University of Pisa

Department of Information Engineering

Master's Degree Artificial Intelligence and Data Engineering

Large-Scale and Multi-Structured Databases Project: EasyDrugs

Professors:

Pietro Ducange

Alessio Schiavo

Students:

Nilo Fabiano

Lorenzo Valtriani

Samuele Marchi

ACADEMIC YEAR 2024/2025

Abstract

Easy Drugs aims to create a diverse ecosystem where Patients, Doctors, Pharmacies, and Researchers can help one another, both directly and indirectly, through services and information sharing: Patients can receive prescriptions from doctors and purchase them at pharmacies or get OTC drugs independently in the same place; doctors can additionally view the history of the patient's prescriptions in order to have more details to prescribe correct therapies; researchers can access statistics on global and local drug requests, analyzing the relationships between patients and doctors and studying new solutions for drugs with a reduced impact on the patient, understanding the correlation between active ingredients and side effects; all users can browse drugs and discover potential benefits or criticalities. Your data will be aggregated with all other user data and will not be used for any purposes not specified in the license agreement. We refer to data from the European Medicines Agency and the Italian Ministry of Health.
Our mission is to create a transparent, collaborative environment where every piece of data contributes to better health outcomes for all. You can download the app on your smartphone or just visit the page at www.easyDrugs.it.

"Alone we can do so little; together we can do so much." – Helen Keller

Contents

List of Figures

Chapter 1

Introduction

EasyDrugs is an application that aims to become a point of reference in the pharmaceutical sector, targeting different types of users: patients, doctors, pharmacists, and researchers. All users must register to the application in order to access it.

Below, we briefly list the main functionalities available for each role:

- The patient can monitor their active prescriptions, issued by their doctor, consisting of drugs that have not yet been purchased. They also have access to their purchase history.
- The doctor can prescribe drugs to their patients and can consult both the purchase history related to their prescriptions and the prescriptions that are still active.
- The pharmacist can record the purchases made by a specific patient and view their active prescriptions.
- The researcher has access to analysis tools and can modify information related to the drugs.

Finally, all users can search for information about a specific drug.

Key instruments to build the architecture were:

- MongoDB and Redis are the selected databases used to store information.
- Swagger and Postman are development tools used to document and test the APIs.
- Java is the programming language used to develop the application.
- Python is the programming language used to scrape data and collect information for the initial population of the databases for testing purposes.

The project follows Agile approach and standard basis of modern Software engineering. The application and other resources are available on GitHub repository: https://github.com/trianitriani/EasyDrugServer/tree/main. IntelliJ-community edition and OpenVPN should be installed before trying the application.

Chapter 2

Design

A well-designed application architecture requires the definition of actors who interact with the system, requirements to satisfy stakeholders and end users while preventing malfunctions, basic components and common interactions, and a formal model to describe it. At the end, a user manual will be provided, using mockups.

2.1 Actors

In this section, actors and their functionalities are presented. Each of them has a specific role and privileges.

- Patient: users who purchase drugs.
- Doctor: users who prescribe drugs to patients.
- Pharmacy: users who sell drugs to patients.
- Researcher: users who analyze system statistics.

All the actors mentioned before can browse information about drugs. The researcher is the system administrator. He can also add, delete, or edit drug information. Doctors, pharmacies, and researchers have accounts with specific responsibilities and therefore require certification, as they perform sensitive actions affecting patients (prescribing, selling, modifying drug information).

2.2 Requirements

This chapter presents both functional and non-functional requirements, describing essential elements for the proper use of the system. It describes user actions and aspects related to performance, security, and compliance with standards.[9]

2.2.1 Functional Requirements

USER

- The system must allow only registered users to access services.
- The system must allow users to log in with their identify code and password.
- The system must allow users to view their private information in the profile area.
- The system must allow users to delete their account.
- The system must allow users to view a list of drugs by entering a name prefix.
- The system must allow users to view all details of a selected drug from the list.

PATIENT

- The system must allow patients to view all active prescriptions, organized by prescription date, including the list of drugs, where each has a name, price and quantity.
- The system must allow patients to determine whether a drug in an active prescription has been purchased.
- The system must allow patients to view their last five purchases, including prescribed and over-the-counter (OTC) drugs, with drug name, price, and quantity.
- The system must allow patients to view their purchased drugs in groups of ten, sorted by the most recent first.
- The system must allow patients to edit their profile information, including city, district, region, doctor code, and password.
- The system must display an error message if the patient inserts an already existing tax code in the sign-up form during the registration

DOCTOR

- The system must allow doctors to search for patients by surname prefix.
- The system must allow doctors to access a patient's active prescriptions using his identify code.
- The system must allow doctors to access a patient's prescription cart using his identify code.

- The system must allow doctors to search for prescribable drugs by name prefix.
- The system must allow doctors to add or remove a prescribable drug from a prescription cart by id.
- The system must allow doctors to edit the quantity of a prescribable drug in a prescription cart by id.
- The system must allow doctors to create a new active prescription for a specific patient, containing the drugs in the prescription cart.
- The system must allow doctors to view prescribed drugs that have been purchased by a specific patient, in groups of ten, sorted by the most recent first.
- The system must allow doctors to edit their profile information, including city, district, region, and password.
- The system must display an error message if the doctor inserts two entries for the same prescribable drug into the patient's prescription cart.

PHARMACY

- The system must allow pharmacies to visualize a patient's purchase cart using their identify code.
- The system must allow pharmacies to view all active prescriptions, organized by prescription date, including the list of drugs, where each has a name, price, and quantity.
- The system must allow pharmacies to determine whether a drug in an active prescription has been purchased.
- The system must allow pharmacies to search for OTC drugs by name prefix.
- The system must allow pharmacies to add or remove an OTC drug from a patient's purchase cart by id.
- The system must allow pharmacies to add or remove a prescribed drug from a patient's purchase cart by id and prescription date.
- The system must allow pharmacies to edit the quantity of an OTC drug in a patient's purchase cart by Id.
- The system must allow pharmacies to edit their profile information, including the owner's tax code and password.
- The system must display an error message if the pharmacy inserts two entries for the same OTC drug into the purchase cart.

RESEARCHER

- The system must allow the researchers to view the most purchased drugs and the number of purchases for each of them, within a selected time period.
- The system must allow researchers to view the patient-to-doctor ratio by city in ascending or descending order.
- The system must allow the researchers to view the least frequently indications in drugs and displaying drug names and the number of occurrences.
- The system must allow researchers to view the percentage and total number of drug purchases for each region within a selected time period.
- The system must allow researchers to add, edit, and delete drug information by id.
- The system must allow researchers to edit their profile information, including city, district, region, and password.

2.2.2 Non functional requirements

- The system must be implemented as PWA (progressive web app).
- The system must be intuitive and user-friendly without requiring specific training.
- The system must be developed using an object-oriented programming language.
- The system must ensure high availability.
- The system must ensure partition tolerance.
- The system must be able to store and quickly query a large volume of data.
- The system must ensure data integrity in case of a crash.
- The system must ensure inter-database consistency.
- The system must encrypt passwords with actual security standard.
- The system must store all active prescriptions in the key-value database for at most one month.
- The system must store all drugs in the prescription cart in the key-value database for at most one day.
- The system must store all drugs in the purchase cart in the key-value database for at most one hour.

2.3 Class diagram

A UML is a class diagram Figure 2.1 and models data within the application, representing main entities, relationships and data structures used by the system. It is fundamental to ensuring integrity and coherence of stored information.

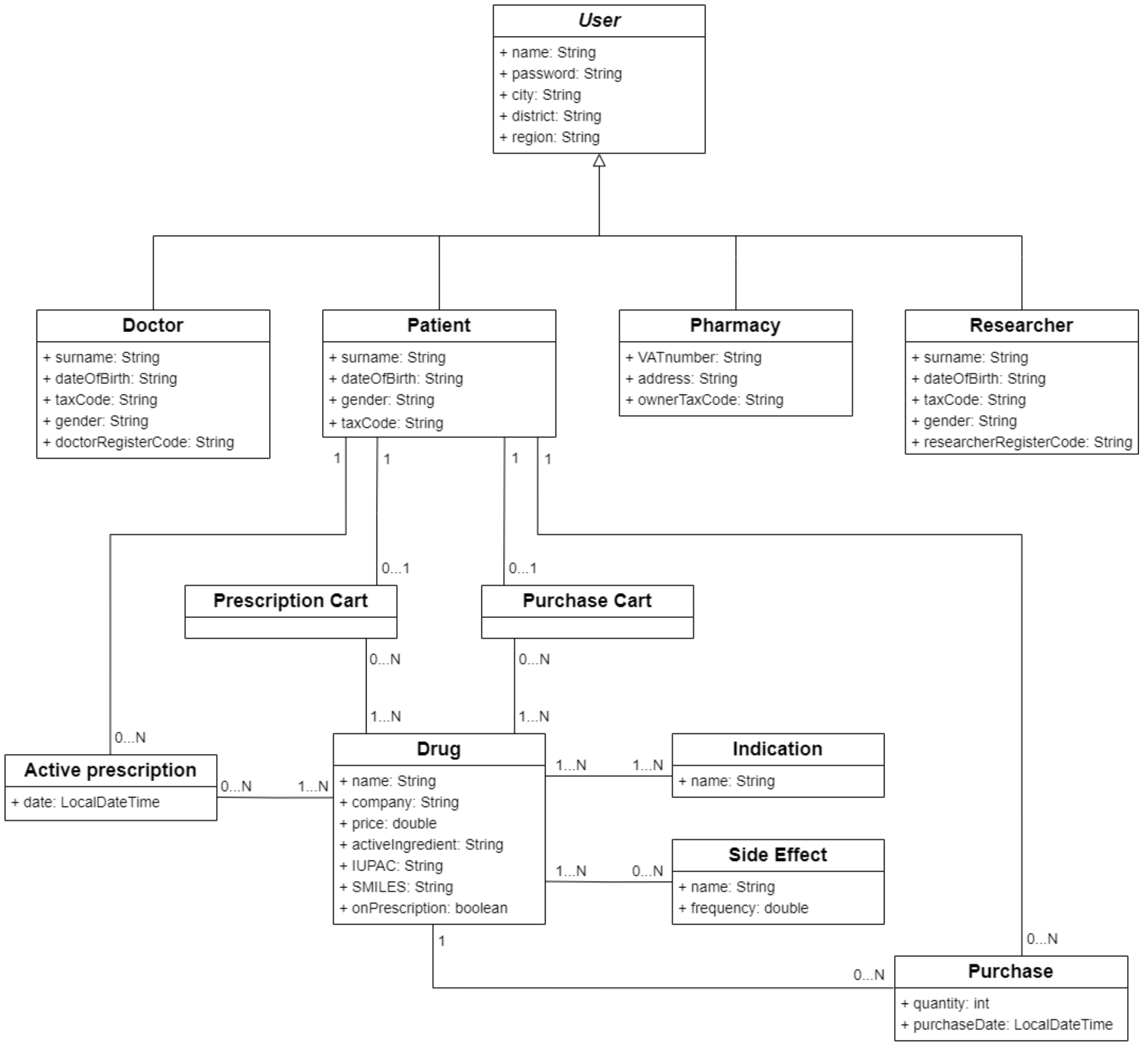


Figure 2.1: UML class diagram

2.4 Manual

This section provides a detailed presentation on how to use the application, including screenshots of the mockup for the mobile version. The authentication process, navigation, and specific functionalities based on the user's role in the system will be shown. Icons and HUD elements may differ from the browser version.

2.4.1 Log-in phase

When the application is opened, the user must log in by entering their identify code and password and then pressing "Log In" Figure 2.2.

Figure 2.2: Login-page

If he doesn't have an account, he can click on "Are you not registered yet?" and make a new account. The first thing to set is the "Role" field where you can choose what type of account he wants to create : Patient, Researcher, Doctor, Pharmacy Figure 2.3.

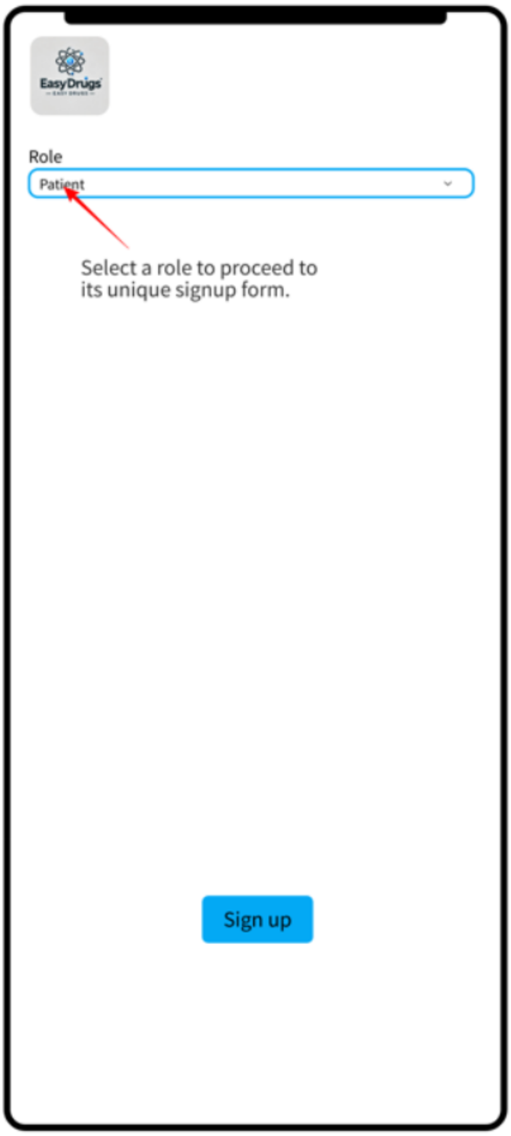


Figure 2.3: Selection of role in the sign up

Next, a form will be displayed with different fields depending on the selected role. In general, users must provide their name, surname, gender, date of birth, city, district, region, and a proper password. Doctors and researchers must also enter a valid registration code for the medical or clinical research registry, respectively. Pharmacies must provide their name, address, city, district, region, owner's tax code, VAT number, and password.

Clicking the "Sign-Up" button will save the account if all fields are completed and valid Figure 2.4.

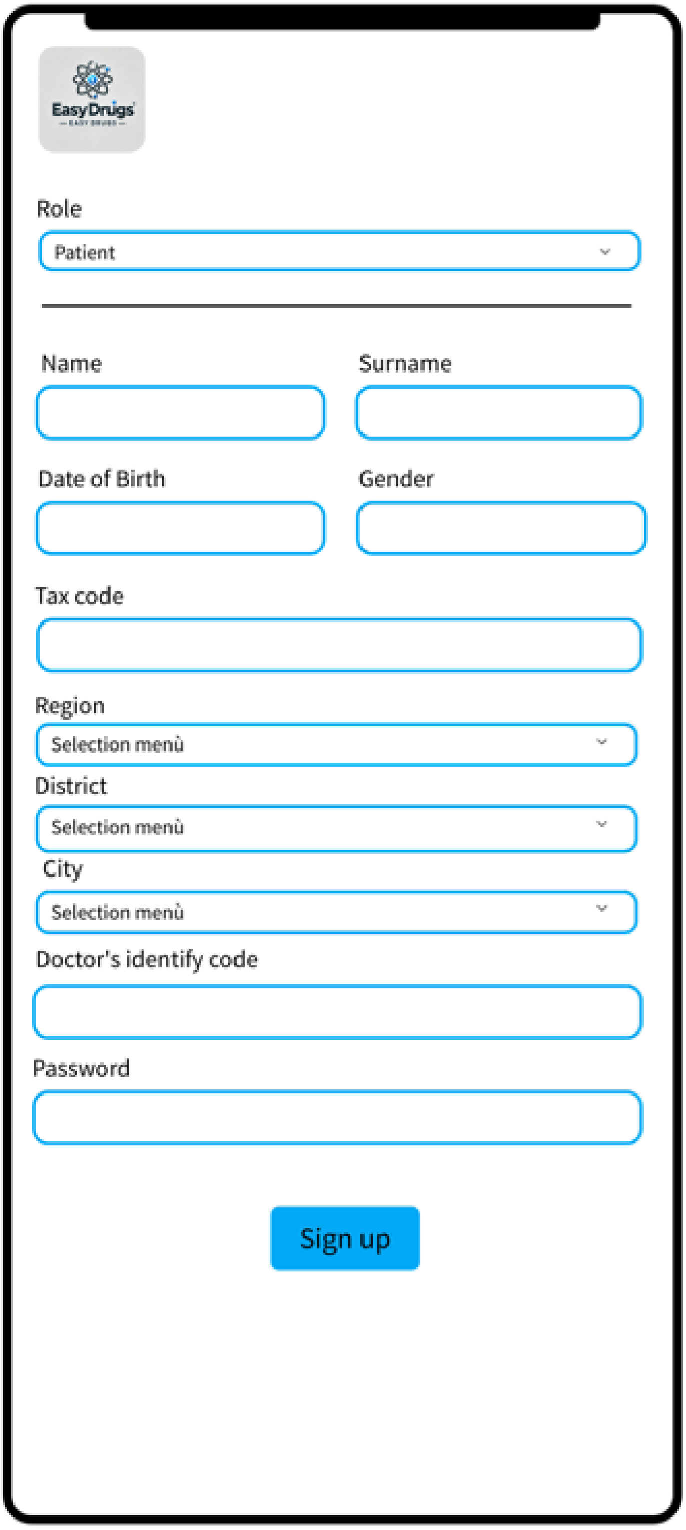


Figure 2.4: Sign up form

The identify code will be structured as follows:

- For patient: "P" + [tax code]
- For doctor: "D" + [tax code]
- For researcher: "R" + [tax code]
- For pharmacy: "Ph" + [VAT number]

A user can have only one account per role (e.g., a doctor can have both a doctor and a patient account but not two doctor accounts). After this phase, the user is directed to the home page.

2.4.2 For all users

Each user has a unique interface depending on their role (see the following subsections), but all users will always have three icons at the bottom of the window: a search icon, a home icon, and a profile icon. Clicking the home icon will always return the user to the home page, while clicking the search icon will open the browsing area.

In the browsing area, users can type the name of a drug (see Figure 2.5) and then click the search icon to view a list of occurrences Figure 2.6.
Clicking on a drug from the list will display its details, including the name, price, company,list of indications (maximum of 10), list of side effects (maximum of 10) with probabilities, active ingredient, IUPAC code, and SMILES code. If there is an icon under the price, the drug requires a prescription. This page is also accessible from any context within the application when clicking on a drug's name Figure 2.7.

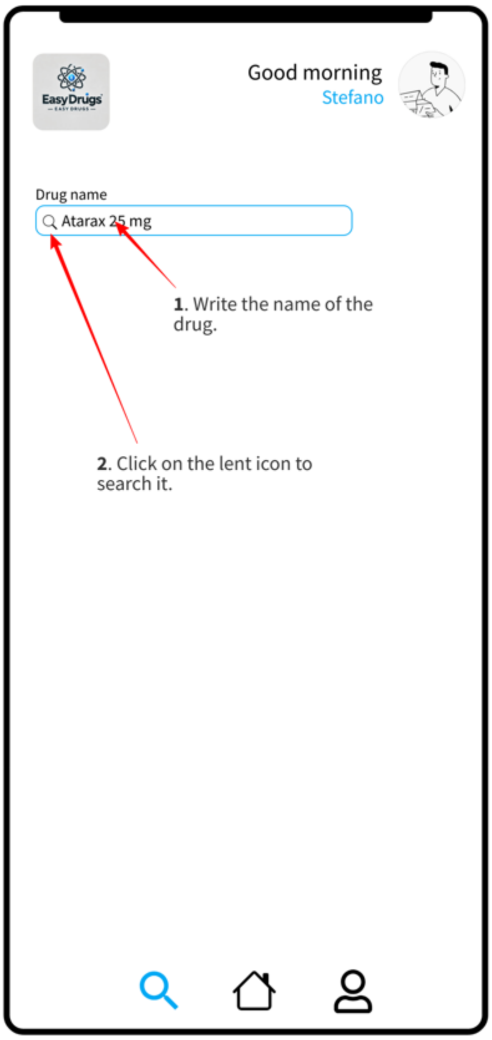


Figure 2.5: Browsing area from the patient's perspective (initial view)

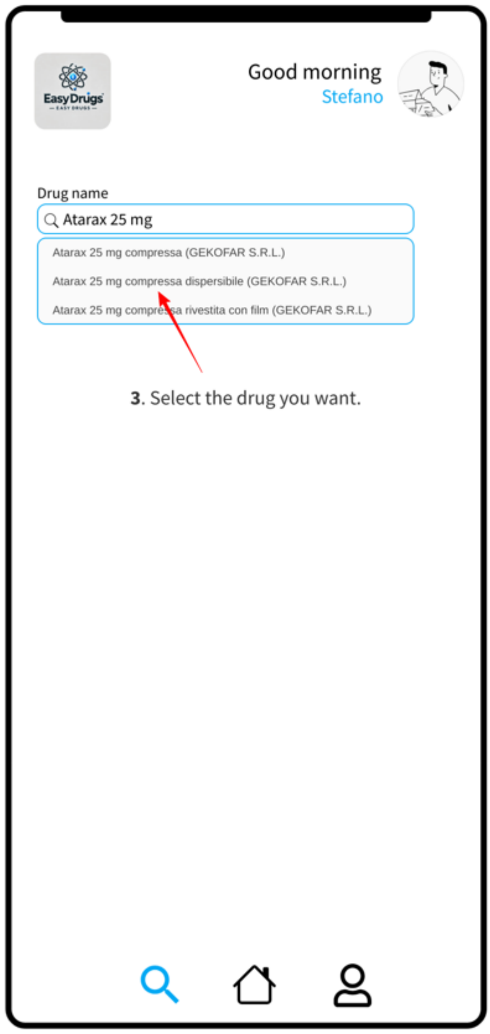


Figure 2.6: Browsing area from the patient's perspective (search results)

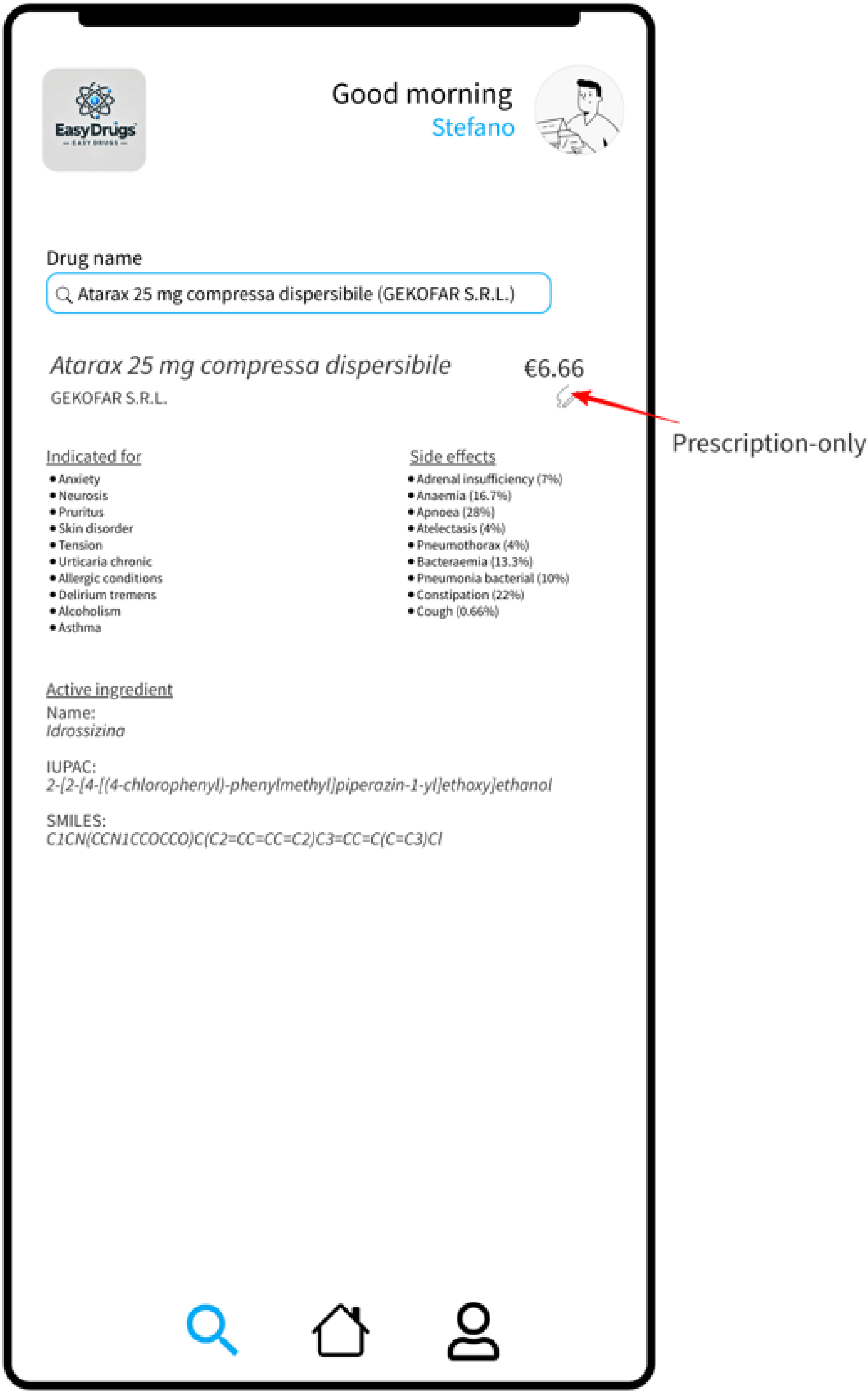


Figure 2.7: Information about a drug

Users can access their profile page by clicking the profile icon, where they can view personal information, change their password, and update their residence details. Patients can also update their assigned doctor's identify code, while pharmacies can modify their address and owner tax code. To confirm changes, users must click the "Edit" button. To delete an account, users can click the "Delete account" button below the "Edit" button Figure 2.8.

