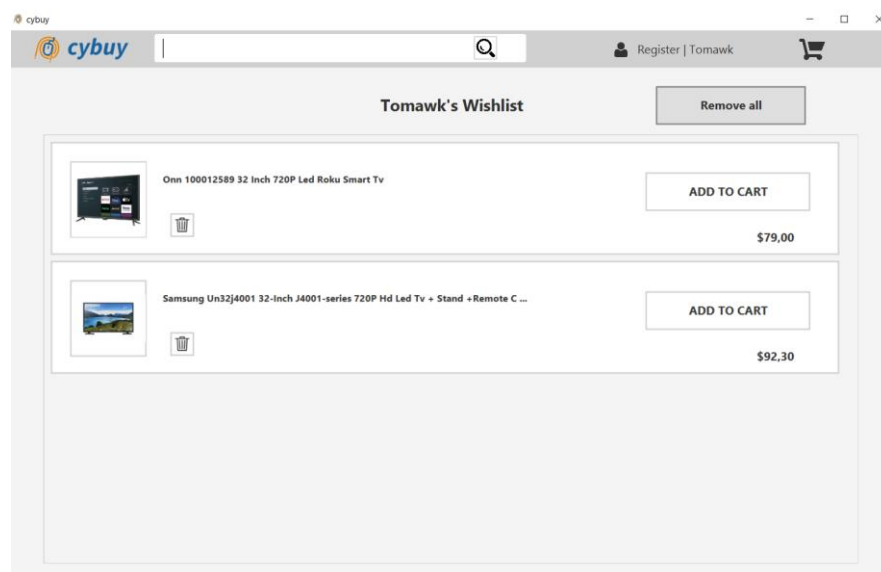
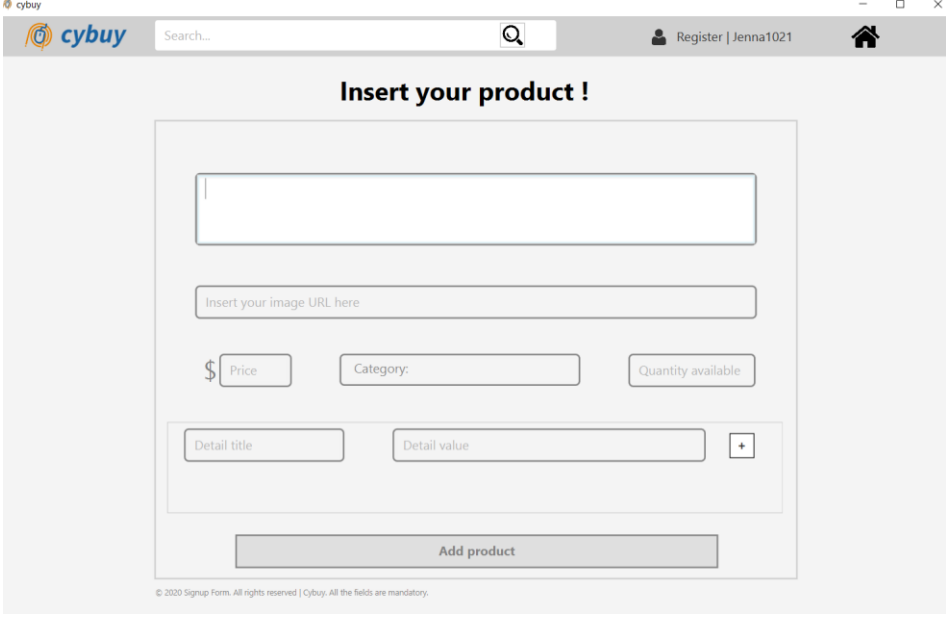


Figure 69. Wishlist Page

3.2.9 Seller functionalities

These functionalities are exclusive to Sellers and Admins. Normal logged user cannot perform any of these services provided by the application.

3.2.9.1 Add a new product



The screenshot shows the 'Insert your product !' page in the cybuy application. The page has a header with the cybuy logo, a search bar, and user links. The main form contains the following fields:

- A large text input for the product description.
- A text input for the image URL with the placeholder text 'Insert your image URL here'.
- Three input fields: 'Price' (with a dollar sign icon), 'Category:', and 'Quantity available'.
- Two input fields for 'Detail title' and 'Detail value', followed by a '+' button to add more details.
- An 'Add product' button at the bottom.

At the bottom of the form, there is a small copyright notice: '© 2020 Signup Form. All rights reserved | Cybuy. All the fields are mandatory.'