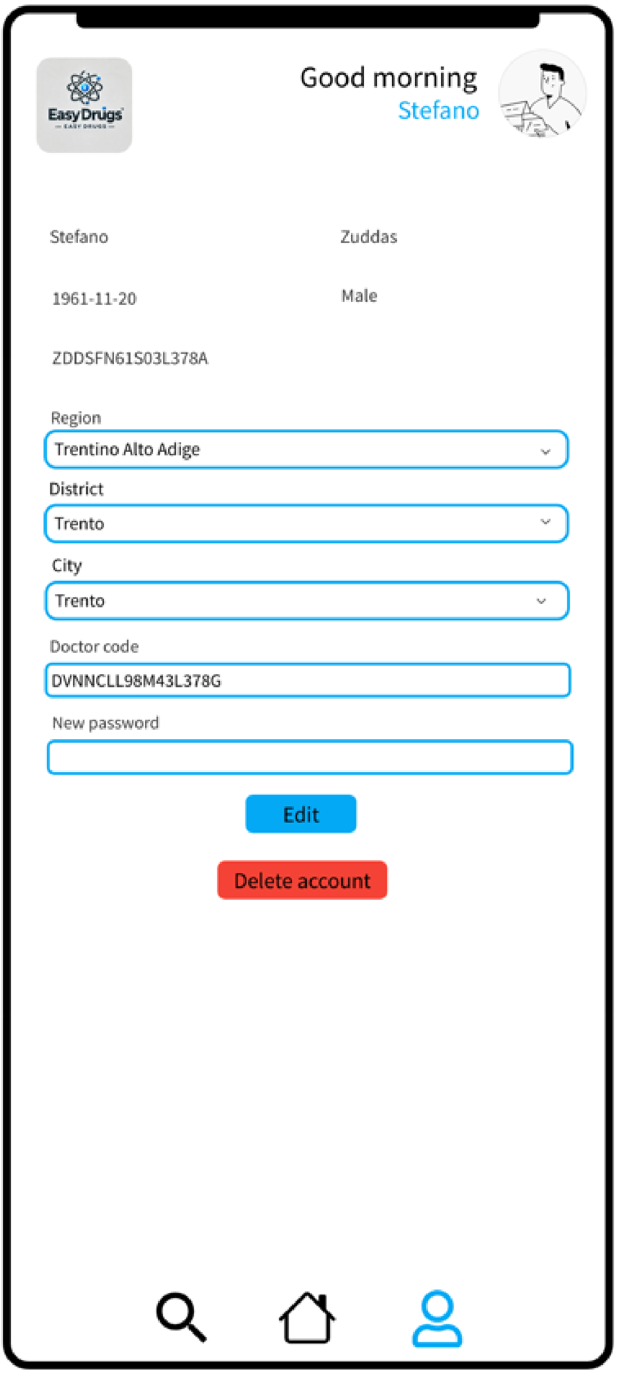


Figure 2.8: Profile Area

2.4.3 Patient

In the home section, there is a window with two selectable tabs: "My active prescriptions" and "My old purchases". The first tab displays the patient's active prescriptions (with also drugs that have not yet been fully purchased). Each prescription includes the date of prescription in the top-right corner of the box and a list of drugs, showing the price per unit and the quantity. Drugs that have already been purchased are displayed in a light gray font Figure 2.9.

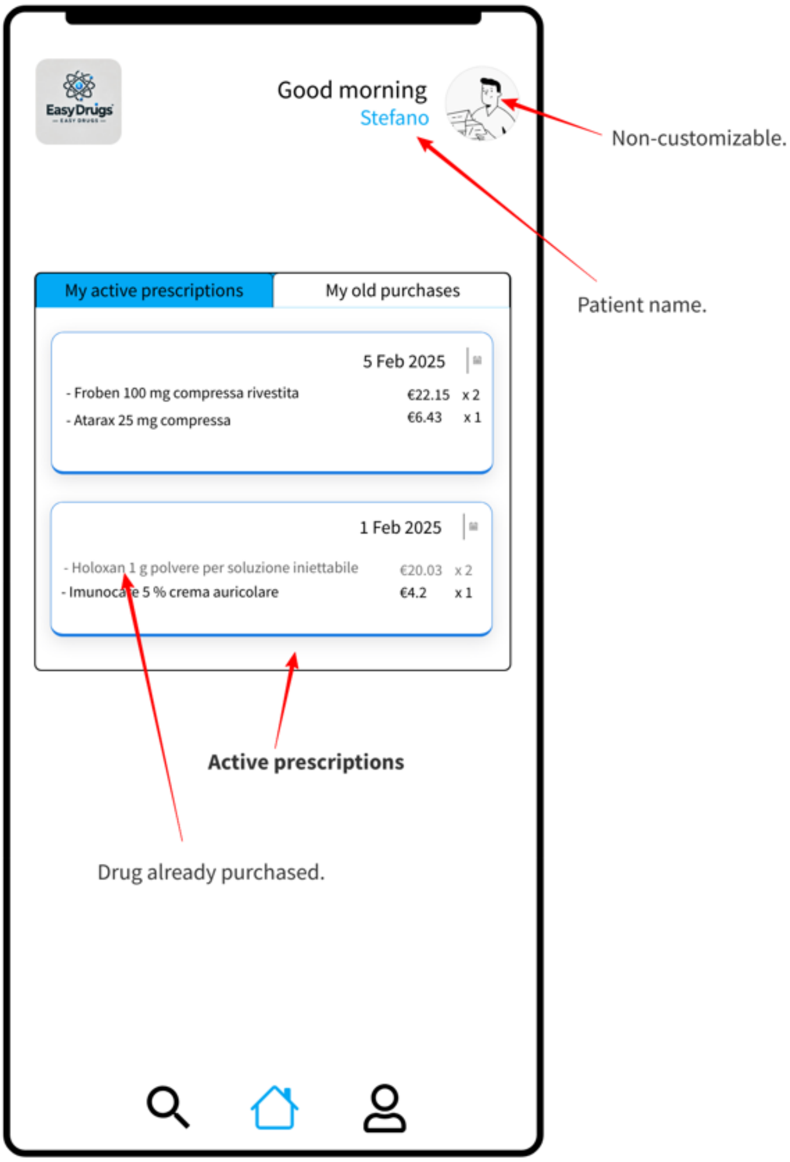


Figure 2.9: Patient active prescriptions

The second tab displays the last five purchase lists. Each list, similar to a prescription, includes the purchase date in the top-right corner of the box, a list of drugs with their respective quantities and prices. Prescribed drugs have a pencil icon next to the price Figure 2.10.

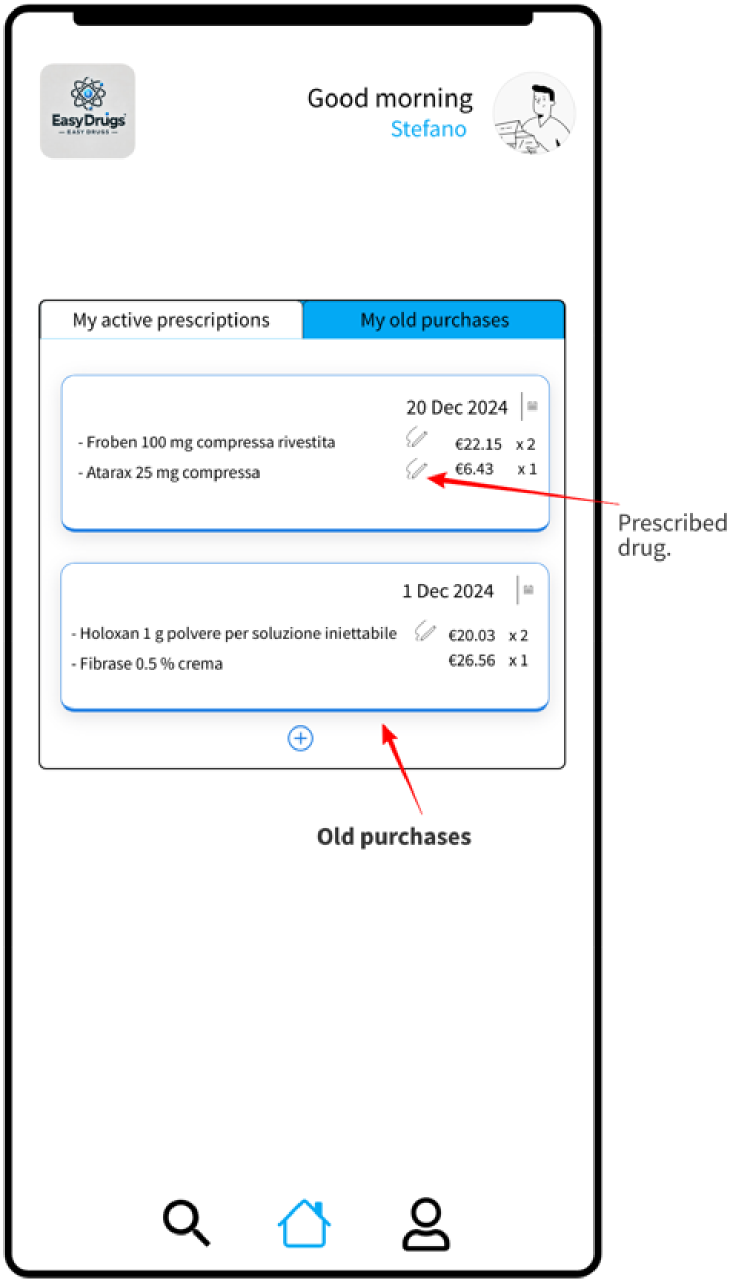


Figure 2.10: Patient old purchases

2.4.4 Doctor

In the home section, initially, there is only a search field for finding patients by surname Figure 2.11. Clicking on the search icon will display a list of possible results Figure 2.12, and selecting a patient will open two windows Figure 2.13.

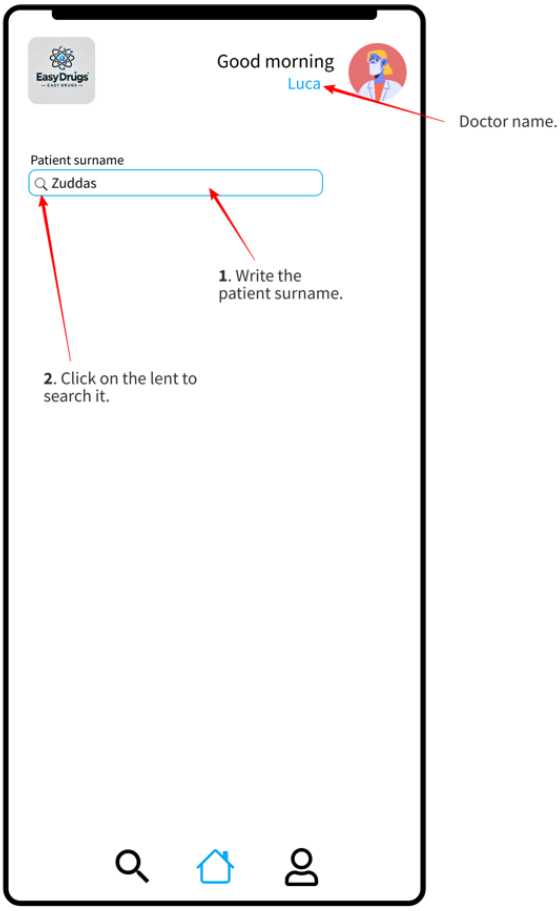


Figure 2.11: Doctor's search bar

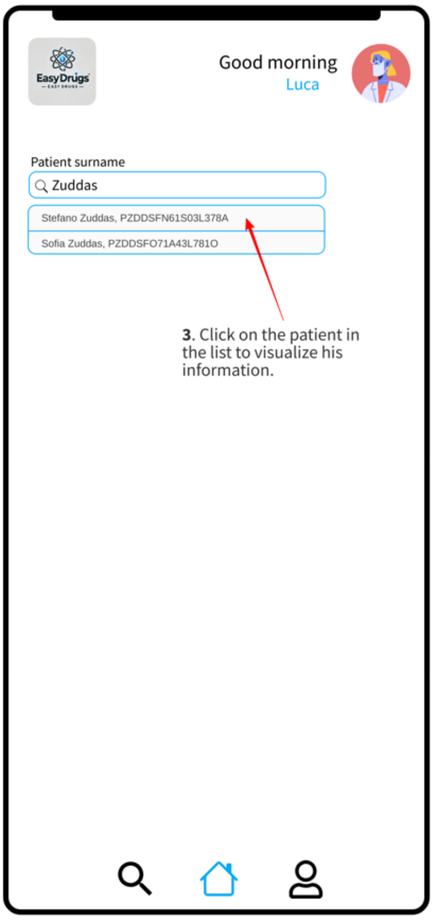


Figure 2.12: Patient selection

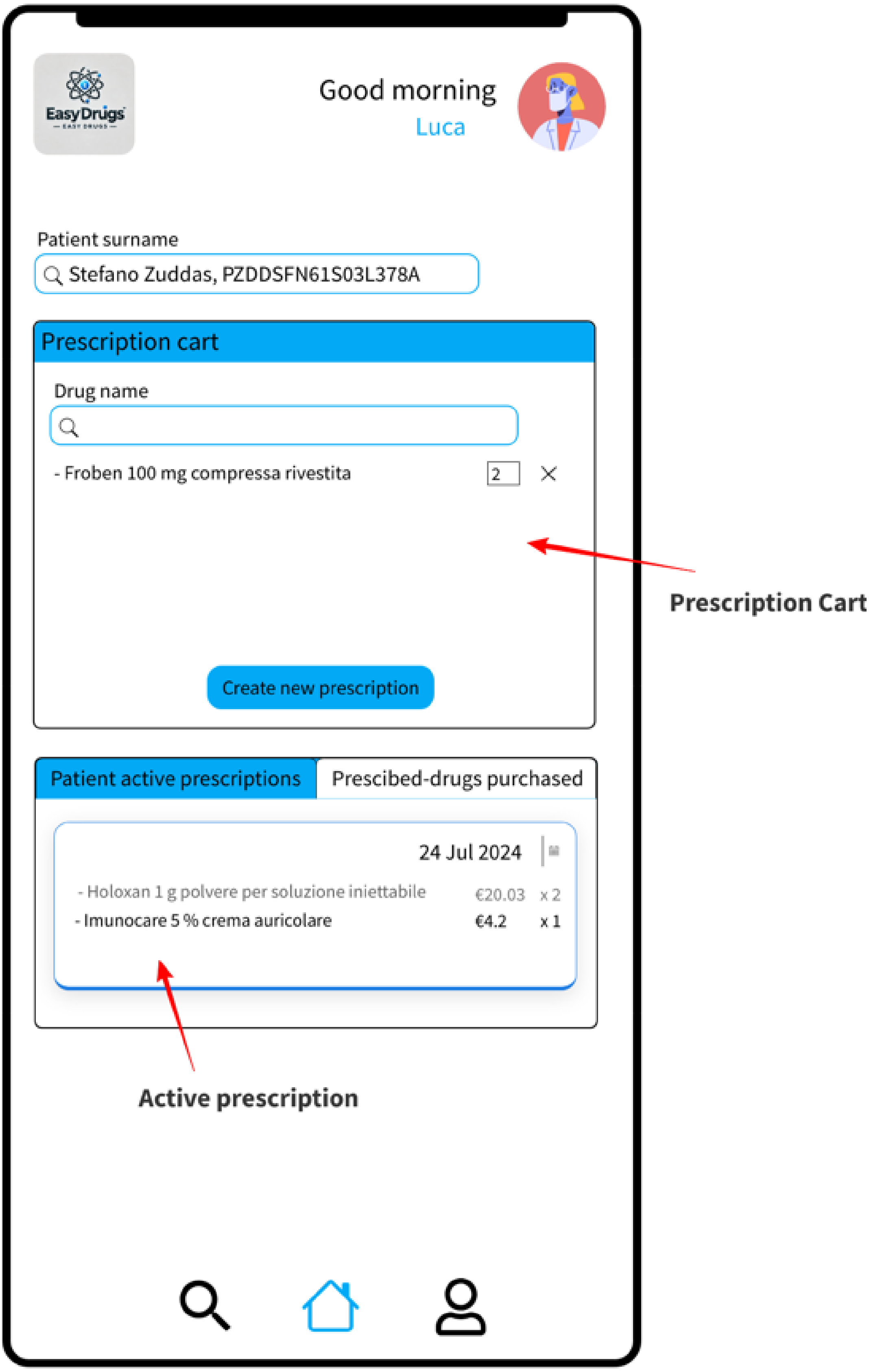


Figure 2.13: Doctor main windows

The top window is the prescription cart. Inside, there is a search bar to browse prescribable drugs Figure 2.14. After clicking on the search icon, a list of available drugs appears. Selecting one will reveal a quantity field and a plus icon Figure 2.15. Once the quantity is selected and the plus icon is clicked, the drug is added to the

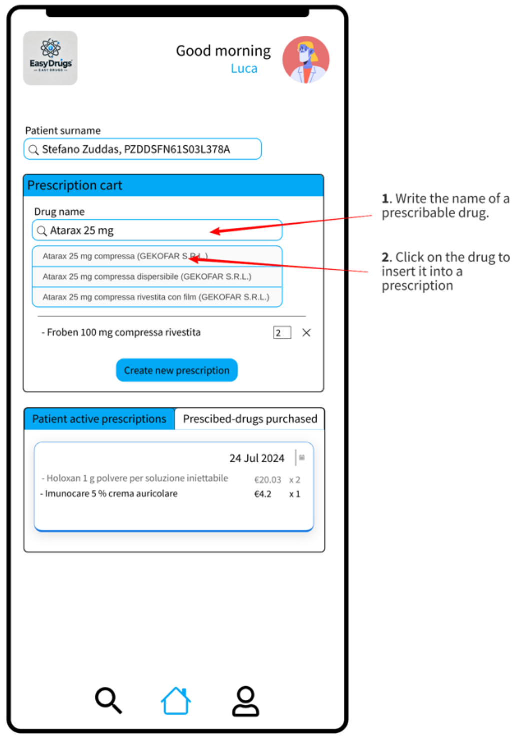


Figure 2.14: Search a prescribable drug

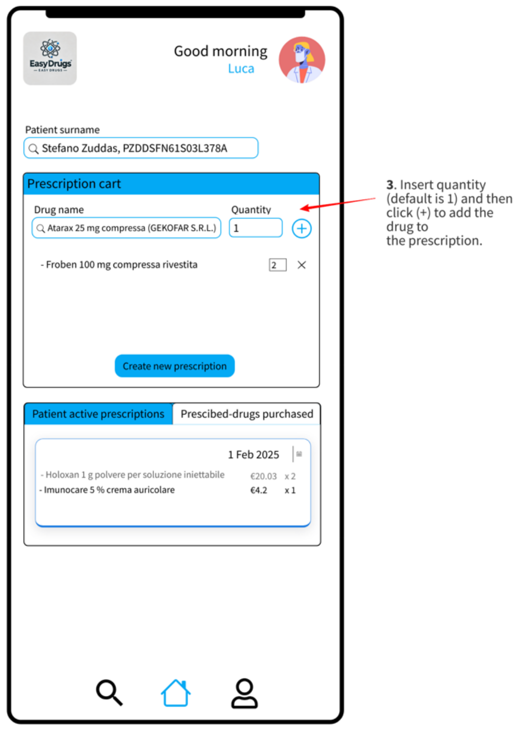


Figure 2.15: Add prescribable drug to the prescription cart

cart and displayed under the search bar. The quantity can be edited directly in the corresponding field. To remove an item, the user can click on the cross icon next to the quantity field. Below the list, there is a "Confirm new prescription" button to activate the prescription Figure 2.16.

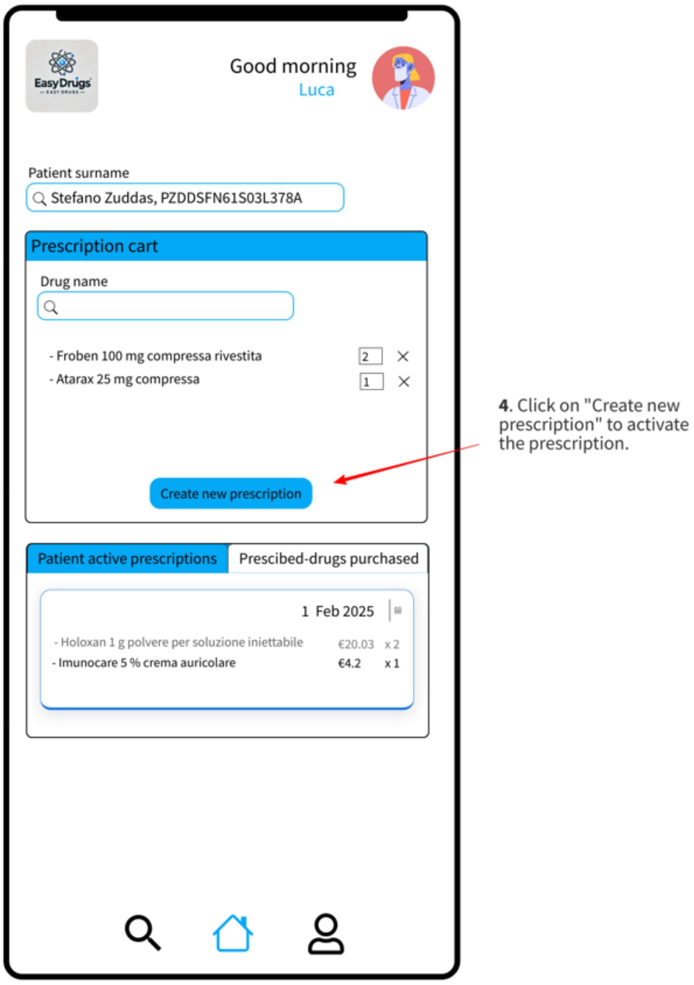


Figure 2.16: Activate a prescription

Below the first window, there is a second one with two selectable tabs: "Patient active prescriptions" and "Prescribed-drugs purchased". The first tab displays a list of active prescriptions, where new activated prescriptions appear. Each prescription has a date in the top-right corner, along with a list of drugs showing their quantities and prices Figure 2.17.

Figure 2.17: Patient active prescription

The second tab displays all drugs purchased by the patient that were prescribed by the doctor. Each entry follows the same format as in the first window Figure 2.18. Clicking on the plus icon at the bottom of the list will load ten more drugs at a time.