Figure 70. Insert Product Page

In this page (Figure 70) an Admin or a Seller can add a new product to sell.

The description, URL image, price, category and the quantity available are mandatory fields. The details instead are additional features that the user can also omit.

If a user wants to insert more than one detail feature about the product, he can simply click on the “+” button and other fields will appear.

When all the fields are correctly filled, the user can finally click on the “Add Product” button and the product will be inserted in the application.

3.2.9.2 Analytics

This page (Figure 71) is very useful for admins and sellers. In fact, it allows them to view statistics about the products they are selling.

They can view their most sold product, the number of products globally sold and other useful information. They can easily manage reviews and they can view an history of their sales in the last month also filtering the results by the age of the purchasers or the gender.

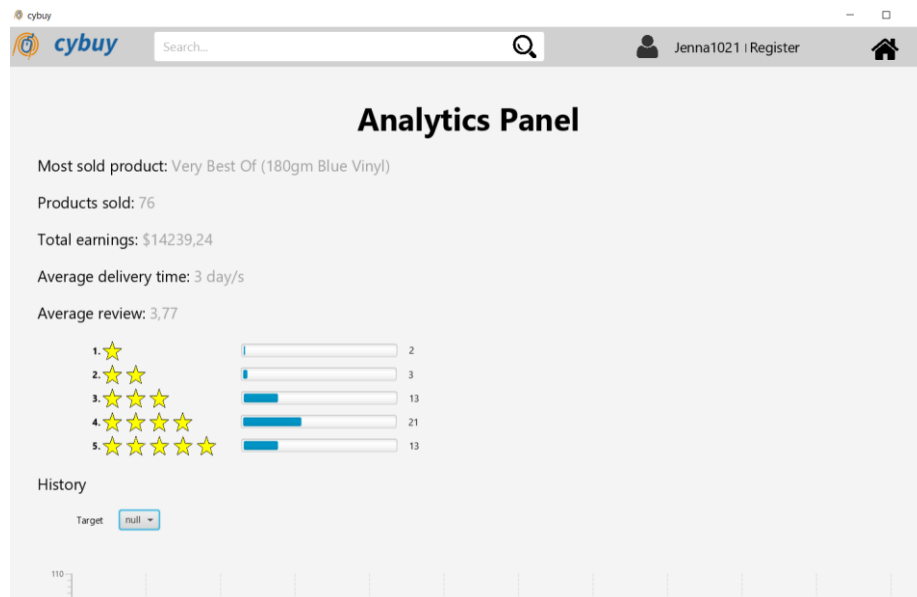


Figure 71. Analytics Page

3.2.10 Administrator functionalities

3.2.10.1 Browsing Sellers

This page (Figure 72) can be accessed only from admin users by clicking on the corresponding label in the user page.

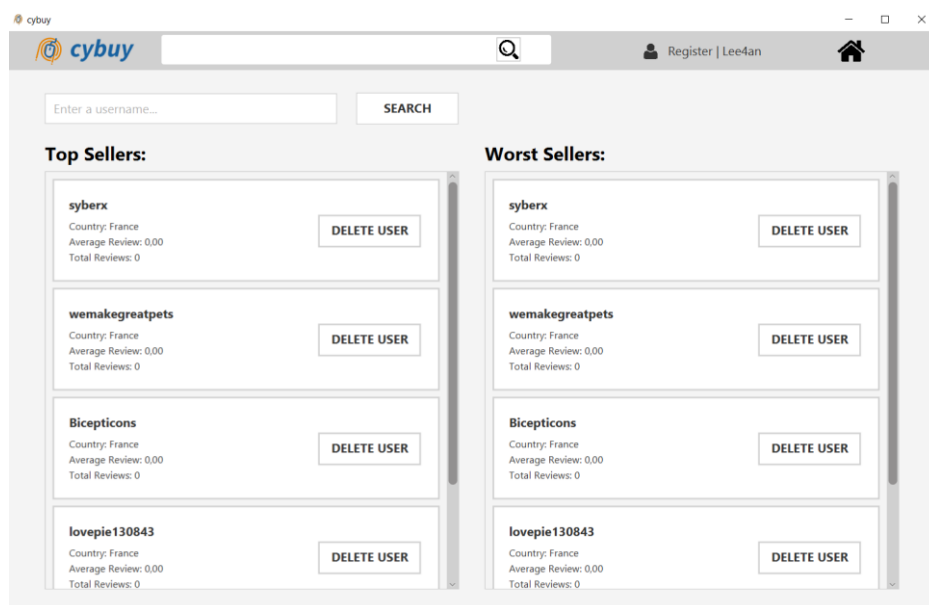


Figure 72. Browse Sellers Page

This page is used by the admins to manage the sellers, checking all their information and their reviews. If a seller must be removed, the admin can easily search for him using the search bar and clicking on the “delete user” button. In this page we have two panels, in the panel on the left will be displayed the top sellers in terms of reviews and in the other panel the worst sellers.