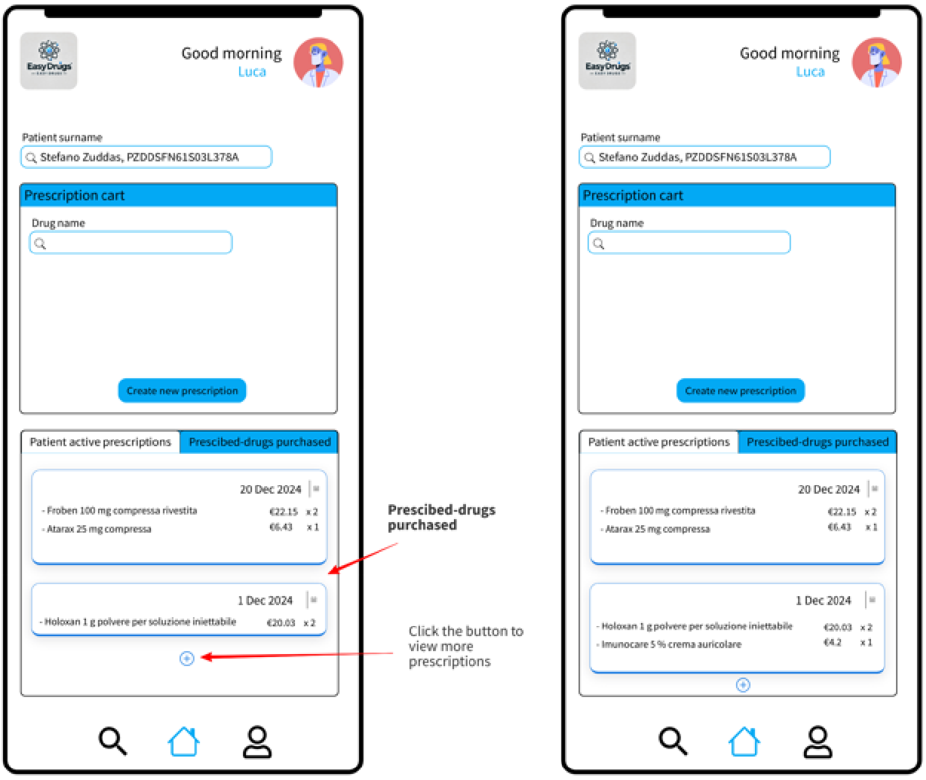


Figure 2.18: Prescribed-drug purchased

2.4.5 Pharmacy

In the home section, initially, there is only a field to search for the patient's identify code Figure 2.19. If the code is valid, two windows will be displayed Figure 2.20. The top window is the purchase cart. Inside, there is a search bar to browse OTC drugs. After clicking on the search icon, a list of available drugs appears. Selecting one will reveal a quantity field and a plus icon Figure 2.21. Once the quantity is selected and the plus icon is clicked Figure 2.22, the drug is added to the cart and displayed under the search bar. The quantity can be edited directly in the corresponding field. To remove an item, the user can click on the cross icon next to the quantity field. Below the list, there is a "Confirm purchase" button to validate the purchase Figure 2.24.

Below the first window, the second window displays a list of the patient's active prescriptions. Each prescription has a date in the top-right corner and a list of drugs with prices, quantities, and a cart icon Figure 2.23. Clicking on the cart icon adds the drug directly to the purchase cart. Prescribed drugs have a fixed quantity but can be removed using the cross icon.

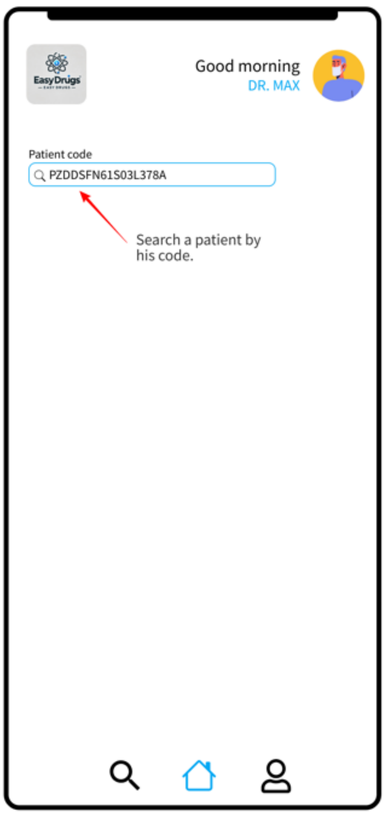


Figure 2.19: Pharmacy home

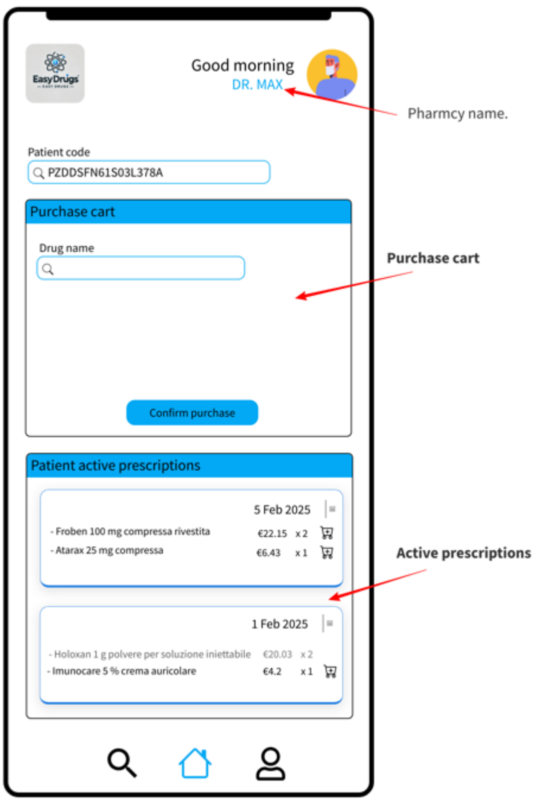


Figure 2.20: Main windows in pharmacy home

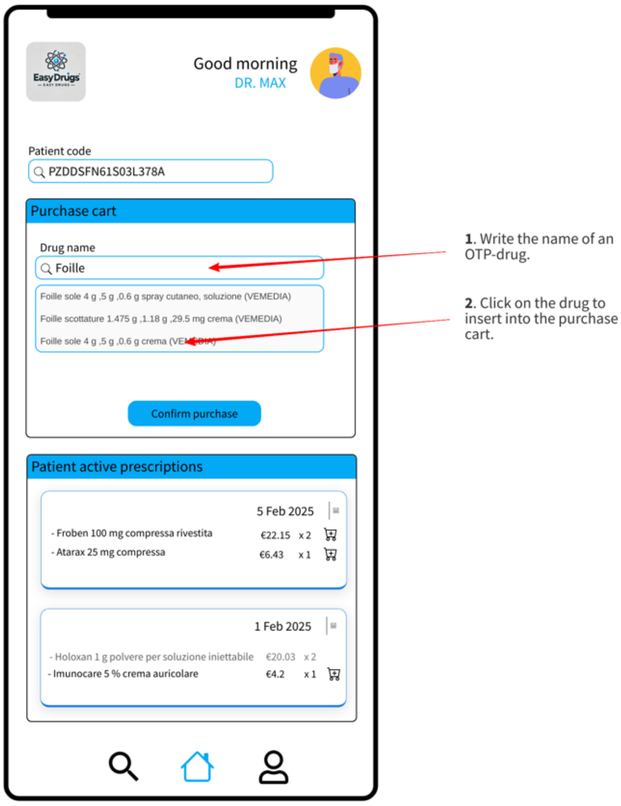


Figure 2.21: Add a OTP drug

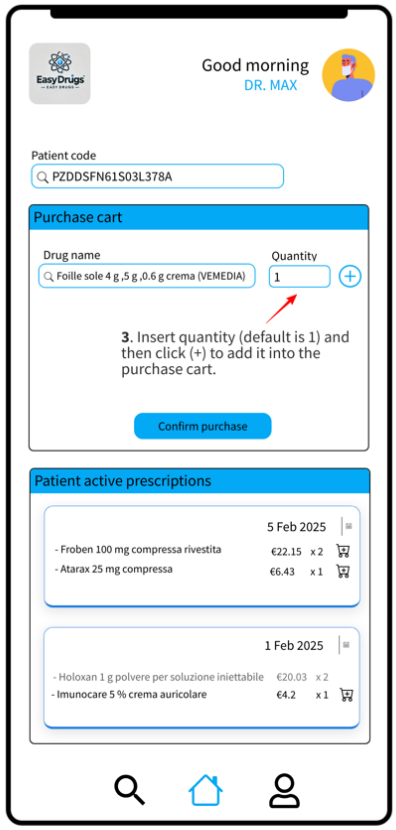


Figure 2.22: Add quantity

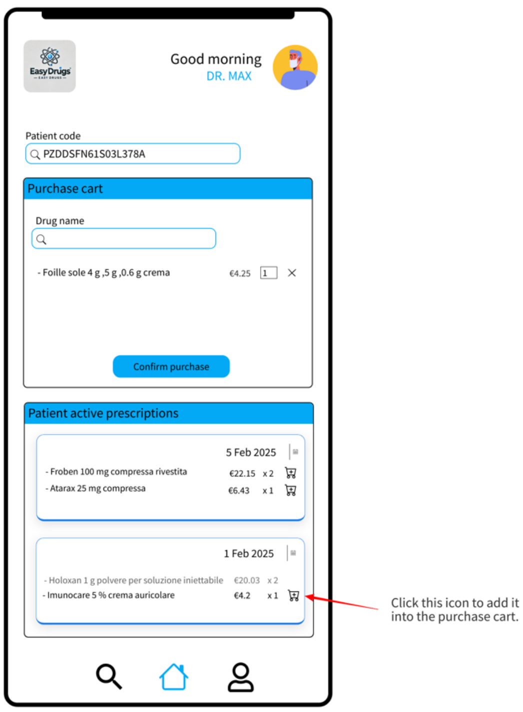


Figure 2.23: Add a prescribed drug

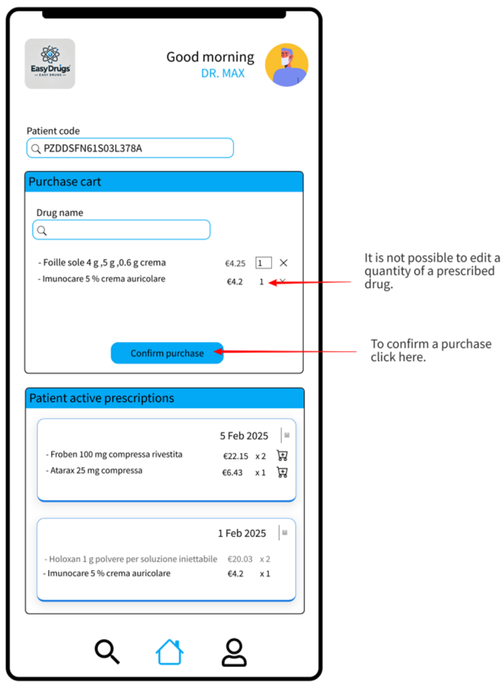


Figure 2.24: Confirm purchase

2.4.6 Researcher

In the home section, there is a dropdown menu for performing analytical queries:"Top drugs purchased", "Distribution of drug purchases on Italian territory", "Diseases with fewer drugs" and "Calculate patient/doctor ratio by city". Initially, the first query is selected Figure 2.28. Above this menu, there are two navigation buttons: "Edit information about drug" and "Analitycs query". The user can click only on the button marked with a green plus icon; the other is not interactive. When in the home section, the user can navigate only to the Edit Area Figure 2.25, and while in the Edit Area, they can return only to the home section Figure 2.26.

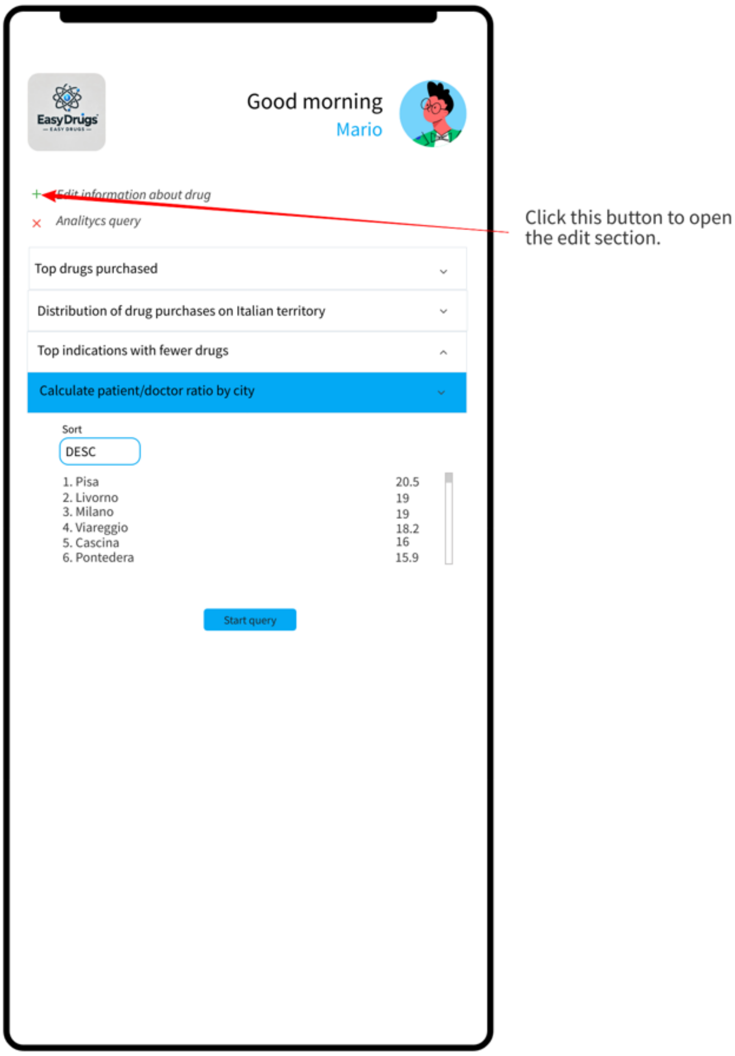


Figure 2.25: How to go in the edit area

In the Edit Area, users can type the name of a drug, click the search icon, and view a list of possible results. Selecting a drug displays its details in the corresponding fields. Users can edit the drug's name, price,, company, active ingredient, IUPAC code, SMILES code, prescription-only button check-box(if unchecked, the drug is classified as OTC). New indications or side effects can be added by entering text in the respective field and clicking the plus icon. After making changes, users can click the save button to update the database Figure 2.26.

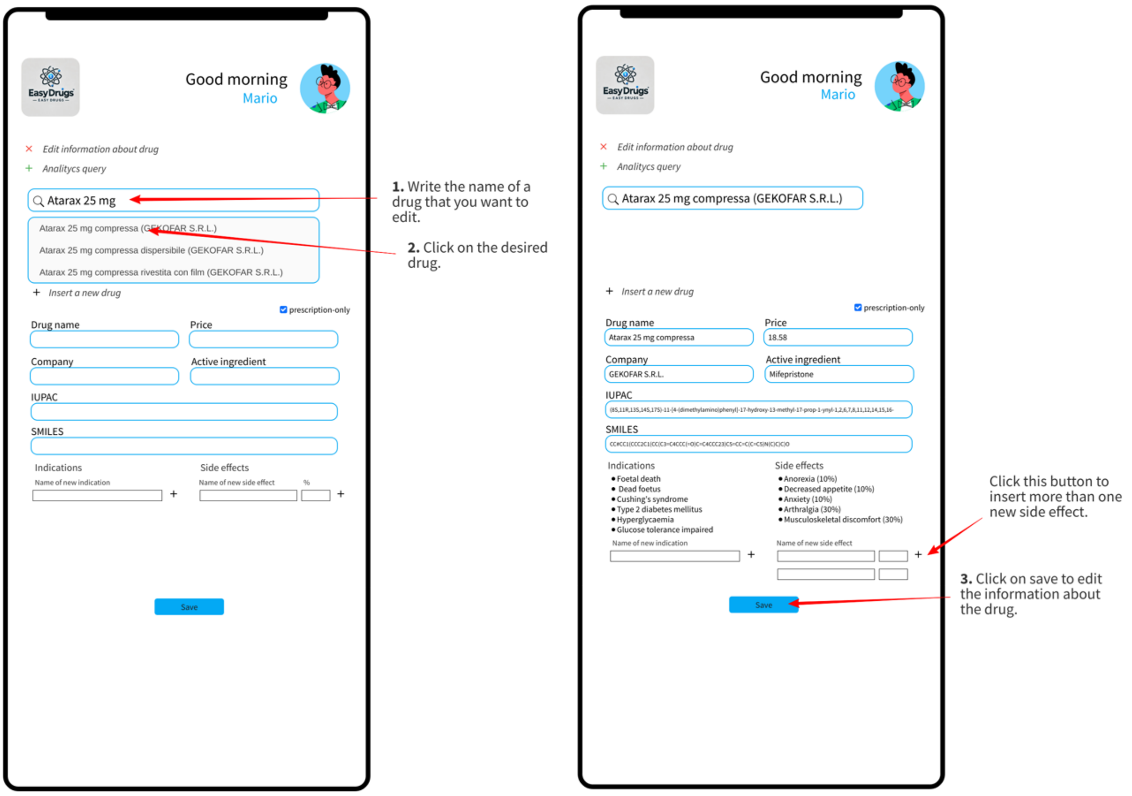


Figure 2.26: Editing a drug

Above the form, there is also an "Insert a new drug" button, which clears all fields. After filling in the required fields and clicking "Save," the drug is added to the database and becomes visible to all users Figure 2.27. A drug entry is considered valid if all required fields (except side effects, because drugs without them may exist) are completed and at least one indication is provided.

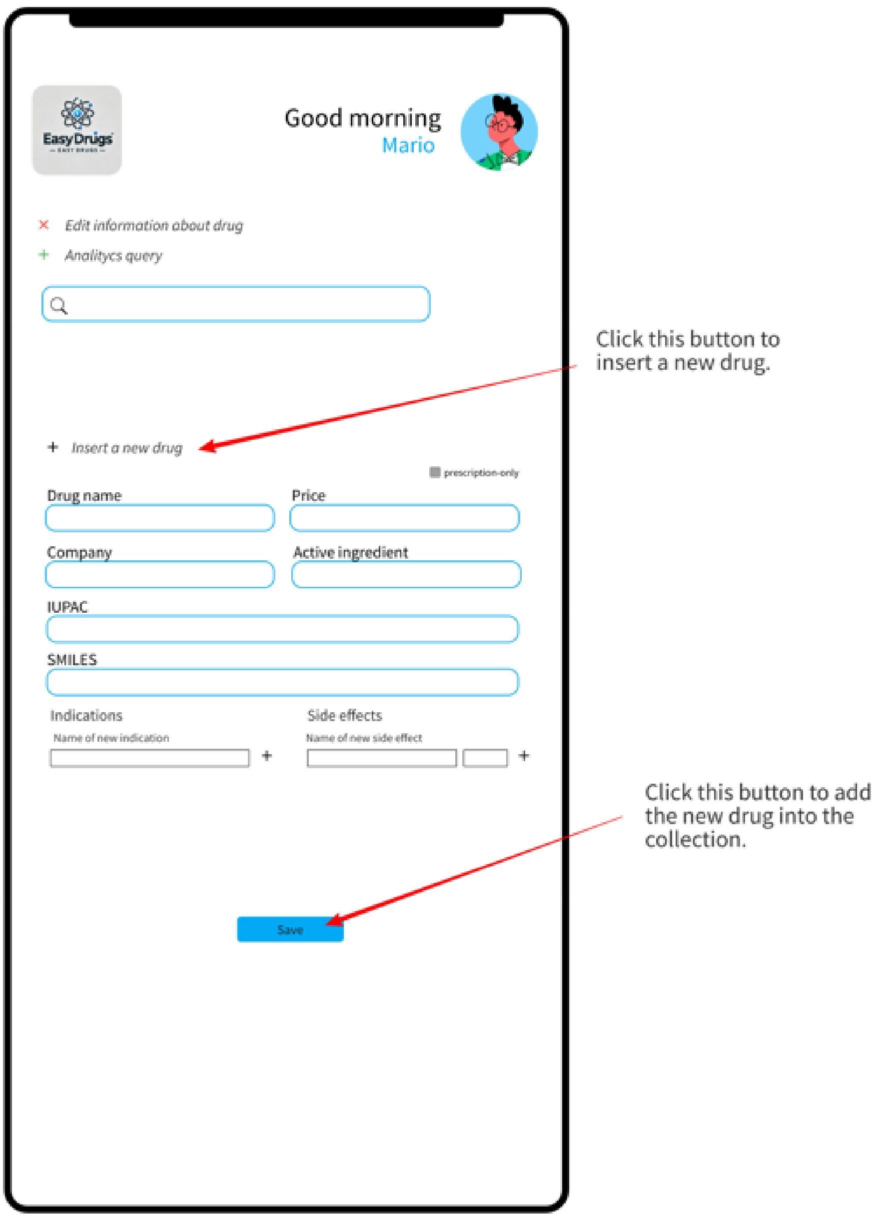


Figure 2.27: Adding a new drug

Top drugs purchased

There are three fields and a button to start the query. The user must select a start date, an end date, and a number, which defines how many top-purchased drugs should be displayed in the ranking.

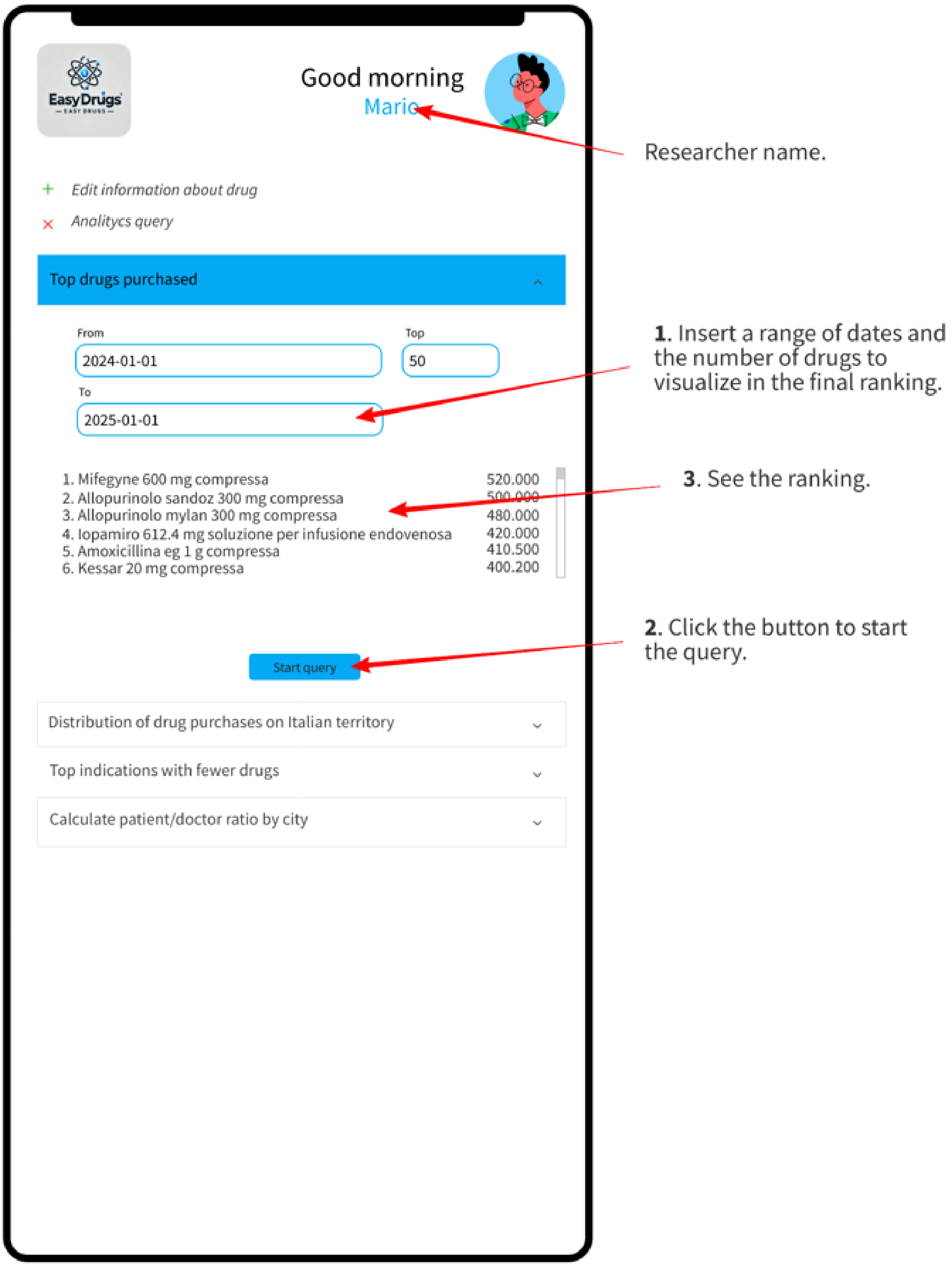


Figure 2.28: Top drugs purchased

Distribution of drug purchases on Italian territory

There are four fields and a button to start the query. The user must select a drug, a start date, an end date, and a sorting order (DESC for descending or ASC for ascending order). The query results will display a ranking of Italian regions, each showing the number of purchased units and the percentage relative to total sales

Figure 2.29.

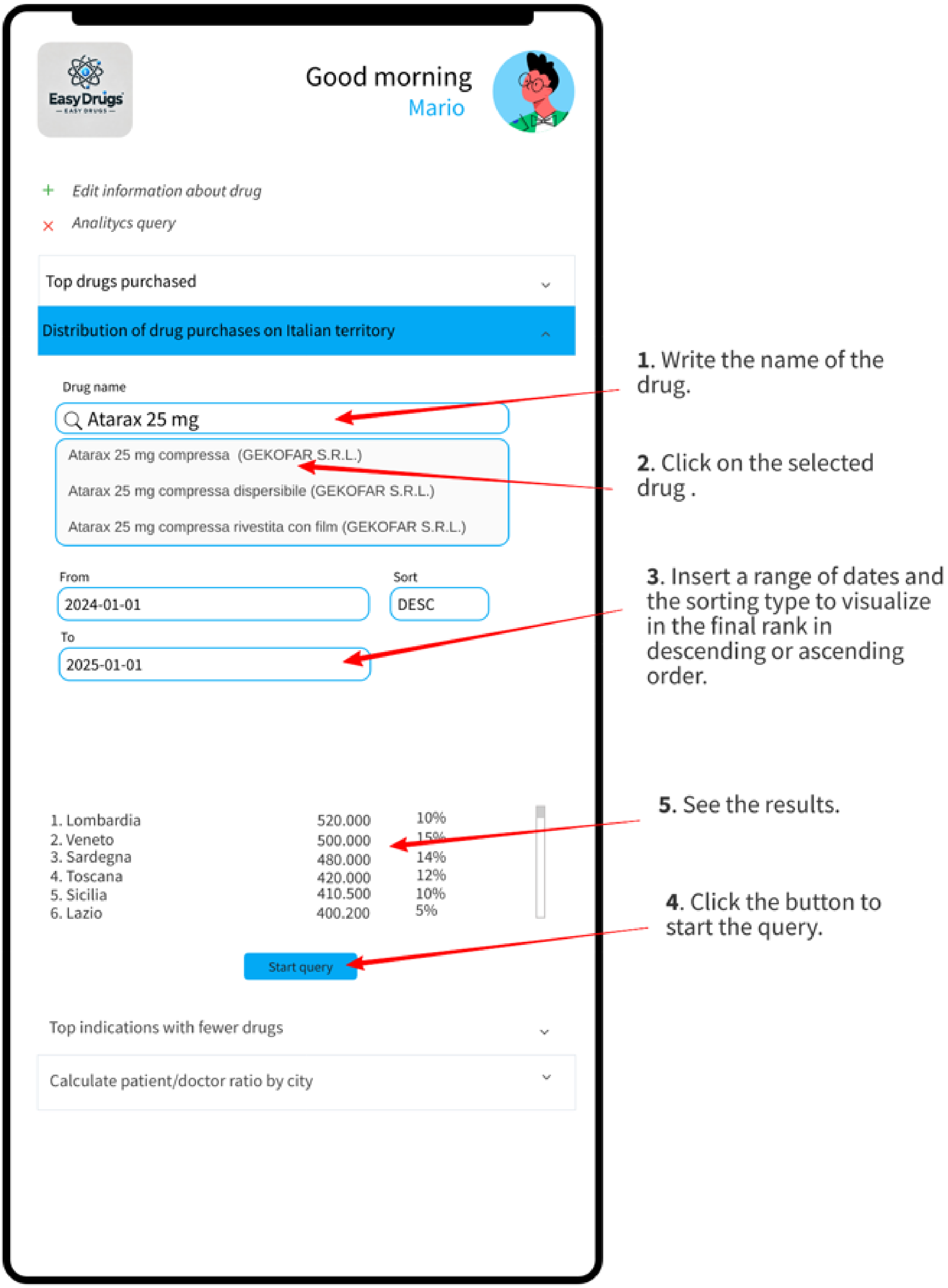


Figure 2.29: Distribution of drug purchases on Italian territory

Top indications with fewer drugs

There is one field and a button to start the query. The user specifies the number of top-ranking results to display. Clicking the start button generates a ranked list of indications with corresponding available drugs Figure 2.16. Each indication entry includes the number of related drugs and a plus icon that, when clicked, displays the drug names. Clicking the minus icon (which appears after the first click Figure 2.16) hides the additional information.

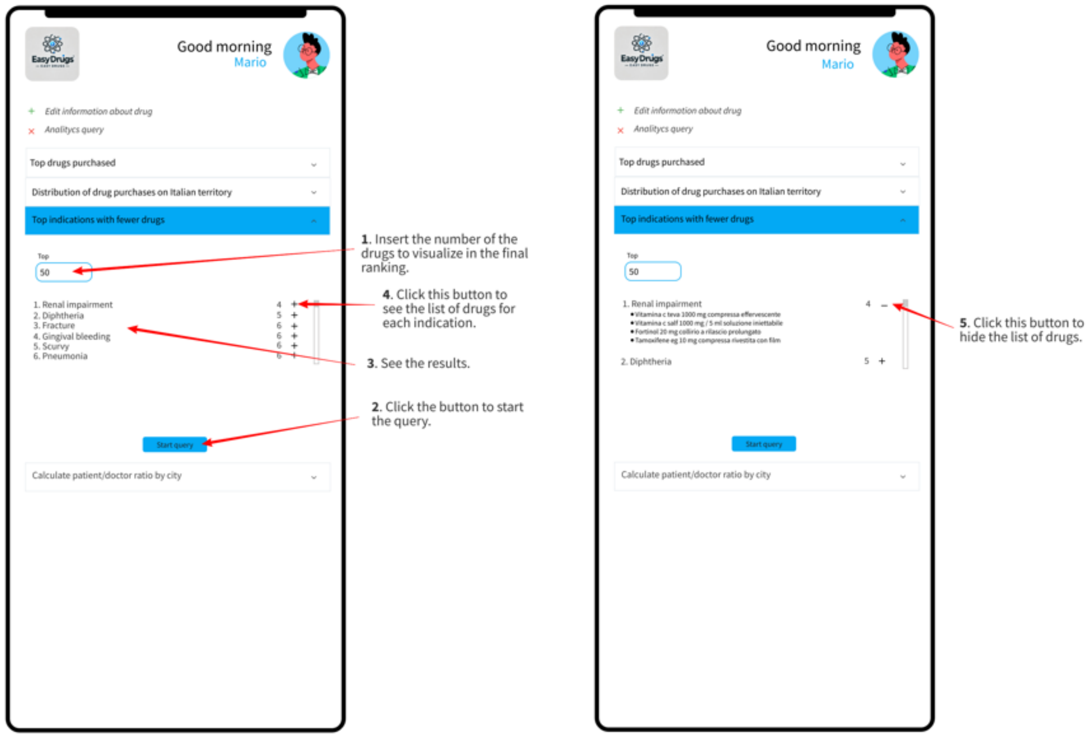


Figure 2.30: Top indications with fewer drugs

Calculate patient/doctor ratio by city

There is one field and a button to start the query. The user selects a sorting order (DESC for descending or ASC for ascending order), and clicking the button generates a ranked list of cities. Each city displays the ratio of patients to doctors, with higher values indicating a greater number of patients per doctor.

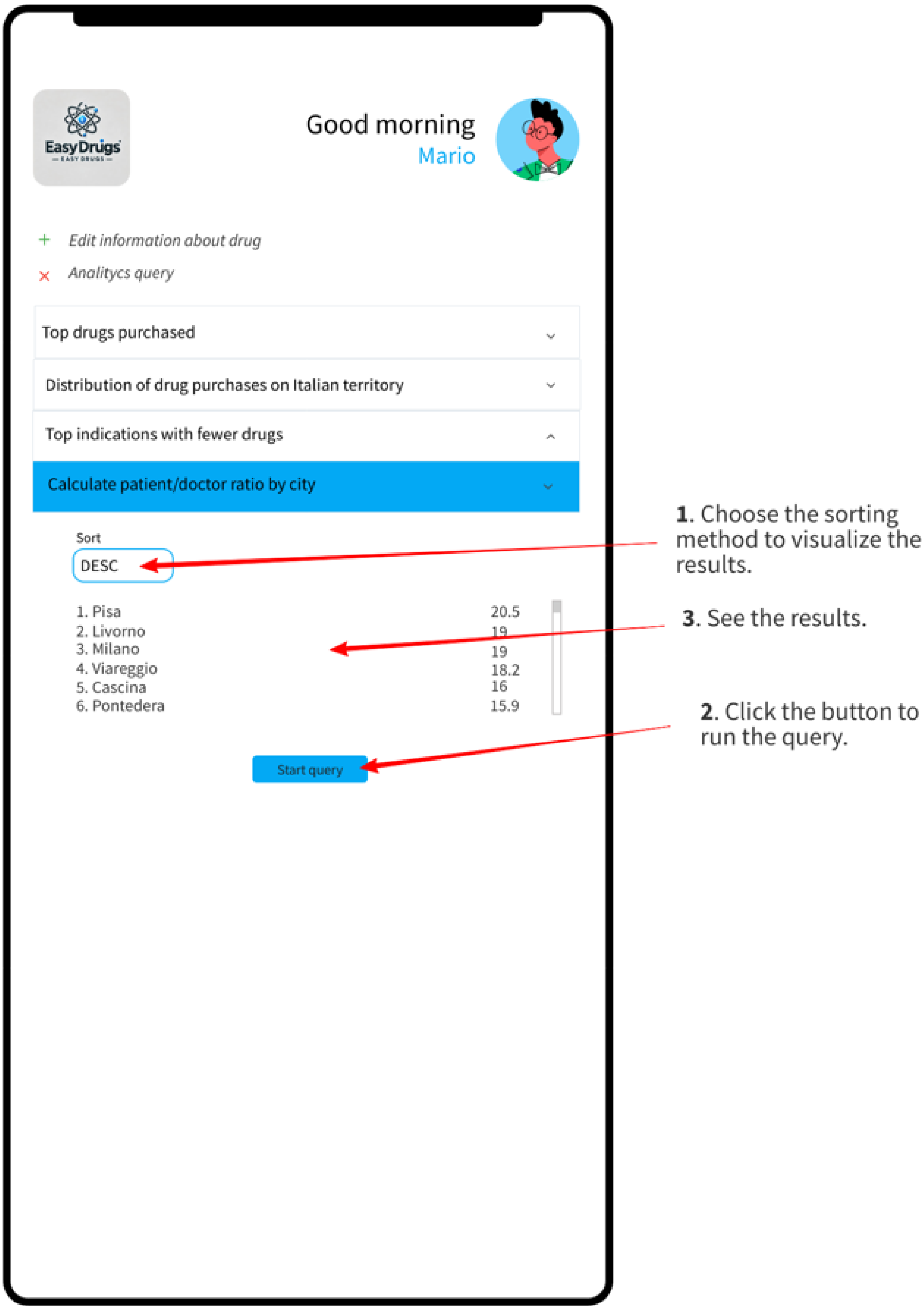


Figure 2.31: Calculate patient/doctor ratio by city

2.5 Distributed Database Design

2.5.1 Why AP?

The CAP theorem says that distributed databases can only guarantee two out of three properties: Consistency, Availability, and Partition Tolerance.
EasyDrug is a service that must always be available for all users. It would be unacceptable if a patient couldn't buy a prescribed drug, a doctor couldn't access a patient's medical history, or a pharmacy couldn't sell drugs to its customers.
Given these requirements, the choice is the AP side of the triangle, meaning we always ensure Availability and Partition Tolerance, while giving up Consistency at all times.
Luckily, consistency is not a major issue, and we can safely rely on eventual consistency. For example, it's not critical to immediately see new purchases or newly prescribed drugs. Also, drug updates are very rare.
E-commerce systems are usually designed with AP in mind, and the interactions between patients, doctors, and pharmacies are quite similar to that kind of application.

2.5.2 Replication

MongoDB Replication

To meet the non-functional requirement of high service availability, we have implemented a replication system across multiple virtual nodes, ensuring access to all data at any time, even in the event of a server failure.

The replicas are structured as follows:

- The primary node has the IP address 10.1.1.20
- The other two replicas have IP addresses 10.1.1.21 and 10.1.1.22

To initialize the Replica Set, a replication mechanism that provides redundancy and high availability for MongoDB databases, the following command is used:

```
rsconf = {
    _id: ‘‘lsmdb’’,
    members: [
        { _id: 0, host: ‘‘10.1.1.20:27017’’, priority: 6},
        { _id: 1, host: ‘‘10.1.1.21:27017’’, priority: 4},
        { _id: 2, host: ‘‘10.1.1.22:27017’’, priority: 0.5}
    ],
    settings: {
        setDefaultRWConcern : { w: 1 }
    }
};
```

```

rs.initiate(rsconf);
```

The idea is that the primary node is more likely to be chosen from the first two nodes, especially the first one, so that the majority of the application load is handled by these two. This allows the third node to serve as the master for Redis-related processes, which have a significant weight within the application.

The **Write Concern** defines how many copies of the data must be confirmed before a write operation is considered complete. This mechanism is crucial for balancing reliability and performance in write operations. The chosen value is $w = 1$ to maximize availability, ensuring that the user receives a response as soon as the changes are written to the primary node.

Below is the MongoDB connection URI, where the **Read Preference** parameter is defined. This parameter specifies the policy used to select the node from which to perform reads. In this case, queries are executed on the "nearest" node in terms of network latency, regardless of whether it is primary or secondary. This option helps reduce query latency.

```
mongodb://10.1.1.20:27017,10.1.1.21:27017,10.1.1.22:27017/EasyDrugDB?repl
    icaSet=lsmdb&readPreference=nearest
```

It is important to note that these parameters do not guarantee consistency between nodes at all times, but strict consistency is not a requirement for the application.

Redis Replication

A cluster composed of three nodes, has been implemented to ensure high availability for the data stored in the key-value database, organized as follows:

- One master node, with IP 10.1.1.22
- Two replica nodes, with IPs 10.1.1.20 and 10.1.1.21

All Redis access requests, under normal conditions, are directed to the master node. If the master node shuts down or becomes unreachable, one of the two replicas is promoted to master, replacing the failed node and continuing to handle both read and write requests. This mechanism ensures that all database data remains available at all times.

The configuration of each node is contained in the `redis.conf` file, located in the virtual machine corresponding to each address. This file is automatically created during the Redis installation in the `/etc/redis/` directory. Below is the only modification made (for the master server, as an example):

```
bind 10.1.1.22 -::1
```

The process of electing a new master is managed by sentinels. A sentinel is a process that monitors database nodes, continuously checking the status of both the master and replica nodes. Sentinels elect a replica as the new master in the event of a master failure and communicate this change to clients. To ensure a reliable failover mechanism, three sentinels were defined, as at least two votes are needed to elect a new master. This ensures that even if one sentinel fails, the remaining two can still reach a decision.

```java
@Configuration
public class RedisConfig {
    private JedisSentinelPool jedisSentinelPool;
    private final String sentinel1 = "10.1.1.20";
    private final String sentinel2 = "10.1.1.21";
    private final String sentinel3 = "10.1.1.22";

    @Bean
    public JedisSentinelPool jedisSentinelPool() {
        Set<String> sentinels = new HashSet<>();
        sentinels.add(sentinel1+":26379");
        sentinels.add(sentinel2+":26379");
        sentinels.add(sentinel3+":26379");
        jedisSentinelPool = new JedisSentinelPool("mymaster", sentinels);
        return jedisSentinelPool;
    }

    @PreDestroy
    public void shutdownJedis() {
        System.out.println("Chiusura delle connessioni Redis...");
        try {
            if (jedisSentinelPool != null) {
                jedisSentinelPool.close();
                System.out.println("JedisSentinelPool chiuso correttamente.");
            }
        } catch (Exception e) {
            System.err.println("Errore durante la chiusura di Redis: " + e.getMessage());
        }
    }
}
```

The configuration of each sentinel is as follows:

```
[sentinel.conf]
port 26379
```

```
sentinel monitor mymaster 10.1.1.22 6379 2
sentinel down-after-milliseconds mymaster 5000
sentinel failover-timeout mymaster 10000
```

The sentinel configuration file is also located in the `/etc/redis/` directory.

In the rare case that the master node fails, the sentinel-based election of a new master is not instantaneous. To handle this, every method that accesses the key-value store is annotated with `@Retryable`.

The `@Retryable` annotation in Spring is used to automatically retry a method in case of an exception, up to a defined maximum number of attempts. This gives sentinels enough time to elect a new master, ensuring that the method that initially failed will eventually reconnect and perform its operations successfully.

```
    @Retryable(
        retryFor = { JedisConnectionException.class },
        maxAttempts = 3,
        backoff = @Backoff(delay = 5000)
)
```

If an error occurs due to a failure to connect to the Redis server, the method will retry after 5 seconds. If the issue persists after three attempts, an exception is thrown.

2.5.3 Sharding

We analyzed whether sharding is an effective solution to enhance our system's performance by distributing access loads across multiple nodes in the Document DB.

Patients Collection

Sharding Key: Hashed on `doctorCode`
The sharding key is selected based on the `doctorCode` attribute, using a hashing strategy. This decision is motivated by the fact that doctors frequently search for their own patients. As a result, all patients associated with a specific doctor are stored on the same node, optimizing access efficiency.

Drugs Collection

Sharding Key: Range-Based on `drugName`
For the `drugs` collection, the most effective sharding key is range-based on the `drugName` attribute. This choice is justified by the frequent searches for drugs with names sharing a common prefix. Since this query is widely used by different types of

users, it is executed often. By partitioning shards using this sharding key, the load is evenly distributed across multiple nodes, and queries require access to a single server since drugs with the same prefix are stored in the same shard.

Purchases Collection

Sharding Key: Hashed on `region`
The `region` attribute is used as a hashed sharding key to balance access across nodes when users retrieve their purchase history. This approach ensures that access is limited to a single shard, as it is highly likely that a patient frequently purchases drugs within the same region.

Other Collections

For the remaining collections `doctors`, `researchers`, and `pharmacies` the most reasonable choice is to use a hashed sharding key based on the `id`.
By ensuring uniform distribution across shards, this strategy automatically balances the load across nodes. These collections support this type of sharding since documents can be randomly distributed, as there are no range-based queries or logical correlations among the data.

2.5.4 Handling inter database consistency

The use of two separate databases can introduce consistency issues between them. In particular, a critical operation in the system that requires write access to both the key-value database and the document database is the confirmation of purchases in the pharmacy's shopping cart. In this scenario, the information related to purchased drugs must be removed from the shopping cart stored in the key-value database and inserted into the document database. It is crucial that these two operations always occur together. If one operation is completed without the other, an inconsistency issue arises.
The purchase confirmation process is structured as follows:

1. All keys containing information about the drugs in the shopping cart are read from Redis.
2. The retrieved information is stored in different collections in MongoDB.
3. The previously read keys are either removed or modified in Redis.

The following possible scenarios are identified:

Interruption during drugs insertion into MongoDB

The insertion of purchased drugs involves two different collections: `patients` and `purchases`. This could lead to an inconsistent state if, for example, MongoDB were to crash, as it handles atomicity at the collection level. This issue is managed by adding the `@Transactional` annotation to the `insertPurchasesTransaction` method. This ensures that the entire set of operations within the function is executed atomically.

Successful MongoDB insertion but partial Redis modification

The Redis function is responsible for modifying the data in Redis by removing the purchased drugs from the user's cart and marking prescribed drugs as *purchased*. Since this function consists of multiple independent instructions, it may fail before completing, leading to inconsistencies both within Redis and between Redis and MongoDB. This issue is addressed using two mechanisms:

- Internal Inconsistency in Redis:
  To ensure atomicity within Redis, all operations are executed as a single atomic unit using Lua scripting [18]. Lua is integrated into Redis to enable atomic execution directly on the server.
  Additionally, using Append-Only File (AOF) persistence ensures that operations are logged. When executed via a Lua script, the operations are stored as a single atomic unit. Upon restart, Redis replays the AOF log, guaranteeing the re-execution of all operations, including the Lua script, restoring consistency.
  In a Redis cluster with replication, Lua's default configuration ensures that scripts propagate to all replicas while maintaining atomicity. By combining Lua scripting and AOF persistence, we ensure that Redis maintains internal consistency even in failure scenarios.

- Inconsistency with respect to Mongo DB:
  Since the MongoDB transaction has been successfully completed, all insertions are finalized. If Redis modifications fail, the inserted data in MongoDB must be rolled back to maintain consistency. To achieve this, the `insertPurchasesTransaction` function is modified to store information about the successfully inserted drugs IDs in a new MongoDB collection called `commit_log`. This allows for rollback operations, at a later time, if Redis modifications fail.

```
_id: ObjectId('67dd9a923277657c5a18e98e')
patientId : "PPDRVNT13A43L378D"
operationType : "DELETE"
▾ purchaseIds : Array (2)
    0: "67dd9a913277657c5a18e98c"
    1: "67dd9a913277657c5a18e98d"
timestamp : 2025-03-21T16:57:54.039+00:00
processed : true
```

Figure 2.32: Example of a commit log document

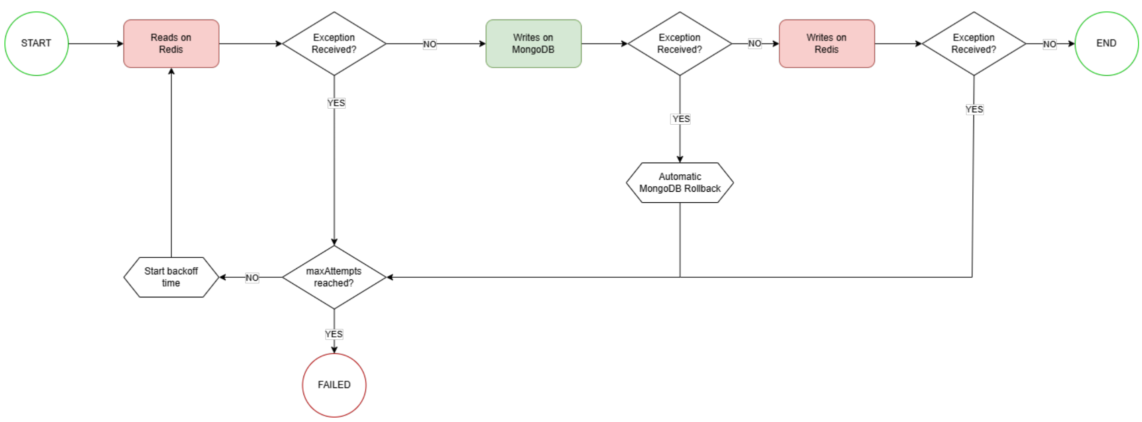
During this phase, a field `processed` is set to `false`, indicating that the purchase has not been fully completed.
However, rollback is not performed immediately upon catching an exception from Redis. Some errors, such as connection failures, may occur after the script is sent to the Redis server but before the response is received, making it unclear whether the modifications were applied. Thus, rollback is postponed and handled asynchronously. A scheduled method executes periodically, scanning all unprocessed logs (i.e., those with `processed = false`). It verifies whether the transaction was correctly completed; if not, it performs a rollback using the stored log information, ensuring eventual consistency between the databases.

This mechanism relies on the Lua script in Redis, which, as its final operation, inserts a key in Redis of the form `log:id_log`, where `id_log` is the unique identifier of the corresponding entry in `commit_log` on MongoDB. The periodic script then checks the existence of this key in Redis:

- If the key exists in Redis: The purchase operation was successful, and the attribute `processed` is set to `true` in the log.
- If the key does not exist in Redis: The Redis operation was not completed, or the master node crashed before propagating changes to replicas or appending the Lua script execution to the AOF log. In this case, a rollback is performed to remove the purchased drugs from MongoDB and restore consistency between the two databases, updating `processed` to `true`.

To optimize this process, an index is added on the `processed` field within the `commit_log` collection. This ensures that retrieving unprocessed logs is efficient without scanning the entire collection.