4 LIST OF FIGURES

Figure 1. UML diagram of the main Use Cases.....	4
Figure 2. UML diagram of the Analysis Classes	5
Figure 3. Section of the UML diagram of the Analysis Classes	6
Figure 4. Example of document in the Users collection (Standard User)	7
Figure 5. Example of document in the Users collection (Seller).....	8
Figure 6. Example of document in the Products collection	9
Figure 7. Example of document in the Orders collection.....	10
Figure 8. Example of document in the Analytics collection	11
Figure 9. Section of the UML diagram of the Analysis Classes	12
Figure 10. Ring structure	14
Figure 11. Two-tier Client-Server Architecture	16
Figure 12. System architecture.....	17
Figure 13. Directory structure of the project repository	19
Figure 14. Command <code>crontab -l</code> executed on the primary server (IP: 172.16.3.138)	20
Figure 15. Function <code>updateAnalytics()</code> code – PART I	20
Figure 16. Function <code>updateAnalytics()</code> code – PART II	21
Figure 17. Function <code>updateAnalytics()</code> code – PART III	21
Figure 18. Class Server variables.....	22
Figure 19. Part of function <code>main()</code> in Server class.....	23
Figure 20. Package Structure.....	24
Figure 21. Add Product Query	26
Figure 22. Modify Product Query	27
Figure 23. Delete Product Query	28
Figure 24. Get Product List Query.....	28
Figure 25. Get Number of Products Query.....	29
Figure 26. Get Product from Id Query	30
Figure 27. Insert User Query	30
Figure 28. Delete User Query	31
Figure 29. Find User Query.....	32
Figure 30. Insert Review Query	32
Figure 31. Insert Order Query	33
Figure 32. Delete Order Query	34
Figure 33. Change Order State Query	35
Figure 34. Get Seller Analytics Query.....	35
Figure 35. Get Seller List By Ranking Query	36
Figure 36. Aggregation that retrieves the total number of reviews and the average review of a seller	37
Figure 37. The analytic to retrieve the distribution of the Reviews	38
Figure 38. The analytic to calculate the average delivery duration.....	39
Figure 39. Aggregation that retrieves the total earnings of a seller	39
Figure 40. Aggregation that retrieves the total number of sales for a seller	40

Figure 41. Aggregation that retrieves the daily analytics for a seller	41
Figure 42. Insert Product in Cart.....	42
Figure 43. Remove Product from Cart	43
Figure 44. Retrieve all Products in Cart.....	43
Figure 45. Add Product to Wishlist	44
Figure 46. Remove Product from Wishlist	45
Figure 47. Retrieve all Products in Wishlist.....	45
Figure 48. Insert Product Redundancies	46
Figure 49. Incrementing the usage value	47
Figure 50. Decrementing the usage value.....	48
Figure 51. Cleaning Redundancies on a Product	48
Figure 52. Performance improvement introduced by the indexes	50
Figure 53. Queries performance with and without the index on seller_username in the Products collection	50
Figure 54. Navigation bar	51
Figure 55. Navigation bar for a logged standard user.....	52
Figure 56. Registration form.....	52
Figure 57. Login page	53
Figure 58. Browse Page.....	53
Figure 59. Product Details Page (Users Only).....	54
Figure 60. Products Details Page (Seller or Admin only).....	55
Figure 61. Account Page.....	55
Figure 62. Active Orders Page (Normal Logged Users)	56
Figure 63. Past Orders Page (Normal Logged Users)	56
Figure 64. Active Orders Page (Sellers/Admins)	57
Figure 65. Past Orders Page (Sellers/Admins)	57
Figure 66. Empty Cart Page	58
Figure 67. Cart Page.....	58
Figure 68. Empty Wishlist Page	59
Figure 69. Wishlist Page	59
Figure 70. Insert Product Page	60
Figure 71. Analytics Page.....	61
Figure 72. Browse Sellers Page.....	61