Finally, the Retry mechanism is implemented: if the operation fails due to one of the exceptions specified in the annotation, the method is retried up to 3 times, with a 2-second delay between each attempt. This mechanism might result in duplicate customer purchase entries in the MongoDB collections. However, this is not an issue, as it would only occur in case of a failure with Redis. Any excess entries will be automatically removed the next time the periodic realignment function runs.



```java
@Retryable(  1 usage  supersamu2001 +2 *
        retryFor = { DataAccessException.class, TransactionSystemException.class,
                     RetryException.class },
        maxAttempts = 3,
        backoff = @Backoff(delay = 2000)
)
public LatestPurchase confirmPurchase(String id_pat, String id_pharm) {
    if(id_pat == null || id_pat.isEmpty())
        throw new BadRequestException("The patient id can not be null");

    Jedis jedis = null;
    CommitLog log = new CommitLog();
    NewPurchaseDTO newPurchaseDTO;

    try {
        // Lettura delle informazioni dei farmaci da comprare
        ConfirmPurchaseCartDTO confirmPurchaseCartDTO = purchaseCartRedisRepository.confirmPurchaseCart(id_pat);

        List<PurchaseCartDrugDTO> purchasedDrugs = confirmPurchaseCartDTO.getPurchaseDrugs();
        // eseguiamo la transazione atomica di MongoDB
        newPurchaseDTO = insertPurchasesTransaction(id_pat, id_pharm, purchasedDrugs, log);

        // Eseguiamo le modifiche su Redis in modo atomico utilizzando uno script Lua
        confirmPurchaseCart(id_pat, confirmPurchaseCartDTO, log.getId());
        jedis = confirmPurchaseCartDTO.getJedis();

        return newPurchaseDTO.getLatestPurchase();

    } catch (JedisException e) {
        // non viene effettuato il rollback perché non abbiamo la sicurezza che le informazioni
        // su redis non siano state eseguite realmente
        throw new RetryException(e.getMessage());

    } finally {
        // viene restituito il pool di connessione
        if(jedis != null)
            jedis.close();
    }
}
```

Chapter 3

Data modelling

3.1 Dataset building and Web scraping

In the process of constructing a database, an important step is obtaining a sufficient amount of data to test how the application will function and adapt to the designed architecture (as discussed later). This step can also influence the design, as it may reveal infeasible implementations. The final result of this process consists of six JSON files with the following sizes:

- `doctor.json`: 3 MB
- `drugs.json`: 3 MB
- `patient.json`: 89.3 MB
- `pharmacy.json`: 1.8 MB
- `purchase.json`: 94.7 MB
- `researcher.json`: 0.62 MB

Thus, the total size is **192.42 MB**. At the end of this step, the JSON files are imported into MongoDB.

To generate these files, two Python scripts from different platforms were used:

- `dbBuild.ipynb` from Google Colab
- `drugScrap.ipynb` from Anaconda

This choice was made because Google Colab does not support Selenium (a library used for web scraping). Datasets and reports were used to generate realistic data. Unfortunately, they do not refer to the same year, but they were the only available combination of sources capable of producing realistic data while aligning with the project's objectives.

drugs.json

The official AIFA website has a specific section for browsing drugs [6] distributed across the Italian territory. Since the site does not offer an API or dataset to obtain the data, a Python algorithm was used to scrape it (`drugScrap.ipynb`). The code browses active ingredients, obtained by combining datasets from the Italian Ministry of Health Open Data [20] and AIFA Open Data [7], and retrieves the first five drugs.

For each drug, the following information is retrieved: the name on the packaging, the company that owns it, the active ingredient, the ATC and AIC codes, and whether it requires a prescription.

In this phase, 1,778 drugs were collected.
The AIC code is a unique identifier for all drugs in Italy, while the ATC code uniquely identifies the clinical area for which the drug is intended (thus, some drugs from different agencies may have the same ATC code because they are used for the same disease, but they will have different AIC codes).
Additionally, datasets from AIFA Open Data, which associate AIC codes with drug prices, were downloaded.
Moreover, datasets on drug indications and side effects were downloaded from Sider [13], identified by ATC codes.

After downloading all the files, different parts of `dbBuild.ipynb` were executed to build the JSON files with the designed structure. For each drug, a maximum of ten indications and ten side effects were associated. Drugs without indications were discarded. IUPAC and SMILES definitions were retrieved using GET requests from PubChem [15]. The family name of each drug was extracted from its packaging name to group them for future queries.
Occasionally, drugs with no listed price were found. In these cases, a price was generated based on reports from ISTAT [10], Federfarma [8], and AIFA [4], which provide information on how to calculate the average price of drugs for each first-level ATC category (with a random value added in the range of 0-1€).
Some data sanitization was applied to the file, such as converting the literal frequency of side effects (e.g., "very frequent", "frequent") into numeric values and renaming attributes like "_id" to be used as the ID field in MongoDB.

Account area

Two cells of Google Colab were used to create accounts for `patient.json`, `researcher.json`, `pharmacy.json`, and `purchase.json`. The first one generates `Italia.json`, which contains information on the population and doctor distribution across the Italian territory (specifically regions and cities). This information was obtained from AGENAS [2] and ISTAT [11].
The second cell creates 50,000 accounts: 12,500 doctors, 1,500 researchers, and 36,000 patients. Additionally, around 12,500 pharmacy accounts were generated.

For each region, a certain number of doctors was created, each assigned a random city based on population probability (for example, it is more likely for five doctors to be from Florence than from Peccioli). It also generates doctors' patients and real existing pharmacies, based on Datiopen.it [12].
Names were assigned probabilistically using a dataset containing the most common names in Italy [22], while surnames were randomly selected from another dataset containing the most common ones [21]. The age (i.e., possible year of birth) was generated using a report from AdminStat Italia [1], which provides the age distribution of the Italian population (ensuring that doctors and researchers are at least 34 years old).
Passwords were hashed using bcrypt.

purchase.json

Thanks to AIFA open data, a classification of the most used drugs in Italy, based on the number of units distributed, was created. This classification, merged with the actual drugs in `drug.json`, resulted in a "Top 70" list. Then, in the Top 70 list, a field to classify drugs by age and gender was manually inserted as follows (so that, for example, arthrosis drugs are not assigned to children): *for adults, for children and elderly people, for elderly people, for men, for women, for children.*
The main script iterates through patient accounts and randomly generates purchased drugs for the years 2024-2025, based on the patient's age, according to the following table Table 3.1.

Table 3.1: Probable amount of drugs purchased in 2024-2025

Age range	**Total drugs purchased**
$3-14$	$0-8$
$15-25$	$0-3$
$26-35$	$1-6$
$36-55$	$2-10$
$56-65$	$5-10$
>64	$8-15$

First, the purchase date is generated, followed by the prescription date, which is at most 10 days earlier (for OTC drugs, a `Null` value is assigned instead of the date). The quantity of each drug has a 50% probability of being 1 unit, 45% of being 2 units, and 5% of being 3 units. Drugs are grouped into purchases of different "sizes," chosen randomly.
There is a small probability that a drug not in the Top 70 will be randomly selected from `drugs.json`. Single drug purchases are added to `purchase.json`, and the last five purchases are stored in the `latestPurchaseDrugs` attribute of the corresponding

patient account. `DrugId` values are added to specific attributes of the patient for document linking (see later).

3.2 MongoDB

3.2.1 Collections

We chose MongoDB for its advantage in storing related information within a single document rather than distributing it across multiple tables, optimizing most of the queries required by our system. Additionally, its horizontal scalability across multiple servers, leveraging sharding and replication, ensures high availability, which is an essential non-functional requirement.
The collections that make up our MongoDB database are as follows:

- Patients
- Doctors
- Pharmacies
- Researchers
- Drugs
- Purchases

Patient

Below is an example of a document from the patients collection:

```
_id: "PZDDSFN61S03L378A"
password : "$2a$12$.ufLfEeXFvXwckagJgN9pO.KYPibpZsOdq3fEZDyP64lR23UzPxQy"
city : "trento"
district : "trento"
region : "trentino alto adige"
name : "stefano"
surname : "zuddas"
dateOfBirth : "1961-11-20"
gender : "m"
taxCode : "ZDDSFN61S03L378A"
doctorCode : "DVNNCLL98M43L378G"
▾ latestPurchasedDrugs : Array (5)
  ▾ 0: Object
      timestamp : 2025-01-14T17:07:00.000+00:00
    ▾ drugs : Array (1)
      ▾ 0: Object
          drugId : ObjectId('67aba9215da6705a000d402c')
          drugName : "canesten 0.3 g crema"
          quantity : 1
          price : 4.7
  ▸ 1: Object
  ▸ 2: Object
  ▸ 3: Object
  ▸ 4: Object
▾ purchases : Array (12)
    0: ObjectId('67aba9225da6705a000d457a')
    1: ObjectId('67aba9225da6705a000d457b')
    2: ObjectId('67aba9225da6705a000d4577')
    3: ObjectId('67aba9225da6705a000d4578')
    4: ObjectId('67aba9225da6705a000d457c')
    5: ObjectId('67aba9225da6705a000d457d')
    6: ObjectId('67aba9225da6705a000d4580')
    7: ObjectId('67aba9225da6705a000d4579')
    8: ObjectId('67aba9225da6705a000d4575')
    9: ObjectId('67aba9225da6705a000d4576')
    10: ObjectId('67aba9225da6705a000d457e')
    11: ObjectId('67aba9225da6705a000d457f')
▾ prescriptions : Array (4)
    0: ObjectId('67aba9225da6705a000d4578')
    1: ObjectId('67aba9225da6705a000d4579')
    2: ObjectId('67aba9225da6705a000d4575')
    3: ObjectId('67aba9225da6705a000d4576')
```


Figure 3.1: Example of a patient document

The field _id represents the patient's identifier, composed of the letter 'P' followed by their tax code. Key aspects to highlight in this document include the *latestPurchasedDrugs* array, which contains the patient's five most recent purchases. Each entry consists of a timestamp and an array of purchased drugs, along with their key information. This field is used to fulfill the functional requirement that specifies displaying the patient's five most recent purchases, under the assumption that users are more interested in their latest transactions rather than their entire purchase history. For this reason, we opted for document embedding to avoid accessing the *purchases* collection, thereby improving query execution efficiency.
However, if the patient wishes to view older purchases, they can retrieve them in batches of 10 using the *purchases* array, which contains the _id of each purchased drug. This enables access to the *purchases* collection via document linking. The

array stores IDs in chronological order, with newly purchased drugs appended to the end. This approach eliminates the need to sort the array every time it is displayed chronologically, thus enhancing efficiency.
We did not embed all purchased drugs because, given that the application does not remove purchases from the document after a certain period, this could lead to a significant increase in document size and, consequently, in the overall collection size. The same reasoning applies to *prescriptions*, an array containing the _id of prescribed drugs that were later purchased. This array is used to display the list of prescribed drugs bought by the patient, allowing for faster access than querying the *purchases* array, which also includes OTC drugs that should not be retrieved in this case.
Finally, *DoctorsCode* contains the _id of the patient's family doctor.

Doctor

Below is an example of a document from the doctors collection:


```
_id: "DVNNCLL98M43L378G"
password : "$2a$12$B4iVINGp7vlYRXUbusm2qekXe.mrR9tCKvKu5t0TA949zOnf/ADxm"
city : "trento"
district : "trento"
region : "Trentino Alto Adige"
name : "camilla"
surname : "vanini"
dateOfBirth : "1998-08-04"
gender : "f"
taxCode : "VNNCLL98M43L378G"
doctorRegisterCode : "21552"
```


Figure 3.2: Example of a doctor document

The doctors collection primarily contains personal data. The only thing to highlight is that the identifier code in this case is composed of 'D' followed by the doctor's tax code.

Pharmacy

Below is an example of a document from the pharmacies collection:


```
_id: "Ph80106982737"
name : "bolghera"
password : "$2a$12$lVVb.th4WiewPYgjVpIzE.8L1fYf/MXarU7.uKjunzHlUWx5L.I/K"
VATnumber : "80106982737"
address : "largo medaglie d'oro 8"
city : "trento"
district : "trento"
region : "trentino alto adige"
ownerTaxCode : "PGLVTR62R43L378P"
```


Figure 3.3: Example of a pharmacy document

The pharmacy collection primarily contains data related to the company. The only thing to highlight is that the identifier code in this case is composed of 'Ph' followed by the company's VAT number.

Researcher

Below is an example of a document from the researchers collection:

```
_id: "RRSLLEA99P43C850R"
password : "$2a$12$/.5OkeWlYm0xrg9EsNdnb.wm1tY6CoFzy3sLG5PpVP2Gc2JpKrOmO"
city : "collebeato"
district : "brescia"
region : "lombardia"
name : "lea"
surname : "orsolo"
dateOfBirth : "1999-09-04"
gender : "f"
taxCode : "RSLLEA99P43C850R"
researcherRegisterCode : "77406"
```

Figure 3.4: Example of a researcher document

The researchers collection also primarily contains personal data. The only point to highlight is that the identifier code in this case is composed of 'R' followed by the researcher's tax code.

All of these collections have an _id that is not the automatic one assigned by MongoDB upon insertion. Instead, the _id is defined before the document is inserted into the collection. This means that before inserting a document, it must be verified that no other document with the same _id already exists in the collection. This check is necessary regardless, as a fundamental requirement prohibits patients, doctors, and researchers from registering more than once with the same tax code and role. The same rule applies to pharmacies: no two pharmacies can have the same VAT number.
As can be observed, the decision to separate users into different collections was made because there is no query that simultaneously operates across different user types. Therefore, creating a single users collection could lead to query inefficiencies, such as when searching exclusively for patients without including other user types.

Drug

Below is an example of a document from the drugs collection:

```
_id: ObjectId('67a7702c189d387b9f61f952')
drugName : "mifegyne 600 mg compressa"
price : 18.58
company : "exelgyn"
activeIngredient : "mifepristone"
IUPAC : "(8S,11R,13S,14S,17S)-11-[4-(dimethylamino)phenyl]-17-hydroxy-13-methyl…"
SMILES : "CC#CC1(CCC2C1(CC(C3=C4CCC(=O)C=C4CCC23)C5=CC=C(C=C5)N(C)C)C)O"
onPrescription : true
▾ indications : Array (6)
    0: "foetal death"
    1: "dead foetus"
    2: "cushing's syndrome"
    3: "type 2 diabetes mellitus"
    4: "hyperglycaemia"
    5: "glucose tolerance impaired"
▾ sideEffects : Array (10)
  ▾ 0: Object
      sideEffectName : "anorexia"
      frequency : 10
  ▾ 1: Object
      sideEffectName : "decreased appetite"
      frequency : 10
  ▸ 2: Object
  ▸ 3: Object
  ▸ 4: Object
  ▸ 5: Object
  ▸ 6: Object
  ▸ 7: Object
  ▸ 8: Object
  ▸ 9: Object
```

Figure 3.5: Example of a drug document

This collection uses MongoDB's automatic _id of type ObjectId . It stores information about the drug, including an indications array, which lists the diseases the drug is prescribed for. Additionally, the collection includes a sideEffects array, containing documents that specify the name of each possible side effect and its occurrence frequency, expressed as a percentage. Additionally, the collection includes the "onPrescription" field, a boolean that indicates whether the drug requires a prescription or is available over the counter (OTC).

Purchase

Below is an example of a document from the purchases collection:

```
_id: ObjectId('67aba9d55da6705a000f5362')
drugId : ObjectId('67aba9215da6705a000d3fda')
name : "proctolyn 0.1 mg ,10 mg supposta"
quantity : 1
purchaseDate : 2024-01-01T10:00:00.000+00:00
region : "lombardia"
prescriptionDate : 2023-12-25T17:35:00.000+00:00
price : 27.1
```

Figure 3.6: Example of a purchase document

Each document represents a purchased drug and has a unique _id of type ObjectId, automatically managed by MongoDB. This field is also used when retrieving a patient's purchased drug, achieved through document linking, as previously mentioned. The prescriptionDate field may be absent if the drug was purchased without a prescription. The region field indicates the region of the pharmacy where the patient made the purchase.

Why separate collection?

The decision not to create a single collection containing all users but instead to generate a specific collection for each user type is based on the fact that no queries involve multiple user types simultaneously. In our system, every query is always related to a single, specific category of users.
If all users were stored in a single collection, accessing the document database would be slower and less efficient, as different types of documents never requested together would be stored in the same disk block. This would lead to unnecessary data reads, retrieving information that would not be used.
For what concerning the rapid growth of documents, the designed structure doesn't lead to any problem. Patient collection is the only one that may lead to issues because it has two arrays of ids per documents, but the evolution of a single one is very slow (about 12 ObjectIds per year for each array).

3.2.2 Indexes

Drugs Collection

```
@CompoundIndex(def = "{'drugName': 1, 'onPrescription': 1}")
```

The compound index on `drugName` and `onPrescription` speeds up the following queries:

- Visualize a list of drugs given a prefix of their name.
- Visualize a list of OTC drugs given a prefix of their name.
- Visualize a list of prescribable drugs given a prefix of their name.

Let's take the following operation as an example:

```
db["drugs"].find({ drugName : { $regex: '^caspofungin'}, onPrescription : true })
.explain("executionStats");
```

As shown in the figures below, the number of documents examined has significantly decreased with the addition of the index.

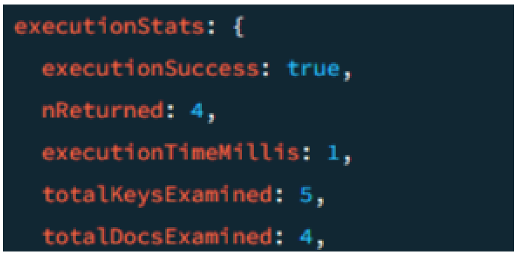


Figure 3.7: With index

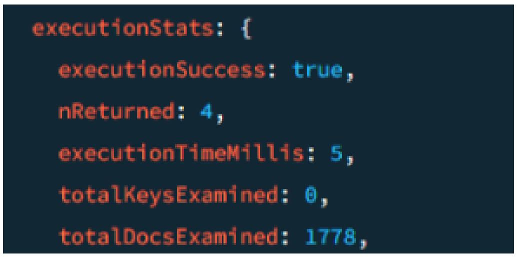


Figure 3.8: Without index

The drugs collection is frequently accessed for reading, as all users need to view a list of drugs to perform their designated functions. In contrast, write operations are much less frequent. Adding a new drug is a rare occurrence, while deletions and modifications are even rarer. Therefore, the presence of an index does not significantly impact system performance.

Patient Collection

```
@CompoundIndex(def = "{'doctorCode': 1, 'surname': 1}")
```

The compound index on `doctorCode` and `surname` speeds up the following query:

- The system must allow the doctor to visualize a list of patients given a prefix of their surname.

Let's take the following operation as an example:

```
db.patients.find({ 'doctorCode': 'DVNNCLL98M43L378G',
'surname' : { $regex: '^zudd' }}).explain("executionStats")
```

In this case, with the help of the index, only one document is examined, compared to over twenty thousand when the index is not used.

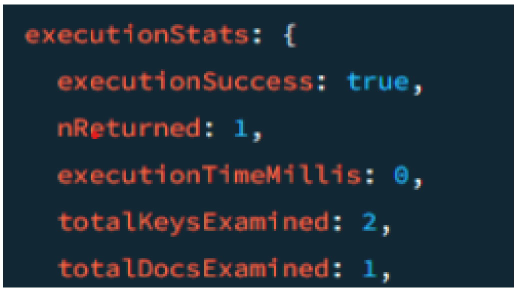


Figure 3.9: With index

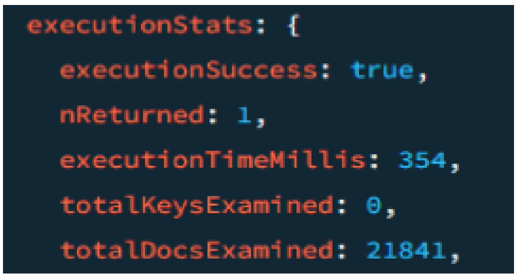


Figure 3.10: Without index

Similarly to the previous case, for the patient collection, read operations are substantial, while write operations are much rarer. This is because we consider the system in its steady-state operation, meaning it has been running for a long time, and the addition of new users is relatively infrequent. Therefore, for the same reason, introducing an index is beneficial.

Purchase Collection

```
@Indexed({'purchaseDate': 1})
```

The index on "purchaseDate" speeds up the following queries:

- Visualize the most purchased drugs in a given period
- See percentage and number of purchase of drug for every region, in a given period

Let's take the following operation as an example:

```
db.purchases.find({ purchaseDate: { $gte: ISODate("2024-01-01T10:00:00.000Z"),
$lte: ISODate("2024-01-31T10:00:00.000Z") },
drugId: ObjectId('67aba9215da6705a000d3fda')}).explain("executionStats")
```

In this case as well, the number of documents analyzed decreases significantly. The smaller the search time range, the more efficient the index becomes, as fewer documents need to be scanned.

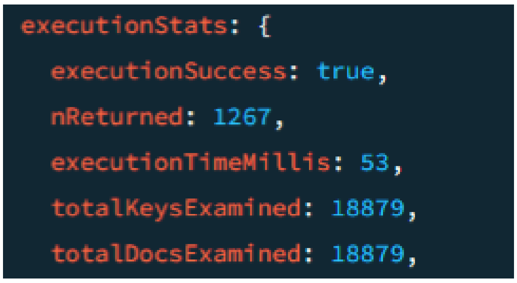


Figure 3.11: With index

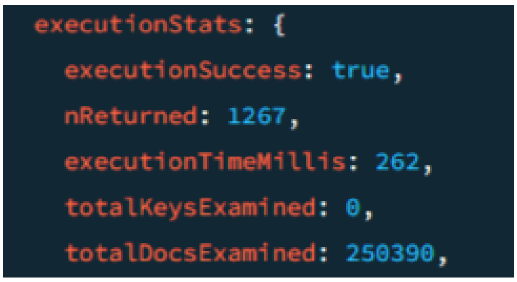


Figure 3.12: Without index

In this case, write operations are frequent, but introducing the index is necessary to prevent analytics involving date ranges from becoming too slow. Indeed, since purchases will never be deleted in our application, serving as an archive that stores data from the system's launch, the purchases collection will grow increasingly large over time. Therefore, using an index on the date is essential to avoid scanning the entire collection for every researcher's analytics query.

3.3 Redis

The application needs to handle the following customer carts: prescription, active prescription and purchase cart. The concept is similar to a restaurant booking application, but in this case, a patient "books" a list of drugs with his doctor using the prescription and active prescription carts. The first one helps the doctor manage a specific prescription before definitively inserting it into the system. Then, the patient can complete the "reservation" with the purchase cart at the pharmacy (the restaurant).
These three carts are accessed very frequently because, in a common healthcare system, there is a huge number of active patients visiting pharmacies and doctors, unlike in a restaurant scenario. This confirms why it is important to manage everything correctly and as quickly as possible.
As stated in the non-functional requirements, elements in the cart must have timeouts to reduce database load and prevent trashing.
Choosing a key-value database is essential because data insertion is temporary. In particular, an in-memory database like Redis ensures fast data access and low query response time.

3.3.1 E-R diagram

The following image is the entity-relation diagram from which the keys are structured.

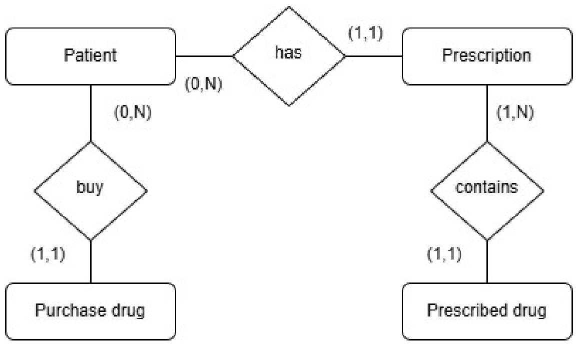


Figure 3.13: Entity-relation diagram for Redis design

A patient can buy different drugs (purchase drug entity) and add them to the purchase cart. A patient may also have multiple prescriptions given by the doctor, which can be represented in the active prescription entity or the prescription entity, as mentioned earlier, depending on the cart to which they belong. Each prescription has a list of prescribed drugs. From the four entities in the diagram, only three will be represented in the K-V DB because the patient does not have any relevant attributes, and there are no queries in Redis that access patient data. The identity code is fundamental for fetching the correct cart of a patient. Prescription and prescribed drugs will be linked by the prescription ID.

3.3.2 Namespace

There are three groups of keys, one for each entity in the E-R: pres, pres-drug, and purch-drug. The Patient entity has no attributes of interest, and there are no queries used to retrieve information about it. For this reason, it doesn't have keys related to him.

3.3.3 Keys Design

These are the keys related to patient prescriptions:

- *pres:id_pres:id_pat:timestamp*
- *pres:id_pres:id_pat:toPurchase*

A prescription is identified by an `id_pres`, a unique incremental numeric identifier stored in the database. `id_pat` identifies the patient using their identity code and maintains the 1:N relationship with the patient entity.
The `timestamp` contains an ISO-formatted date string only when the prescription is active. When inactive, it contains an empty string representing the patient's prescription cart (not yet activated).
Using an empty string is preferable to leaving it unset because the secondary key (`...toPurchase`) is created only after prescription activation. This key contains the quantity of drugs to be purchased from the prescription.
Why create it after activation? Because the quantity needs updating whenever drugs are added to or removed from the prescription cart (based on the doctor's decisions). In this case, the count will reflect only the drugs present when the doctor activates the prescription for the patient.
These are the keys related to prescribed drugs:

- *pres-drug:id_pres-drug:id_pres:id*
- *pres-drug:id_pres-drug:id_pres:info*
- *pres-drug:id_pres-drug:id_pres:purchased*

.
A prescribed drug is identified by `id_pres-drug`, a unique incremental numeric identifier stored in the database. `id_pres` identifies the prescription and maintains the 1:N relationship with the prescription entity.
The `info` attribute contains a JSON object structured as follows:

{ "name": "OKI", "price": 2.56, "quantity": 2 }

The idea is that this information is always used together, making it more efficient

to perform a single read or write operation instead of three. Quantity may be an exception, as users editing a drug's quantity typically don't need to modify other attributes. However, such operations are rare and usually correlate with user errors. The `purchased` flag indicates whether the prescribed drug has been purchased (meaning it was added to the purchase cart and the transaction was successfully completed).
Why not simply remove it from pres_drug? A prescription may include multiple drugs, and if only one is purchased, the prescription must remain active to show which drugs have been purchased and which haven't.
These are the keys related to patient purchases:

- *purch-drug:purch-drug_id:id_pat:id*
- *purch-drug:purch-drug_id:id_pat:info*

A drug in the purchase cart is identified by `id_purch-drug`, a unique incremental numeric identifier stored in the database. `id_pat` serves the same purpose as previously described and will be used as an input parameter for all key-value queries.
Drug information is stored in a JSON object (for the same reasons as `pres_drug`):

{ "name": "OKI", "price": 2.56, "quantity": 2, "id_pres": 13, "id_pres_drug": 45}.

3.3.4 Key life cycle

- `pres`: Keys related to the prescription cart will be deleted after one day, and keys related to active prescriptions will be deleted one month after activation.
- `pres-drug`: Keys related to drugs in the prescription cart will be deleted after one day, and keys related to prescribed drugs will be deleted after one month.
- `purch`: Keys will be deleted after one hour.

3.3.5 Cart optimization

The current key structure efficiently retrieves patient information from drugs, but the reverse operation - obtaining all drugs from a patient's cart (a frequently used application feature) - requires iterating through all possible keys matching the pattern:

purch-drug:purch-drug_id:id_pat:id

Here, `id_pat` remains constant while `purch_drug_id` is an incremental identifier, forcing the system to verify each key's existence using the `EXISTS` command. While `EXISTS` is a lightweight Redis operation, the main issue isn't the individual check

but rather the massive number of required requests.
This approach presents two fundamental criticalities:

1. **Network Overload** - Each `EXISTS` call generates a separate request to the Redis server, significantly increasing network traffic.

2. **Reduced Scalability** - As the number of patients and drugs in carts grows, the volume of calls increases, degrading overall service performance.

When implementing this solution and initializing the key-value store with 150 patients (each having an active purchase cart containing 2 drugs), the measured response times were as follows:

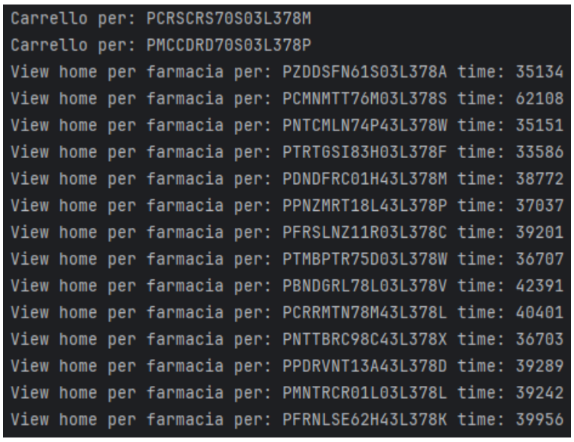

```
Carrello per: PCRSCRS70S03L378M
Carrello per: PMCCDRD70S03L378P
View home per farmacia per: PZDDSFN61S03L378A time: 35134
View home per farmacia per: PCMNMTT76M03L378S time: 62108
View home per farmacia per: PNTCMLN74P43L378W time: 35151
View home per farmacia per: PTRTGSI83H03L378F time: 33586
View home per farmacia per: PDNDFRC01H43L378M time: 38772
View home per farmacia per: PPNZMRT18L43L378P time: 37037
View home per farmacia per: PFRSLNZ11R03L378C time: 39201
View home per farmacia per: PTMBPTR75D03L378W time: 36707
View home per farmacia per: PBNDGRL78L03L378V time: 42391
View home per farmacia per: PCRRMTN78M43L378L time: 40401
View home per farmacia per: PNTTBRC98C43L378X time: 36703
View home per farmacia per: PPDRVNT13A43L378D time: 39289
View home per farmacia per: PMNTRCR01L03L378L time: 39242
View home per farmacia per: PFRNLSE62H43L378K time: 39956
```

Figure 3.14: Initial response time

The minimum response time to load the cart of a specific patient is 35 seconds, with only 150 active carts. This is a very high value, considering the potential scenario of having millions of keys at the same time in Italy along with active prescriptions. The service must not maintain such high response times, so some redundancies are introduced in the key-value structure:

- *pres:id_pat:set*
- *pres-drug:id_pres:set*
- *purch-drug:id_pat:set*

These keys represent a data structure that memorizes groups of unique and unordered elements. Respectively:

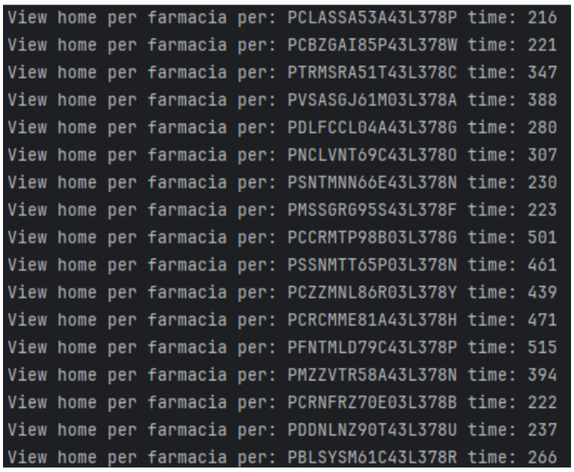
- “id_pres”, IDs of patient’s prescriptions.
- “id_pres_drug”, IDs of drugs from a specific prescription of a specific patient.

- "id_purch_drug", IDs of drugs inside a purchase cart of a specific patient.

These redundancies allow reading directly used keys, avoiding checking for non-existing or unimportant keys (from other patients). The advantages are:

1. **Limited network overload** - The use of sets leads to lower overload on the network because every read is on relevant keys, avoiding unnecessary requests to the server. To further increase efficiency, MGET is used instead of single GET operations, executing multiple key reads in one request and reducing the round-trip between client and server.
2. **High scalability** - Although the amount of patient information stored in Redis at the same time is very high, the number of calls will not be related to this, thus not affecting the service's performance. The reason is that, thanks to Sets, calls will only be on keys for the specific patient (and not a large set of keys shared between patients).

In this case, considering 1500 patients instead of 150, the performance results are the following:



```
View home per farmacia per: PCLASSA53A43L378P time: 216
View home per farmacia per: PCBZGAI85P43L378W time: 221
View home per farmacia per: PTRMSRA51T43L378C time: 347
View home per farmacia per: PVSASGJ61M03L378A time: 388
View home per farmacia per: PDLFCCL04A43L378G time: 280
View home per farmacia per: PNCLVNT69C43L378O time: 307
View home per farmacia per: PSNTMNN66E43L378N time: 230
View home per farmacia per: PMSSGRG95S43L378F time: 223
View home per farmacia per: PCCRMTP98B03L378G time: 501
View home per farmacia per: PSSNMTT65P03L378N time: 461
View home per farmacia per: PCZZMNL86R03L378Y time: 439
View home per farmacia per: PCRCMME81A43L378H time: 471
View home per farmacia per: PFNTMLD79C43L378P time: 515
View home per farmacia per: PMZZVTR58A43L378N time: 394
View home per farmacia per: PCRNFRZ70E03L378B time: 222
View home per farmacia per: PDDNLNZ90T43L378U time: 237
View home per farmacia per: PBLSYSM61C43L378R time: 266
```

Figure 3.15: Optimized response time

Now the response times are in the order of tenths of a second, with 0.3s on average. This is the experimental demonstration on how the new solution leads to incredibly low response times.
It is important to note that the number of keys in the database is not comparable to the final version of the system hypothesized and for practical reasons the tests were conducted on a limited set of keys.

3.3.6 Eviction Policy

In Redis, eviction policies determine how the system handles memory limits when the configured maxmemory threshold is reached. The maxmemory-policy setting

specifies which keys should be removed to free up space.
The chosen policy is *volatile-TTL*, that removes keys with expire filed set to true and the shortest remaining time-to-live (TTL) value.
The reason for this choice is that, most of the time, the keys with the shortest TTL are related to drugs added to carts, since their expiration time is much shorter than that of prescriptions. This is an advantage because removing drugs from carts doesn't cause any issues. If needed, users can simply add them again before the confirmation of the cart.

3.3.7 Persistence

Persistence refers to the process of writing data to a durable storage medium, such as a solid-state drive (SSD). Redis provides various persistence options. [17]
For this project, the most logical choice is the Append Only File (AOF) option, which logs every write operation received by the server. These operations can then be replayed upon server restart, reconstructing the original dataset. Commands are stored in the same format as the Redis protocol itself. The selected and Redis-recommended policy is `fsync everysecond` [16], which offers a balance between speed and reliability. This choice is motivated by the fact that a patient's active prescriptions may remain in the database for up to a month. Therefore, we aim to minimize the risk of data loss that would require doctors to re-enter prescriptions manually.
To optimize Redis restart times, it's enabled the option `aof-use-rdb-preamble yes`, which allows Redis to combine an RDB-format snapshot at the beginning of the AOF file, followed by incremental AOF commands. This way, during a restart, Redis can quickly load the database state from the RDB dump, avoiding the slower process of replaying the entire AOF log. This strategy significantly enhances recovery performance by reducing the time required to restore data.

3.3.8 First deployment

As mentioned previously, Redis is populated with data for 1,500 patients. Each patient has 2 randomly drugs prescribed by the doctor and 3 random OTC drugs in their purchase cart.

3.4 Volume Considerations

55 million people live in Italy, but it's impossible for everyone to download an app. The most famous one is IO, an application owned by the Italian public service, which has been downloaded by 10 million people.
The hypothesis is to have the same total number of accounts, as EasyDrugs might be an app distributed by the Italian Ministry of Health and potentially promoted like the IO app. DAU (daily active users) could reach about 30%, since it operates

in heterogeneous fields like medicine, health, and payments [5]. This would mean approximately 3,000,000 active users daily. In Italy, there are about 40,000 family doctors, 70,000 drug researchers, and 30,000 pharmacies. Another hypothesis is that the government might recommend installation through a request from the Italian Ministry of Health. All these professionals would likely use the application daily, meaning that out of the 3,000,000 DAU, about 140,000 would be special accounts (always active), leaving 2,860,000 patients.
Two volume tables are provided: one is computed considering the ratio between MongoDB and Redis operations (as explained later), and the other does not include this ratio.

Figure 3.16: Frequency of utilization of common app

Action	Occurrence for 1 user	Occurrence for DAU	MongoDB operations	Redis operations	Total operations	Usage
Load patient home*	1	2,860,000	0	9R	3,785,294	4.1%
Load pharmacy home*	200	6,000,000	0	10R	8,823,529	9.5%
Load doctor home*	25	1,000,000	0	10R	1,470,588	1.6%
Search a drug	2**	5,720,000	1R	0	5,720,000	6.1%
Make a purchase	200	6,000,000	1R+2W	25R+39W	67,588,235	72.5%
Activate a prescription	17	680,000	2R	15R+30W	5,260,000	5.6%
Browse patient purchase	15	600,000	1R	0	600,000	0.6%
TOTAL		22,940,000			93,247,646	100%

Figure 3.17: Volume table with ratio consideration

Action	Occurrence for 1 user	Occurrence for DAU	MongoDB operations	Redis operations	Total operations	Usage
Load patient home*	1	2,860,000	0	9R	25,740,000	4.8%
Load pharmacy home*	200	6,000,000	0	10R	60,000,000	11.2%
Load doctor home*	25	1,000,000	0	10R	10,000,000	1.9%
Search a drug	2**	5,720,000	1R	0	5,720,000	1.1%
Make a purchase	200	6,000,000	1R+2W	25R+39W	402,000,000	75.0%
Activate a prescription	17	680,000	2R	15R+30W	31,960,000	5.9%
Browse patient purchase	15	600,000	1R	0	600,000	0.1%
TOTAL		22,940,000			536,020,000	100%

Figure 3.18: Volume table without ratio consideration

Notes:

- * : load pharmacy, doctor, patient home means load active prescriptions of a specific patient (very similar actions).

- ** : A drug is browsed about 2 times because a patient may search already prescribed drugs (want to know side-effects about what is going to buy) and two is the mean number of drugs prescribed by a doctor.

3.4.1 About Redis operations

Load patient home

- 1R: obtain the list of prescriptions from the user's set (in this case there is only one prescription).
- 1R: get the timestamp of the prescription.
- 1R: get the drug IDs in the prescription.
- 2×3R: get info for each drug in the prescription (2 drugs, 3 keys).

Total: 9R.

Load pharmacy home

A patient in a pharmacy typically stays for about 5 minutes. Approximately 12 people may be seen per hour, and about 96 people in 8 hours. With 2 pharmacists working, this amounts to roughly 200 patients. This explains why there can be a very high number of occurrences with just one pharmacy account.
The pharmacy homepage contains, in addition to the information displayed to patients on their home page, the patient's purchase cart. However, since in most cases a pharmacy selects a patient first and then accesses their homepage, when a patient initiates a purchase, the cart will initially be empty. Consequently, the read and write operations will be identical to those for loading the patient homepage, with the additional step of reading the list of drugs in the cart (which, as noted, starts empty but may contain items if reloaded during the purchasing process).

Load doctor home

A patient in a doctor's consultation typically stays about 15 minutes. Approximately 4 people can be seen per hour, totaling 28 to 32 people in 7-8 hours. The Italian Ministry of Health estimates about 25 people. This justifies the chosen number of occurrences for one doctor account.
Regarding operations, the same process applies as when loading the pharmacy homepage, except that the cart is for new prescriptions instead.

Make a purchase

The worst case scenario is when a patient buys an OTC drug and 2 prescribed drugs. In that case, an OTC drug must be searched, it must be added to the cart, 2

prescribed drugs must be added to the cart (just loaded), to complete the purchase the three drugs must be deleted from the cart and then the purchase must be added to the drugs and purchases collections.
In the phase of adding one element to the cart, the Redis operations are the following:

- 1R: info about the drug.
- 1R: get id from the stack.
- 2×1W: set drug id and set expire on the key.
- 2×1W: set drug info and set expire on the key.
- 2×1W: set drug id in the list and set expire on the key.

To add 1 OTC and 2 prescribed drugs, it is important to multiply by 3 the count, obtaining 8R+18W. Additionally, there is a control to check if the drug already exists in the cart that causes n+1 R where n is the number of items before the insertion of the new one. So in this case: 14R+18W.
In the purchase completion phase, the Redis operations are the following:

- 1R: get drug ids in the purchase cart.
- 3×2R: get drug keys (2 keys and 3 drugs).
- 2×1R: get prescription date (2 prescribed drugs).
- 2×1R: get toPurchase (2 prescribed drugs).
- 3×3W: delete drug keys in the cart (3 keys and 3 drugs).
- 1W: delete drug list in the cart.
- 2×4W: delete prescribed drugs info (2 prescribed drugs, 4 keys).
- 1W: delete drugs list from prescription.
- 2W: update log and set expire on log.

In this phase there are 11R+21W. Therefore, combining both phases, there are 25R+39W Redis operations required to complete a purchase.

Activate a prescription

Consider the case of 1 prescription with 2 drugs:

- 5R: check if drugs are already in the prescription cart. The number of reads increases with the number of drugs and can be calculated using the formula 2+n R, where n is the number of drugs before insertion.
- 1R: get list of prescriptions.
- 1R: get new id for the prescription.
- 2×1R: get id for each drug.
- 2W: set the date for the prescription and the expiration on the key.
- 2×2W: set the id of the drug and expiration on the key.
- 2×2W: set info of the drug and expiration on keys.
- 2W: add to the list of ids of the new prescription.
- 2×2W: add drug id to the list of drugs in the prescription.
- 1R: check timestamp of prescription (cannot be null).
- 1R: get list of prescribed drugs.
- 2×1W: set if the drug has not been purchased (2 drugs).
- 2×3W: set expiration on keys of the drug (2 drugs and 3 keys).
- 2×2R: get id and info for each drug.
- 2W: set keys of prescription.
- 4W: set expiration (2 keys and 2 lists).

Total: 15R+30W Redis operations.

3.4.2 Usage estimation

Redis operations are faster than MongoDB operations because Redis is an in-memory database. The latency difference is significant, which is why we don't simply sum Redis and MongoDB operations. Instead, we apply a ratio to Redis operations to make them at least comparable.
These estimates show how many reads and writes can be performed in Redis or MongoDB in one second (Note: These are only estimates and values may vary depending on hardware or system versions) [14] [19] [3]:

Table 3.2: Volume table

Database	**reads/s**	**writes/s**	**R latency**	**W latency**
MongoDB	13596	11000	0.074ms	0.09ms
Redis	92081	93109	0.011ms	0.011 ms

Therefore, the approximate ratios for Redis are:

- for reads: 6.8
- for writes: 8.5

Chapter 4

Implementation

The application was developed in Java using the Spring Boot framework to simplify implementation of the MVC (Model-View-Controller) design pattern.
The use of controllers annotated with `@RestController` enables efficient mapping of client HTTP requests. These controllers interact with the database, the model classes designed to represent and handle MongoDB document structures and also custom interfaces implemented for MongoDB and Redis operations.
Spring Data MongoDB was employed to facilitate database interactions with a high level of abstraction. For Redis operations, the Jedis library was utilized.
This architecture ensures:

- Modular handling of components
- Data scalability
- Proper separation of concerns according to MVC principles

.

4.1 Project Structure

JAVA project has the following tree.

```
> EasyDrugServer
    >       .idea
    >   .mvn/wrapper
    >   src
        >   main
            >   java/it/unipi/EasyDrugServer
                >   config
                    +   RedisConfig.java
                >   controller
```

```
        +   AuthController.java
        +   DoctorController.java
        +   DrugController.java
        +   PatientController.java
        +   PharmacyController.java
        +   PurchaseController.java
        +   ResearcherController.java
    >   dto
        +   AccountPatientDTO.java
        +   ConfirmPurchaseCartDTO.java
        +   DoctorHomeDTO.java
        +   DrugDistributionDTO.java
        +   ErrorResponse.java
        +   LoginUserDTO.java
        +   Order.java
        +   PatientDoctorRatioDTO.java
        +   PharmacyHomeDTO.java
        +   PrescribedDrugDTO.java
        +   PrescriptionDTO.java
        +   PurchaseCartDrugDTO.java
        +   PurchaseDrugDTO.java
        +   ResponseDTO.java
        +   SessionUserDTO.java
        +   SignupUserDTO.java
        +   SimpleDrugDTO.java
        +   SimplePatientDTO.java
        +   TopDrugDTO.java
        +   TopRareIndicationDTO.java
        +   UserType.java
    >   exception
        +   BadRequestException.java
        +   ForbiddenException.java
        +   GlobalExceptionHandler.java
        +   NotFoundException.java
        +   UnauthorizedException.java
    >   model
        +   CommitLog.java
        +   Doctor.java
        +   Drug.java
        +   LatestDrug.java
        +   LatestPurchase.java
        +   Patient.java
        +   Pharmacy.java
        +   Purchase.java
        +   Researcher.java
        +   SideEffect.java
    >   repository
        >   mongo
            +   CommitLogRepository.java
            +   DoctorRepository.java
            +   DrugRepository.java
            +   PatientRepository.java
```

```
                    +   PharmacyRepository.java
                    +   PurchaseRepository.java
                    +   ResearcherRepository.java
                >   redis
                    +   PrescriptionRedisRepository.java
                    +   PurchaseCartRedisRepository.java
            >   service
                +   AuthService.java
                +   DoctorService.java
                +   DrugService.java
                +   PatientService.java
                +   PharmacyService.java
                +   PurchaseService.java
                +   ResearcherService.java
                +   UserService.java
            >   utility
                +   ConvertDates.java
                +   MongoBoot.java
                +   PasswordHasher.java
                +   RedisBoot.java
                +   RedisHelper.java
                +   RollbackProcessor.java
            +   EasyDrugServerApplication.java
        >   resources
            >   dbFiles
                +   doctors.json
                +   drugs.json
                +   patients.json
                +   pharmacies.json
                +   purchases.json
                +   researchers.json
            >   serverBats
                +       serverBoot.bat
                +       serverFlush.bat
            +   application.properties
    >   test/java/it/unipi/EasyDrugServer
        +   EasyDrugServerApplicationTests.java
>       target
+   .gitattributes
+   .gitignore
+   mvnw
+   mvnw.cmd
+   pom.xml
```

4.1.1 Config

This package contains only classes responsible for Redis database configuration, which occurs during application startup.

`RedisConfig` creates a Jedis object when the connection starts and closes it when the application stops. The `@Configuration` annotation was used to enable Spring

Boot to call it automatically.

4.1.2 Controller

This package contains classes that handle client interactions. Each class uses the `@RestController` annotation to implement methods that return `ResponseEntity<ResponseDTO>` (see DTO section later), inserted directly in the response body without a default template.
The `@RequiredArgsConstructor` annotation enables Spring to generate default constructors and inject service fields. `@RequestMapping` maps requests and distinguishes between different controllers.
All methods handle exceptions through the `GlobalExceptionHandler`.
In addition to controllers for Patient, Doctor, Pharmacy, Researcher, Drug and Purchase collections, there is also an `AuthController` responsible for login and signup functionality.

```

@RestController
@RequestMapping("/api/drugs")
@RequiredArgsConstructor
public class DrugController {
    private final DrugService drugService;
    private final GlobalExceptionHandler exceptionHandler;

    @PostMapping
    public ResponseEntity<ResponseDTO> insertDrug(@RequestBody Drug drug){
        try {
            drugService.insertDrug(drug);
            ResponseDTO response = new ResponseDTO(HttpStatus.CREATED, drug);
            return new ResponseEntity<>(response, HttpStatus.CREATED);
        } catch (BadRequestException e){
            return exceptionHandler.handleBadRequestException(e);
        } catch (MongoSocketException e) {
            return exceptionHandler.handleMongoDBException(e, HttpStatus.SERVICE_UNAVAILABLE);
        } catch (MongoException e) {
            return exceptionHandler.handleMongoDBException(e, HttpStatus.INTERNAL_SERVER_ERROR);
        } catch (Exception e){
            return exceptionHandler.handleException(e);
        }
    }
```

4.1.3 DTO

In this directory there are all DTOs (Data Transfer Objects) used in the project. A DTO is an object used to transfer data between different levels of the application,

particularly between backend and client.
Data is encapsulated in a simple and optimized structure, separating business logic from the user interface. This approach shows only useful information to the client and keeps the API more clear and secure.
DTOs use `@Getter`, `@Setter`, and `@Data` annotations to let Spring generate default getter and setter methods.
A particularly interesting case is `ResponseDTO`, used to standardize the server response to client requests. As shown in the snippet, the `status` attribute contains an HTTP code as `HttpStatus` object, indicating the response type. The `data` attribute may contain a generic object with specific response information.
This structure provides a standard JSON object composed of `status` and `data`, allowing the client to avoid checking HTTP response headers for the request status.

```
@Getter
public class ResponseDTO {
    private final int status;
    private Object data;

    public ResponseDTO(HttpStatus status, Object data) {
        this.status = status.value();
        this.data = data;
    }

    public ResponseDTO(HttpStatus status) {
        this.status = status.value();
    }
}
```

4.1.4 Exception

In this directory there is a list of exceptions and a GlobalExceptionHandler that defines a method for each of them. It handles service requests in case of failure and returns a ResponseEntity<ResponseDTO> object containing an error message for the user.
The @ExceptionHandler annotation in methods is used to link them to @RequestMapping. The @Component annotation makes Spring recognize the class as a Bean and initialize it, injecting dependencies at application startup (to enable binding with controllers).

```
@Component
public class GlobalExceptionHandler {
    public static final Logger logger =
    ↪  LoggerFactory.getLogger(GlobalExceptionHandler.class);

    @ExceptionHandler(NotFoundException.class)
```

```
    public ResponseEntity<ResponseDTO> handleNotFoundException(NotFoundException ex) {
        logger.warn("NotFoundException: {}", ex.getMessage());
        ResponseDTO error = new ResponseDTO(HttpStatus.NOT_FOUND, ex.getMessage());
        return new ResponseEntity<>(error, HttpStatus.NOT_FOUND);
    }
```

4.1.5 Service

Services are classes that contain the business logic of the application. Their methods define complex operations that work as an intermediate layer between controllers, models, and repositories. For each controller there is one service, and each controller method uses a specific service method.
The `@Autowired` annotation enables Spring to inject `mongoTemplate`, an object representing the `EasyDrugDb` instance in MongoDB (used to operate with database collections). The `@RequiredArgsConstructor` annotation allows Spring to generate default constructors and inject repository fields.

```
@Service
@RequiredArgsConstructor
public class DrugService {
    private final DrugRepository drugRepository;

    public void insertDrug(Drug drug) {
        if(drug.getIndications().isEmpty())
            throw new BadRequestException("No added indication");
        drugRepository.save(drug);
    }
```

4.1.6 Model

Models are classes representing the structure and content of documents in MongoDB collections. Each model defines fields bound with MongoDB attributes, letting the application work with Java instances. @Getter, @Setter and @Data are used to let Spring build getter and setter methods.
@Document annotation is required to bind the bean with the collection, using its name. @CompoundIndex defines multiple fields as a compound index while @Indexed placed before a field indicates it as a single index.
@Id, @Field and @Transient can be placed before a field to handle specific configuration. @Id defines document id field to prevent MongoDB from handling it as a common one and generates ObjectId when needed (accounts use tax code as document id instead of ObjectId). @Field handles nested structure in Java because attributes containing an array of documents in MongoDB need to be defined as a list of objects. The object type in the list is typically a defined model (since a nested document has key-value structure) like LatestPurchase model for latestPurchasedDrugs field and SideEffect model for sideEffects field.

LatestDrug model defines a drug in the list of elements within a purchase (LatestPurchase).

```
@Getter
@Setter
@Data
@Document(collection = "drugs")
@CompoundIndex(def = "{'drugName': 1, 'onPrescription': 1}")
public class Drug {

    @Id
    private String id;
    private String drugName;
    private double price;
    private String company;
    private String activeIngredient;
    private String IUPAC;
    private String SMILES;
    private boolean onPrescription;

    @Field("indications")
    private List<String> indications;

    @Field("sideEffects")
    private List<SideEffect> sideEffects;
}
```

4.1.7 Other annotation

Some Swagger-related annotations were used in the code (not shown in previous examples for readability). These annotations help generate API documentation. For controllers, the following annotations can be used:

- @Operation: provides a name and description for the method
- @ApiResponse: adds a response code description
- @Parameter: provides a name, description, and example for method parameters
- @RequestBody: indicates a parameter that receives values from the request body
- @PathVariable: indicates a parameter that receives values from the request URL

```
@Operation(summary = "Insert drug", description = "Add a new drug in the
↪ system.")
    @ApiResponses(value = {
            @ApiResponse(responseCode = "201", description = "New drug
            ↪ created successfully."),
```

```
            @ApiResponse(responseCode = "400", description = "Not processable request due to a client error (malformed, invalid or deceptive syntax)."),
            @ApiResponse(responseCode = "500", description = "Server encountered a situation it does not know how to handle (generic error)."),
            @ApiResponse(responseCode = "503", description = "Server not ready to handle request (maintenance or overloaded).")
    })
    @PostMapping
    public ResponseEntity<ResponseDTO> insertDrug(@RequestBody @Parameter(name = "Drug struct", description = "Drug details.") Drug drug){

    ...

    @DeleteMapping("/{id}")
    public ResponseEntity<ResponseDTO> deleteDrug(@PathVariable("id") @Parameter(name = "Drug id", description = "Drug id.", example = "67aba9215da6705a000d3e87") String id){
```

For model can be used @Schema annotation that consent to specify a name, description, type and an example of the related attribute in the class.

```
public class Doctor {

    @Id
    @Schema(name ="id", description="Doctor's identify code, composed of 'D' followed by the Tax Code.", type="String",example = "DRSSMTN75E43F205M")
    private String id;
    @Schema(name = "password", description="Doctor's hashed password using bcrypt.",type="String",example = "$2a$12$sOFkuQwL2awh/FU7HEsudOGfW.pVzrvwZ97lVDEWJLt1f4up/cBIq")
    private String password;
    @Schema(name = "city", description="city of residence",type="String",example = "milano")
    private String city;
    @Schema(name = "district",description="District within the city.",type="String", example = "milano")
    private String district;
    @Schema(name = "region",description="Region of residence.",type="String", example = "Lombardia")
```

4.1.8 Repository

Mongo

Each collection in MongoDB is associated with an interface that extends MongoRepository, provided by Spring Data MongoDB. This interface is implemented as a class annotated with @Repository, making data access easier during execution.

MongoRepository provides and implements default methods for CRUD operations and handles data pagination and sorting.
@Query and @Aggregation are used to simply define advanced queries/search methods and aggregation pipelines, respectively.

```
@Repository
public interface DrugRepository extends MongoRepository<Drug, String> {

    @Query("{ 'drugName' : { $regex: '^?0'} }")
    List<Drug> findByDrugNameStarting(String drugName);

    @Query("{ 'drugName' : { $regex: '^?0'}, 'onPrescription' : ?1 }")
    List<Drug> findByDrugNameStartingAndOnPrescription(String name,
     boolean onPrescription);

    @Aggregation(pipeline = {
            "{ $unwind: '$indications' }",
            "{ $group: { " +
                    "_id: '$indications', " +
                    "drugNames: { $addToSet: '$drugName' }, " +
                    "drugCount: { $sum: 1 } } }",
            "{ $sort: { drugCount: 1 } }",
            "{ $limit: ?0 }"
    })
    List<TopRareIndicationDTO> getIndicationsWithLessDrugs(@Param("top")
     int top);

    List<Drug> findByOnPrescriptionTrue();
}
```

Redis

The `PrescriptionRedisRepository` and `PurchaseCartRepository` classes implement methods for interacting with Redis. Both are annotated with the `@Retryable` bean, which facilitates fault tolerance management, as explained before. Each implemented method is enclosed within a `try-with-resources` statement, allowing a connection to be acquired from the pool and automatically released at the end of the method execution.

```
@Setter
@Getter
@Repository
@Retryable(
        retryFor = { JedisConnectionException.class },
        maxAttempts = 3,
        backoff = @Backoff(delay = 5000)
)
public class PrescriptionRedisRepository {
    private final String pres = "pres";
```

```
    private final String presDrug = "pres-drug";
    private final JedisSentinelPool jedisSentinelPool;
    private final RedisHelper redisHelper;
    private final int day = 3600*24;
    private final int month = day*30;

    @Autowired
    public PrescriptionRedisRepository(JedisSentinelPool jedisSentinelPool, RedisHelper redisHelper) {
        this.jedisSentinelPool = jedisSentinelPool;
        this.redisHelper = redisHelper;
    }

    private PrescribedDrugDTO createPrescribedDrugDTO(Jedis jedis, String id, int quantity, String key){
        PrescribedDrugDTO prescribedDrug = new PrescribedDrugDTO();
        JsonObject jsonObject = JsonParser.parseString(jedis.get(key + "info")).getAsJsonObject();
        prescribedDrug.setId(id);
        prescribedDrug.setName(jsonObject.get("name").getAsString());
        prescribedDrug.setPrice(jsonObject.get("price").getAsDouble());
        prescribedDrug.setQuantity(quantity);
        prescribedDrug.setPurchased(Boolean.parseBoolean(jedis.get(key + "purchased")));
        return prescribedDrug;
    }


```

4.1.9 Utility

In this folder there are classes used to support the normal behavior of the system:

- `RedisHelper` defines methods for Redis repository operations. It uses counters and stacks, one for each Redis entity. This prevents `id_purch`, `id_pres`, and `id_pres_drug` from reaching excessively high values over time by reusing old IDs that were previously removed from the database. It also simplifies selecting new IDs during element insertion. Let's see in details:
  - `getReusableId()`: Obtains an ID for new Redis elements. It retrieves an ID from the entity's stack if available; otherwise, it atomically increments the entity counter in Redis and uses that value.
  - `returnIdToPool()`: Called when keys associated with an ID are deleted (e.g., `"purch-drug:" + id_purch_drug + ":" + id_pat + ":info"`). The ID is pushed back into the available IDs stack for reuse.
- `PasswordHasher`: Using the BCrypt library, this class implements two methods for login/signup:

  - Hashes passwords with BCrypt (with a random salt)
  - Verifies user-provided passwords against stored hashes during authentication

- `RedisBoot`: Retrieves all patients and all drugs from MongoDB. It takes 1500 of those patients obtained and assigns 2 prescribed drugs to each, activates the prescription, and adds 3 OTC drugs to the purchase cart.

- `MongoBoot`: Takes the JSON files inside resources directory and import them into MongoDB, creating collections or inserting documents during first-time initialization. Converts ISO-format strings to Date types (since JSON lacks native Date support).

- `RollbackProcessor`: This class implements the manual rollback of MongoDB insert operations performed during the confirmation of a purchase from the shopping cart in case the modifications on Redis fail. This ensures consistency between the two databases, MongoDB and Redis. The class defines the `processRollback` method, which is periodically invoked through the @Scheduled annotation. This method retrieves all log entries where the `processed` field is set to false. These logs contain the IDs of purchases that need to be removed to restore consistency. At this point, the `rollbackPurchases` method is called. This method, made atomic using the @Transactional annotation, executes the actual rollback by deleting from MongoDB collections the purchase-related information that causes inconsistency. Further details are provided in Section *2.5.4 Handling Inter-Database Consistency*.

4.1.10 Resources

There are two directories:

- `dbFiles`: contains all JSON files (the collections) that `MongoBoot` imports into MongoDB.

- `serverBats`: contains `serverBoot.bat` used to provide quick access to OpenVPN (only requiring username and password for each replica) and serverFlush.bat that closes Redis and MongoDB on the replicas (also requiring username and password).

4.2 MongoDB operations

4.2.1 Login

This function allows any type of user to log in. The user's role is extracted from their identify code. For example, if the first letter is 'P', it indicates that the user attempting to log in is a patient. The first step is to check whether a patient with the given ID exists within the specific patient collection. If the patient exists, the password is verified using the method defined in PasswordHasher. If the login is successful, the function returns a SessionUserDTO object, which stores the user's identification code, name, and type. All this data must then be saved on the client side using cookies or session storage.

```java
@Service
@RequiredArgsConstructor
public class AuthService {
    ...
    public SessionUserDTO login(LoginUserDTO user) {
        SessionUserDTO sessionUserDTO = new SessionUserDTO();

        if (user.getIdentifyCode() == null || user.getIdentifyCode().isEmpty()) {
            throw new BadRequestException("Identify code cannot be empty");
        }
        if (user.getPassword() == null || user.getPassword().isEmpty()) {
            throw new BadRequestException("Password code cannot be empty");
        }

        String identifyCode = user.getIdentifyCode();
        String psw = user.getPassword();

        if(identifyCode.startsWith("Ph")){
            // significa che sto effettuando il login di una farmacia
            Optional<Pharmacy> optionalPharmacy = pharmacyRepository.findById(identifyCode);
            if(optionalPharmacy.isEmpty())
                throw new NotFoundException("Pharmacy does not exist");

            Pharmacy pharmacy = optionalPharmacy.get();
            if(!PasswordHasher.verifyPassword(psw, pharmacy.getPassword()))
                throw new UnauthorizedException("Wrong password");

            // se arrivo qui il login è corretto
            sessionUserDTO.setId(identifyCode);
            sessionUserDTO.setName(pharmacy.getName());
            sessionUserDTO.setType(UserType.PHARMACY);

        } else{
            switch (identifyCode.charAt(0)) {
                case 'P':
                    // login patient
                    Optional<Patient> optionalPatient = patientRepository.findById(identifyCode);
                    if (optionalPatient.isEmpty())
                        throw new NotFoundException("Patient does not exist");

                    Patient patient = optionalPatient.get();
```

```
                    if(!PasswordHasher.verifyPassword(psw,
                    ↪  patient.getPassword()))
                        throw new UnauthorizedException("Wrong password");

                    // se arrivo qui il login è corretto
                    sessionUserDTO.setId(identifyCode);
                    sessionUserDTO.setName(patient.getName());
                    sessionUserDTO.setType(UserType.PATIENT);
                    break;
                case 'D':
                    // login doctor
                    Optional<Doctor> optionalDoctor =
                    ↪  doctorRepository.findById(identifyCode);
                    if (optionalDoctor.isEmpty())
                        throw new NotFoundException("Doctor does not
                        ↪  exist");

                    Doctor doctor = optionalDoctor.get();
                    if(!PasswordHasher.verifyPassword(psw,
                    ↪  doctor.getPassword()))
                        throw new UnauthorizedException("Wrong password");

                    // se arrivo qui il login è corretto
                    sessionUserDTO.setId(identifyCode);
                    sessionUserDTO.setName(doctor.getName());
                    sessionUserDTO.setType(UserType.DOCTOR);
                    break;
                case 'R':
                    // login researcher
                    Optional<Researcher> optionalResearcher =
                    ↪  researcherRepository.findById(identifyCode);
                    if(optionalResearcher.isEmpty())
                        throw new NotFoundException("Researcher does not
                        ↪  exist");

                    Researcher researcher = optionalResearcher.get();
                    if(!PasswordHasher.verifyPassword(psw,
                    ↪  researcher.getPassword()))
                        throw new UnauthorizedException("Wrong password");

                    sessionUserDTO.setId(identifyCode);
                    sessionUserDTO.setName(researcher.getName());
                    sessionUserDTO.setType(UserType.RESEARCHER);
                    break;
                default:
                    // identifyCode sbagliato
                    throw new BadRequestException("One or more fields have
                    ↪  unknown type of data");
            }
        }
        return sessionUserDTO;
    }
```

```
    ...

```

4.2.2 Purchase History

The application allows users to view their purchase history. Purchases are stored in the MongoDB collection associated with a specific patient. As explained before, last five purchases are retriven very fast thanks to the document structure. However, in this case, the goal is to display the entire purchase history, starting from the most recently purchased drugs. The data will be displayed in groups of ten.
The function responsible for this process is called getNextPurchaseDrugs and takes as parameters the patient's identify code and the number of drugs already displayed. After verifying that the provided code belongs to a valid patient, it is retrieve the purchases array of ObjectIds from the patient's document. This array contains all the IDs of the drugs purchased by the patient, each of which refers to a document in the purchases collection. The most recent purchases are stored at the end of the array, so it is important to start reading from the bottom. After retrieving a subset of up to ten IDs, it is fetched all the information related to these purchased drugs using the index on the purchases collection.
The final step is to return a list of purchases to the client, rather than a list of individual drugs. To achieve this, it is used a temporary LinkedHashMap, which, unlike HashMap, preserves the order in which key-value pairs were inserted. Since the purchases field is structured chronologically, data will be inserted in the correct order. Purchased drugs on the same date are grouped within the same purchase object. Finally, the LinkedHashMap is converted into a list and returned to the controller.

```java
public List<LatestPurchase> getNextPurchaseDrugs(String id_pat, int n_uploaded) {
    if(!patientRepository.existsById(id_pat))
        throw new NotFoundException("Patient "+id_pat+" does not exist");
    Optional<Patient> optPatient = patientRepository.findById(id_pat);
    List<String> purchasesId = new ArrayList<>();
    List<Purchase> purchases = new ArrayList<>();
    if(optPatient.isPresent())
        purchasesId = optPatient.get().getPurchases();
    int startIndex = purchasesId.size() - n_uploaded;
    if(startIndex <= 0)
        return new ArrayList<>();
    int endIndex = startIndex - N_TO_VIEW;
    if(endIndex < 0) endIndex = 0;
    List<String> idToView = purchasesId.subList(endIndex, startIndex);
    for(String purchId: idToView){
        Optional<Purchase> optPurch = purchaseRepository.findById(purchId);
        optPurch.ifPresent(purchases::add);
    }
    Map<LocalDateTime, LatestPurchase> hashPurchases = new LinkedHashMap<>();
    for(int i=purchases.size()-1; i>=0; i--){
        LatestDrug latestDrug = new LatestDrug();
        Purchase purch = purchases.get(i);
        latestDrug.setDrugId(purch.getDrugId());
        latestDrug.setDrugName(purch.getName());
        latestDrug.setQuantity(purch.getQuantity());
        latestDrug.setPrice(purch.getPrice());
        latestDrug.setPrescriptionDate(purch.getPrescriptionDate());
        if(!hashPurchases.containsKey(purch.getPurchaseDate())){
            LatestPurchase latestPurchase = new LatestPurchase();
            List<LatestDrug> drugs = new ArrayList<>();
            drugs.add(latestDrug);
            latestPurchase.setTimestamp(purch.getPurchaseDate());
            latestPurchase.setDrugs(drugs);
            hashPurchases.put(purch.getPurchaseDate(), latestPurchase);
        } else
            hashPurchases.get(purch.getPurchaseDate()).getDrugs().add(latestDrug);
    }
    return new ArrayList<>(hashPurchases.values());
}
```

Figure 4.1: getNextPurchaseDrugs method

4.2.3 Patient-to-Doctor Ratio by City (Analytics 1)

This query calculates the ratio of patients to doctors in each city. This is important because a high ratio in a given city indicates that doctors are responsible for too many patients, which may make it difficult for patients to book appointments. The query operates under the assumption that family doctors serve patients in the same city. It works on a single collection, the patient collection, and follows these steps:

1. **Grouping Phase:** The query begins with a grouping operation ($group) that aggregates documents based on the city field. For each city, it calculates the total number of patients (nPatients) using the $sum operator. Additionally, the unique doctor identification codes (doctorCode) associated with the patients are collected into a set called distDoctors to ensure uniqueness per city.

2. **Retrieving the Number of Doctors:** A projection operation ($project) is then performed to select and restructure the relevant fields. The city field (obtained from the ID generated in the grouping phase) and the nPatients field are preserved. Furthermore, the number of distinct doctors (nDoctors) operating in each city is determined using the $size operator applied to the distDoctors set.

3. **Calculating the Patient-to-Doctor Ratio:** In the second projection phase, a new field, ratio, is introduced, representing the ratio of patients (nPatients) to doctors (nDoctors) for each city. This ratio is calculated using the $divide operator, which divides nPatients by nDoctors.

4. **Sorting the Results:** Finally, the results are sorted ($sort) based on the ratio field. The sorting order can be ascending or descending, depending on the parameter (?0) passed to the query.

```
@Aggregation(pipeline = {  2 usages  Lorenzo Valtriani
        "{ $group: { _id: '$city', nPatients: { $sum: 1 }, distDoctors: { $addToSet: '$doctorCode' } } }",
        "{ $project: { city: '$_id', nPatients: 1, nDoctors: { $size: '$distDoctors' } } }",
        "{ $project: { city: 1, ratio: { $divide: ['$nPatients', '$nDoctors'] } } }",
        "{ $sort: { ratio: ?0 } }"
})
List<PatientDoctorRatioDTO> getPatientsToDoctorsRatio(@Param("order") int order);
```

Figure 4.2: Analytics 1 code

4.2.4 Retrieving Indications with Fewer drugs (Analytics 2)

This query aims to obtain a limited list of indications that are associated with the fewest drugs. It is important because if only a few drugs are indicated for a certain disease, it may suggest the need for research to develop more drugs with the same purpose, thereby increasing competition among pharmaceutical companies. The process follows these steps:

1. **Unwind Phase:** The query starts by performing an unwind operation on the "indications" field, which is an array within a drug document. This generates multiple documents for the same drug, each corresponding to a different element of the original array.

2. **Grouping Phase:** Next, a grouping operation (group) aggregates documents by indication. For each indication, the related drugs are stored in the drugNames field, and their count is stored in the drugCount field.

3. **Sorting the Results:** The results are sorted ($sort) based on the drugCount field.

4. **Limiting the Results:** Finally, only the top N documents are returned, with N being a variable input parameter for the analytics.

```
@Aggregation(pipeline = {  1 usage  supersamu2001
        "{ $unwind: '$indications' }",
        "{ $group: { " +
                "_id: '$indications', " +
                "drugNames: { $addToSet: '$drugName' }, " +
                "drugCount: { $sum: 1 } } }",
        "{ $sort: { drugCount: 1 } }",
        "{ $limit: ?0 }"
})
List<TopRareIndicationDTO> getIndicationsWithLessDrugs(@Param("top") int top);
```

Figure 4.3: Analytics 2 code

4.2.5 Retrieving the Most Purchased drugs (Analytics 3)

This query retrieves a list of the most purchased drugs within a specified time period. This is crucial for understanding which drugs are in high demand, helping pharmaceutical companies optimize production. Additionally, each year the Ministry of Health compiles a ranking of the best-selling drugs. The process follows these steps:

1. **Filtering Documents by Purchase Date:** The query begins with a match operation on the "purchaseDate" field, filtering purchases that fall within the specified date range. This operation is optimized using an index on the purchase date field.

2. **Grouping Phase:** A grouping operation (group) then aggregates documents by the unique drug ID and name. The total number of purchases for each drug is stored by summing the quantity field across all purchases.

3. **Sorting the Results:** The results are sorted ($sort) in descending order based on the totalQuantity field.

4. **Limiting the Results:** Finally, only the top N documents are returned, with N being a variable input parameter for the analytics.

```
@Aggregation(pipeline = {  1 usage  supersamu2001
        "{ $match: { purchaseDate: { $gte: ?0, $lte: ?1 } } }",
        "{ $group: { _id: { drugId: '$drugId', name: '$name' }, totalQuantity: { $sum: '$quantity' } } }",
        "{ $sort: { totalQuantity: -1 } }",
        "{ $limit: ?2 }"
})
List<TopDrugDTO> getTopDrugs(@Param("from") LocalDateTime from, @Param("to") LocalDateTime to, @Param("top") int top);
```

Figure 4.4: Analytics 3 code

4.2.6 Geographical Distribution of drug Purchases (Analytics 4)

This query retrieves information about the regional distribution of sales for a specific drug within a given time frame. For each region, it counts the number of units sold and calculates the percentage of total sales in Italy for that drug. This helps distributors understand which regions have the highest demand and adjust supply accordingly. The process follows these steps:

1. **Filtering Documents by Purchase Date and drug ID:** The query begins with a $match operation, filtering purchases of a specific drug within the given date range.

2. **Grouping by Region:** Documents are then grouped by region, and the total number of drugs sold is calculated by summing the quantity field.

3. **Calculating Total Sales in Italy:** A new document is created with two key fields:

   - regionalSales: An array containing the number of drugs sold per region.
   - numberOfSoldDrugsInItaly: The total number of drugs sold nationwide.

4. **Computing Regional Sales Percentage:** Using $project, the percentage of sales per region is calculated as:

$$\textit{percentage} = \textit{(regional sales / total sales in Italy) * 100}$$

5. **Sorting the Results:** Finally, the results are sorted by the percentage field, based on the specified sorting order (?1).

```
@Aggregation(pipeline = {  2 usages  new *
        "{ $match: { purchaseDate: { $gte: ?2, $lte: ?3 }, drugId: ?0 } }",
        "{ $group: { _id: '$region', numberOfSoldDrugs: { $sum: '$quantity' } } }",
        "{ $group: { " +
                "_id: null, " +
                "regionalSales: { $push: { region: '$_id', numberOfSoldDrugs: '$numberOfSoldDrugs' } }, " +
                "numberOfSoldDrugsInItaly: { $sum: '$numberOfSoldDrugs' } } }",
        "{ $unwind: '$regionalSales' }",
        "{ $project: { " +
                "region: '$regionalSales.region', " +
                "numberOfSoldDrugs: '$regionalSales.numberOfSoldDrugs', " +
                "percentage: { $multiply: [ " +
                "{ $cond: { " +
                "if: { $eq: ['$numberOfSoldDrugsInItaly', 0] }, " +
                "then: 0, " +
                "else: { $divide: [{ $toDouble: '$regionalSales.numberOfSoldDrugs' }, '$numberOfSoldDrugsInItaly'] } } }, " +
                "100 ] } } }",
        "{ $sort: { percentage: ?1 } }"
})
List<DrugDistributionDTO> getDistributionByDrug(@Param("drugId") ObjectId drugId, @Param("order") int order,
                                                @Param("from") LocalDateTime from, @Param("to") LocalDateTime to);
```

Figure 4.5: Analytics 4 code

4.3 Redis operations

Below, we present some of the main methods of these repositories: PrescriptionRedisRepository and PurchaseCartRepository.

4.3.1 getPrescriptionCart

This method returns a PrescriptionDTO object representing the prescription cart. After obtaining a Jedis object, the method jedis.smembers("pres:" + id_pat + ":set") retrieves a list of id_pres. Each of these IDs will form the following key: "pres:" + id_pres + ":" + id_pat + ":timestamp", which points to the timestamp string representing when that prescription was created. Iterating over the list, each key is checked, looking for one that points to an empty string. If such a key exists, it will be the only one for that patient and will indicate the prescription cart. Next, the smembers method retrieve the list of id_pres_drug, allowing quick access to all drugs within the shopping cart. All their details are fetched in a single call using the mget method. After reading the values associated with the relevant keys, it is constructed the prescription object and returned to the service.

```java
public PrescriptionDTO getPrescriptionCart(String id_pat) {  1 usage  Lorenzo Valtriani +1 *
    try (Jedis jedis = jedisSentinelPool.getResource()) {
        PrescriptionDTO prescription = new PrescriptionDTO();
        prescription.setTimestamp(null);
        String listKey = this.pres + ":" + id_pat + ":set";
        List<String> presIds = new ArrayList<>(jedis.smembers(listKey));
        if(presIds.isEmpty())
            return prescription;

        // ottenimento delle SOLE N prescrizioni dell'utente
        for (String id_pres : presIds) {
            String keyPres = this.pres + ":" + id_pres + ":" + id_pat + ":";
            String listKeyPresDrug = this.presDrug + ":" + id_pres + ":set";
            // controllo che la prescrizione sia o meno attiva
            String timestamp = jedis.get(keyPres + "timestamp");
            if (timestamp.isEmpty()) {
                // in questo caso la prescrizione è inattiva
                // otteniamo i SOLI id dei farmaci del carrello della prescrizione
                List<String> presDrugIds =  new ArrayList<>(jedis.smembers(listKeyPresDrug));
                List<String> keys = new ArrayList<>();
                for (String id_pres_drug : presDrugIds) {
                    String keyPresDrug = this.presDrug + ":" + id_pres_drug + ":" + id_pres + ":";
                    keys.add(keyPresDrug + "id");
                    keys.add(keyPresDrug + "info");
                }
                // effettuiamo una sola chiamata a MGET per tutti e i SOLI farmaci della prescrizione
                List<String> values = jedis.mget(keys.toArray(new String[0]));
                for (int i = 0; i < presDrugIds.size(); i++) {
                    int id_pres_drug = Integer.parseInt(presDrugIds.get(i));
                    String idDrug = values.get(i * 2);
                    String infoJson = values.get(i * 2 + 1);
                    PrescribedDrugDTO drug = createPrescribedDrugDTO(idDrug, infoJson, purchased: "false", id_pres_drug);
                    prescription.getPrescribedDrugs().add(drug);
                }
                prescription.setIdPres(Integer.parseInt(id_pres));
                return prescription;
            }
        }
        return prescription;
    }
}
```

Figure 4.6: getPrescriptionCart method

4.3.2 activatePrescriptionCart

This method activates the prescription cart, making it visible and usable for the patient. Initially, it verifies the input parameters id_pat and id_pres, ensuring that the key "pres:" + id_pres + ":" + id_pat + ":timestamp" exists and points to an empty string. If i

t contains a valid timestamp instead, it means the prescription is already active. Then, the list of id_pres_drug, found in the prescription cart, is retrived and checked that the count is not zero, as that would indicate an empty cart. For each drug in the cart, the "purchased" field is set to false. Additionally, for each drug, the expiration date of its key is modified by extending it by one month, which is the validity period for prescriptions. The expiration dates of the prescription-related keys are also set to one month—first "timestamp" (set to the current timestamp) and then "toPurchase" (set to the number of distinct drugs in the cart). Finally, the method returns a prescription object to the service, containing the drugs in the cart, its id_pres, and the current timestamp.

```java
public class PrescriptionRedisRepository {
    ...
    public PrescriptionDTO activatePrescriptionCart(String id_pat, int id_pres) {
        try(Jedis jedis = jedisSentinelPool.getResource()) {
            PrescriptionDTO prescription = new PrescriptionDTO();
            prescription.setIdPres(id_pres);
            prescription.setTimestamp(LocalDateTime.now());
            // controllare che la prescrizione esista e sia un carrello (inattiva)
            String keyPres = this.pres + ":" + id_pres + ":" + id_pat + ":";
            String timestamp = jedis.get(keyPres + "timestamp");
            if(timestamp == null)
                throw new NotFoundException("Not found any prescription with id " + id_pres + " related to patient " + id_pat);
            if(!timestamp.isEmpty())
                throw new NotFoundException("Not found any prescription cart related to patient " + id_pat);

            // ottenimento di tutti e i SOLI farmaci del carrello delle prescrizioni
            String listPresDrug = this.presDrug + ":" + id_pres + ":set";
            String listPres = this.pres + ":"  + id_pat + ":set";
            List<String> presDrugIds = new ArrayList<>(jedis.smembers(listPresDrug));
            int nDrugs = presDrugIds.size();
            if(nDrugs == 0)
                throw new ForbiddenException("The patient "+id_pat+" has no drugs in the cart.");

            // otteniamo la lista di tutte le chiavi da leggere da redis
            List<String> keys = new ArrayList<>();
            for (String id_pres_drug : presDrugIds) {
                String keyPresDrug = this.presDrug + ":" + id_pres_drug + ":" + id_pres + ":";
                keys.add(keyPresDrug + "id");
                keys.add(keyPresDrug + "info");
                // se è la prima volta ad inserirle
                jedis.set(keyPresDrug + "purchased", "false");
                // modify time to expire for all drugs into the prescription to activate
                jedis.expire(keyPresDrug + "id", this.month);
                jedis.expire(keyPresDrug + "info", this.month);
                jedis.expire(keyPresDrug + "purchased", this.month);
            }
```

```java

            // effettuiamo una sola chiamata a MGET per tutti e i SOLI farmaci della prescrizione
            List<String> values = jedis.mget(keys.toArray(new String[0]));
            for (int i = 0; i < presDrugIds.size(); i++) {
                int id_pres_drug = Integer.parseInt(presDrugIds.get(i));
                String idDrug = values.get(i * 2);
                String infoJson = values.get(i * 2 + 1);
                PrescribedDrugDTO drug = createPrescribedDrugDTO(idDrug, infoJson, "false", id_pres_drug);
                prescription.getPrescribedDrugs().add(drug);
            }

            // setto il numero di farmaci all'interno
            jedis.set(keyPres + "toPurchase", String.valueOf(nDrugs));
            // setto il timestamp a quello di ora e conto i farmaci relativi a quella prescrizione
            jedis.set(keyPres + "timestamp", String.valueOf(prescription.getTimestamp()));

            // un mese dopo la sua creazione la prescrizione viene eliminata
            jedis.expire(keyPres + "timestamp", this.month);
            jedis.expire(keyPres + "toPurchase", this.month);
            jedis.expire(listPres, this.month);
            jedis.expire(listPresDrug, this.month);
            return prescription;
        }
    }
    ...
```

4.3.3 confirmPurchaseCart

This method allows the confirmation of a drug purchase and updates Redis accordingly. First, it retrieves information on each drug in the patient's shopping cart using the smembers and mget methods. Some drugs in the cart may have been previously prescribed by a doctor. These can be identified because the JSON object pointed to by the key *purch-drug:id_purch_drug:id_pat:info* contains the fields "id_pres" and "id_pres_drug", which indicate that the drug was previously prescribed. These two IDs are used to directly access the relevant prescription and drug information. The "purchased" attribute of those prescribed drugs must be set to true, and the "toPurchase" attribute of the corresponding prescription must be decreased by the distinct number of drugs purchased in this session. If the new value of "toPurchase" is zero, it means all drugs in the prescription have been purchased, and the related information can be removed from Redis. Instead of setting this attribute to zero, the prescription data is deleted from the key-value database.
It is important to note that all write operations in this method are performed within a transaction, ensuring their atomicity. This is the only method that uses trans-

actions to guarantee atomicity. This is necessary because, after deleting the drug-related information from Redis, the data must be saved in the document database. To ensure consistency between the two databases, transactions are used to easily manage rollbacks in Redis, as expleined before.

```java
public ConfirmPurchaseCartDTO confirmPurchaseCart(String id_pat){  1 usage  new *
    Jedis jedis = jedisSentinelPool.getResource();
    List<PurchaseCartDrugDTO> purchaseDrugs = new ArrayList<>();
    LinkedHashMap<Integer, List<Integer>> prescribedDrugs = new LinkedHashMap<>();
    LinkedHashMap<Integer, List<Integer>> presToDelete = new LinkedHashMap<>();
    LinkedHashMap<Integer, List<Integer>> presToModify = new LinkedHashMap<>();
    LinkedHashMap<Integer, Integer> newToPurchase = new LinkedHashMap<>();
    String listKey = this.entity + ":" + id_pat + ":set";
    List<String> purchIds = new ArrayList<>(jedis.smembers(listKey));
    List<String> keys = new ArrayList<>();
    for(String id_purch_drug: purchIds){
        keys.add(this.entity + ":" + id_purch_drug + ":" + id_pat + ":id");
        keys.add(this.entity + ":" + id_purch_drug + ":" + id_pat + ":info");
    }
    if (keys.isEmpty()) throw new ForbiddenException("You can not complete the payment if a cart is empty");
    List<String> values = jedis.mget(keys.toArray(new String[0]));
    for (int i = 0; i < purchIds.size(); i++) {
        int id_purch = Integer.parseInt(purchIds.get(i));
        String idDrug = values.get(i * 2);
        String infoJson = values.get(i * 2 + 1);
        PurchaseCartDrugDTO drug = createPurchaseCartDrugDTO(idDrug, infoJson, id_purch);
        if(drug.getIdPres() != null){
            String presDate = "pres:" + drug.getIdPres() + ":" + id_pat + ":timestamp";
            drug.setPrescriptionTimestamp(LocalDateTime.parse(jedis.get(presDate)));
            if(!prescribedDrugs.containsKey(drug.getIdPres())) prescribedDrugs.put(drug.getIdPres(), new ArrayList<>());
            prescribedDrugs.get(drug.getIdPres()).add(drug.getIdPresDrug());
        }
        purchaseDrugs.add(drug);
    }
    for(Map.Entry<Integer, List<Integer>> entry : prescribedDrugs.entrySet()){
        String keyPres = "pres:" + entry.getKey() + ":" + id_pat + ":toPurchase";
        int toPurchase = Integer.parseInt(jedis.get(keyPres));
        if(toPurchase == entry.getValue().size()) presToDelete.put(entry.getKey(), entry.getValue());
        else{
            newToPurchase.put(entry.getKey(), toPurchase - entry.getValue().size());
            presToModify.put(entry.getKey(), entry.getValue());
        }
    }
    ConfirmPurchaseCartDTO confirmPurchaseCartDTO = new ConfirmPurchaseCartDTO();
    confirmPurchaseCartDTO.setPurchaseDrugs(purchaseDrugs);
    confirmPurchaseCartDTO.setPresToDelete(presToDelete);
    confirmPurchaseCartDTO.setPresToModify(presToModify);
    confirmPurchaseCartDTO.setNewToPurchase(newToPurchase);
    confirmPurchaseCartDTO.setJedis(jedis);
    return confirmPurchaseCartDTO;
}
```

Figure 4.7: confirmPurchaseCart method

4.4 Endpoints

researcher-controller

PUT	`/api/researchers`	Update researcher information
GET	`/api/researchers/{id}`	Get researcher by id.
DELETE	`/api/researchers/{id}`	Delete researcher account
GET	`/api/researchers/ratios/patients-to-doctors/order/{order}`	Get patients/doctors ratio
GET	`/api/researchers/purchases/top/{top}/from/{from}/to/{to}`	Get most purchased drugs
GET	`/api/researchers/indications/less-drugs/top/{top}`	Get underrepresented indications
GET	`/api/researchers/drugs/{id_drug}/distribution/from/{from}/to/{to}/order/{order}`	Get drug distribution data

purchase-controller

PUT	`/api/purchases`	Update purchase information
POST	`/api/purchases`	Insert purchase
GET	`/api/purchases/{id}`	Get purchase by id
DELETE	`/api/purchases/{id}`	Delete purchase by id

pharmacy-controller

PUT	`/api/pharmacies`	Update pharmacy information
POST	`/api/pharmacies/patients/{id_pat}/cart/drugs`	Insert drug into purchase cart
PATCH	`/api/pharmacies/{id_pharm}/patients/{id_pat}/cart/checkout`	Confirm purchase
DELETE	`/api/pharmacies/patients/{id_pat}/cart/drugs/{id_purch_drug}`	Delete drug from cart
PATCH	`/api/pharmacies/patients/{id_pat}/cart/drugs/{id_purch_drug}`	Update drug quantity
GET	`/api/pharmacies/{id}`	Get pharmacy by id
DELETE	`/api/pharmacies/{id}`	Delete pharmacy account
GET	`/api/pharmacies/home/patients/{id_pat}`	View pharmacy home

patient-controller

PUT	`/api/patients`	Update patient information
GET	`/api/patients/{patCode}/prescriptions/active`	Get active prescriptions
GET	`/api/patients/{id}`	Get patient by id
DELETE	`/api/patients/{id}`	Delete patient account
GET	`/api/patients/{id}/purchases/latest`	Get latest purchases
GET	`/api/patients/{id}/purchases/from/{n_uploaded}`	Get next purchases
GET	`/api/patients/{id}/profile`	Get patient account by id.

drug-controller

PUT	`/api/drugs`	Update drug information
POST	`/api/drugs`	Insert drug
GET	`/api/drugs/{id}`	Get drug by id
DELETE	`/api/drugs/{id}`	Delete drug by id
GET	`/api/drugs/search/{name}`	Get drugs by name
GET	`/api/drugs/search/{name}/otp`	Get OTP drugs by name
GET	`/api/drugs/search/{name}/on-prescription`	Get prescribable drug by name

doctor-controller

PUT	`/api/doctors`	Update doctor information
POST	`/api/doctors/patients/{id_pat}/cart/{id_cart}/drugs`	Insert drug into cart
DELETE	`/api/doctors/patients/{id_pat}/cart/{id_cart}/drugs/{id_pres_drug}`	Delete drug from cart
PATCH	`/api/doctors/patients/{id_pat}/cart/{id_cart}/drugs/{id_pres_drug}`	Update drug quantity
PATCH	`/api/doctors/patients/{id_pat}/cart/{id_cart}/activate`	Activate prescription
GET	`/api/doctors/{id}`	Get doctor by id
DELETE	`/api/doctors/{id}`	Delete doctor account
GET	`/api/doctors/{id}/patients/surname/{pat_surname}`	Get patient by surname
GET	`/api/doctors/{id_doc}/patients/{id_pat}/prescriptions/from/{n_uploaded}`	Get next prescriptions
GET	`/api/doctors/home/patients/{id_pat}`	View doctor home

auth-controller

POST	`/api/auth/signup`	Sign-up
POST	`/api/auth/login`	Login

Appendix A

Future update

- Add graph DB to see interaction between active ingredient and indication and side effects.
- Add more informative fields to drugs, like usage (oral, nasal, ...).
- Add restriction based on ATC to prevent errors by doctor or pharmacy (drugs not assignable to underage people).
- Add possibility for the doctor to access to the clinical page of a patient.
- Add possibility for the patient to signal possible (certified by his doctor) side effect.
- Add possibility for the patient to signal instance of known side effect (researcher can view if the frequency of a side effect becomes higher and modify the field).
- Add possibility for drugs tracking in pharmacy to know the current availability and perform data analysis.

Appendix B

ChatGpt reference

This chapter is dedicated to the question made on ChatGPT. We want to underline how it can be an important instrument and a source of information as Google, if used with care. Not all information are corrected but ones used were validate with google. The main topic of questions was only where to get dataset for our project.

Screenshots of the main parts of the answer will be inserted and for transparency also link to the original chat. The standard used was ChatGpt4 (free version).

Question one

I'm searching dataset about drugs which contein information on active ingredient and chemical structure.

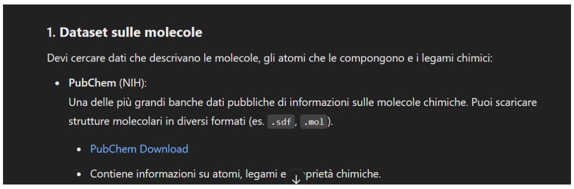


Figure B.1: Answer 1

Result: IUPAC, SMILES and active ingredient of drugs from Pubchem.

https://chatgpt.com/share/67a5c0ff-acc0-8001-bc09-80c88b7966a0

Question two

Can you find me dataset which contein information about drug like : purchase frequency, price, disease, syntoms, type of drug, company, identification.

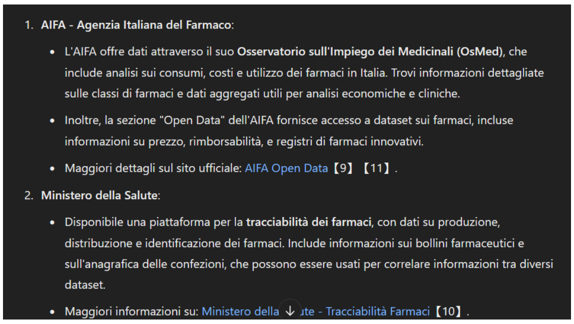


Figure B.2: Answer 2

Result: AIFA and Italian ministery of Health open data for information about drugs.

https://chatgpt.com/share/67a5c4f1-51a0-8001-b7ef-4559bb268f0e

Question three

Can you suggest me scraping technique or library for python ?

Figure B.3: Answer 3

Result: discover Selenium to get information from AIFA browsing system.

https://chatgpt.com/share/67a5c530-b20c-8001-956f-dcd539572ed6

Question four

Is there a dataset that associate a drug with a synthom ? Like Aspirin-headache.

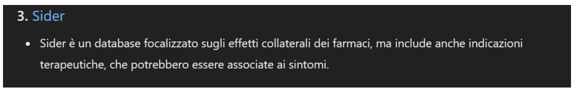


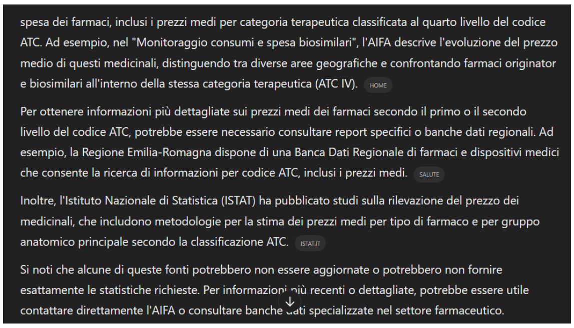
Figure B.4: Answer 4

Result:discover Sider for side-effects information.

https://chatgpt.com/share/67a5c576-7c2c-8001-8315-8da72d636ad4

Question five

Do you know any sources about the mean price of each ATC category?


spesa dei farmaci, inclusi i prezzi medi per categoria terapeutica classificata al quarto livello del codice ATC. Ad esempio, nel "Monitoraggio consumi e spesa biosimilari", l'AIFA descrive l'evoluzione del prezzo medio di questi medicinali, distinguendo tra diverse aree geografiche e confrontando farmaci originator e biosimilari all'interno della stessa categoria terapeutica (ATC IV). HOME

Per ottenere informazioni più dettagliate sui prezzi medi dei farmaci secondo il primo o il secondo livello del codice ATC, potrebbe essere necessario consultare report specifici o banche dati regionali. Ad esempio, la Regione Emilia-Romagna dispone di una Banca Dati Regionale di farmaci e dispositivi medici che consente la ricerca di informazioni per codice ATC, inclusi i prezzi medi. SALUTE

Inoltre, l'Istituto Nazionale di Statistica (ISTAT) ha pubblicato studi sulla rilevazione del prezzo dei medicinali, che includono metodologie per la stima dei prezzi medi per tipo di farmaco e per gruppo anatomico principale secondo la classificazione ATC. ISTAT.IT

Si noti che alcune di queste fonti potrebbero non essere aggiornate o potrebbero non fornire esattamente le statistiche richieste. Per informazioni più recenti o dettagliate, potrebbe essere utile contattare direttamente l'AIFA o consultare banche dati specializzate nel settore farmaceutico.

Figure B.5: Answer 5

Result: reference to calculate the average price of drugs.

https://chatgpt.com/share/67a5c669-7784-8001-8dd1-0ebcb9eeda14

Bibliography

[1] AdminStat. *Mappe, analisi e statistiche sulla popolazione residente*. Gathered info: age distribution related to italian population. 2022. URL: https://ugeo.urbistat.com/AdminStat/it/it/demografia/eta/italia/380/1.

[2] AGENAS. *Rapporto sui Medici di medicina generale (MMG)*. Gathered info: doctor distribution per region. 2021. URL: https://www.agenas.gov.it/comunicazione/primo-piano/2250-rapporto-sui-medici-di-medicina-generale.

[3] Giulio di Bari. *Redis: strumenti di benchmark*. Gathered info: Redis reads and writes per second. 2020. URL: https://www.webarea.it/howto/nosql/redis-strumenti-di-benchmark_159.

[4] The Medicines Utilisation Monitoring Centre. *National Report on Medicines use in Italy*. Gathered info: amount of packages for each ATC class. 2022. URL: https://www.aifa.gov.it/documents/20142/1967301/Rapporto-OsMed-2022.pdf.

[5] Andrew Chen. *DAU/MAU is an important metric to measure engagement, but here's where it fails*. Gathered info: map of DAU of common applications. URL: https://andrewchen.com/dau-mau-is-an-important-metric-but-heres-where-it-fails/.

[6] Agenzia Italiana del Farmaco. *AIFA-ricerca farmaco*. Specific area to browse drugs ditributed in Italy. URL: https://medicinali.aifa.gov.it/it/#/it/.

[7] Agenzia Italiana del Farmaco. *Liste dei farmaci AIFA*. Gathered info: most ordered drugs correlated to active ingredients. URL: https://www.aifa.gov.it/en/liste-dei-farmaci.

[8] Federfarma. *Dati di spesa Gennaio-Dicembre 2023*. Gathered info: total drug expenditure of SSN and average packages per patient. 2023. URL: https://www.federfarma.it/spesa-e-consumi-farmaceutici-ssn.aspx.

[9] Armen Forget. *Non functional requirements and functional requirement example*. Gathered info: functional requirements example and references. URL: https://stackoverflow.com/questions/62546088/non-functional-requirements-and-functional-requirement-example.

[10] Carlo De Gregorio. *Il nuovo impianto della rilevaione centralizzata del prezzo dei medicinali con obbligo di prescrizione*. Gathered info: selling frequency for ATC code and total amount of drug expending. 2006. URL: https://www.istat.it/it/files//2018/07/2006_16.pdf.

[11] ISTAT. *Caratteristiche demografiche e cittadinanza*. Gathered info: population of every city in Italy. 2021. URL: https://esploradati.censimentopopolazione.istat.it/databrowser/#/it/censtest/categories/BULKDOWNLOAD.

[12] della salute e delle politiche sociali Ministero del lavoro. *Anagrafe delle farmacie italiane*. Gathered info: name, location and others information of all pharmacy in Italy. 2012. URL: http://www.datiopen.it/it/opendata/Farmacie.

[13] Kuhn M|Letunic I|Jensen LJ|Bork P. *The SIDER database of drugs and side effects*. Gathered info: Indications and Side effects associated to ATC codes and active ingridients. 2015. URL: http://sideeffects.embl.de/download/.

[14] John Page. *Is a Document Database Faster than an RDBMS?* Gathered info: MongoDB reads per second. 2023. URL: https://medium.com/@johnlpage/is-a-document-database-faster-than-an-rdbms-b731b85365e4.

[15] Pubchem. *Explore Chemistry with Pubchem*. Gathered info: IUPAC and SMILES code from active ingredients. URL: https://pubchem.ncbi.nlm.nih.gov/.

[16] Redis. *Redis configuration file example*. Gathered info: Configuration commands for Redis persistence. URL: https://redis.io/docs/latest/operate/oss_and_stack/management/config-file/.

[17] Redis. *Redis persistence*. Gathered info: Persistence policies for Redis. URL: https://redis.io/docs/latest/operate/oss_and_stack/management/persistence/.

[18] Redis. *Scripting with Lua*. Gathered info: How to implement Lua script. URL: https://redis.io/docs/latest/develop/interact/programmability/eval-intro/.

[19] rlib. *MongoDB 80,000+ insertions per second*. Gathered info: MongoDB writes per second. 2015. URL: https://serverfault.com/questions/743285/mongodb-80-000-insertions-per-second.

[20] Ministero della salute. *Ministero della salute-Opendata*. Gathered info: list of active ingredients. 2011. URL: https://www.dati.salute.gov.it/.

[21] Paolo Sarti. *Lista cognomi italiani*. Gathered info: common italian surnames. 2017. URL: https://github.com/PaoloSarti/lista_cognomi_italiani.

[22] Danilo Tezzutezzu. *Nomi italiani*. Gathered info: common italian names. 2019. URL: https://gist.github.com/tezzutezzu/8f025345cadc5f92b9b311bf032b264d#file-data-csv.