deti universidade de aveiro departamento de eletrónica, telecomunicações e informática

TQS: Product specification report

Alexandre Lopes [88969], André Amarante [89198], João Nogueira [89262], Tiago Melo [89005]
v2020-06-04

1 Introduction

1.1 Overview of the project

This assignment was proposed by Professor Ilídio Oliveira in the scope of the Software Quality And Tests course. In this project, we aimed to develop a platform whilst ensuring and enforcing the quality software development practices taught throughout the course: testing (Unit, Integration and Functional), static code analysis, code coverage, quality gates, the GitHub flow, frequent code reviews, Continuous Integration and Deployment, and, most importantly, Agile Development: rapidly responding to changes through weekly sprints and meetings, keeping everyone up to date with the state of the project.

Our vision and proposition for this project is ReCollect: a marketplace where collectors can finish their old collections or start new ones. Our application's main goal is to provide ways for collector communities to trade this sort of items and to allow them to easily search for whatever they need to achieve any completionist dream.

The developed API focuses on easing access to the item resources as well as the users who posted them. For this reason, it allows a wide range of queries for the former, while enforcing limits in order to prevent excessive results and bandwidth consumption. Further details and documentation on the API will be explained further on.

We also built an Android app representing an external client that uses our REST API. This consists of a bookstore that acts as a partner of our platform and therefore calls our API to show books, which is a category of item that our platform supports, comments, and user details to their customers.

Because of this external client/partner concept, the Android app only uses GET requests from our API and doesn't actually write to our database.

1.2 Limitations

ReCollect focuses on providing a unified marketplace interface that links the producer to the consumer in a flexible environment where any given user can be both. In other words, a customer becomes a producer once they upload an item to the platform, and that same customer can be a consumer if they are looking for products to buy.

That being said, our application currently does not support an internal payment gateway. Since item transactions are done outside the platform, we considered it would be unsafe and insecure to have two halves of the same transaction occurring in different places (payment inside platform versus item outside the platform). The intended interaction for customers would be to find an item they like, contact the seller via email or cell phone using the information made available by our platform, and from then on schedule how the trade could be processed.

Another limitation of our platform is that it does not support direct messaging. Users can and are encouraged to communicate between themselves, but we considered that the MVP would not need such a feature.

2 Product concept

2.1 Vision statement

reCollect is a platform that allows users to sell their old collections, from simple things such as figurines sold some years ago in stores to more rare items like limited editions of a certain product (book, game, etc.). Here, the buyers can find items they are interested in and then finish their collections or even start a brand new personal collection!

This platform is born due to how hard it is to find rare items and "relics" on the internet, mainly items and collections that people built as kids (but not limited to) and want to bring back the collector they have inside and relive some old memories. From the sellers' perspective, many people have such items stored and want to get rid of them, whether because they want to free some space or just because those items are not useful or don't mean anything anymore. So, many times, those collectibles end up in the trash (which not only creates more waste but also does not return to people the investment they made in those products)… So, to solve both buyers and sellers problems, reCollect comes as a place focused on collectors that want a centralized place where they can find and get collectibles more easily; as well as on those people that have some old collections stored, getting dust, and want to make some money out of them.

This concept may recall some other similar stores like OLX or eBay. However, those platforms don't specialize in collectible items and such items end up in the pile of items those platforms display together with some other goods that are not collectibles, rare items nor hard to find around the internet. As reCollect focuses on collectors, it serves as a unique platform to find, as we already mentioned above, items that are more rare and old.

2.2 Personas

Francisco Faria

Francisco is a twenty-four-year-old man from Ílhavo and is finishing a Master's Degree in Marketing at the University of Aveiro and working in a part-time job as a writer in a games blog. He lives alone and is a big fan of movies and comic books. When he was a kid he always stayed fascinated by the superheroes and their adventures in the comics he was reading and since then he has been building a big collection of heroes comics and their figurines as well that their parents bought him. However, as he grew up, some stores around their area closed and their parents had to save money due to his dad losing his job, which made it impossible for their parents to continue to spend money in his collection.

Now, as Francisco is now independent and has a job, he would like to spend some of his savings to finish and continue his collection, mainly because he heard of some limited editions of comics that were sold when he was fifteen. However, Francisco has been searching in many websites around the Internet and is having hard times to find the items he wants and, when he finds them, most of them are still new and very expensive. Since he just wants to finish his collections, used items in good conditions would also fit.

Motivation: Francisco would like to have an online place where he could find affordable collectibles, secondhand or not, in good conditions in order to finish his collections

Alexandra Ramos

Alexandra is a twenty-seven-year-old woman from Portimão and has been working as a researcher in an Aquaculture Institute in the Algarve since she finished her Masters in Marine Biology at the University of Algarve. When she has time for it, she likes hiking and riding her bike around the Ria Formosa. Five years ago she met João, his boyfriend, who came from The Azores to study at the University of Algarve as well. João is two months from finishing his studies and has not yet a fixed and stable job, as he jumps from part-time jobs to part-time jobs. Alexandra, however, has received an invitation to work in a project linked to marine biodiversity conservation in the Azores and the couple is planning to move there once João obtains his degree. Since they don't live together and are planning to buy a house soon, Alexandra thinks this is a good opportunity to start their life as a couple in João's homeland.

As Alexandra sets things up to move along with João to the Azores, she finds some old collectibles stored that she bought with her savings when she was younger and realizes that she can't take it all to the house in the Azores as she will be sharing it with João. So, Alexandra is a little confused about what she should do with those materials: concerned about nature and the environment as she is, she does not want to throw them in the trash, so she decided to sell those items to get back some of the money spent on them, but does not know anyone who could be interested in her items.

Motivation: Alexandra would like to find a platform where she could find people that enjoy collecting old items that can be interested in buying some old collections she has been storing since a kid and is not interested in them anymore.

2.3 Main scenarios

To organize this project's development process, and after the elaboration of personas, the system's functionalities were organized into scenarios that describe how the system helps the personas representing the users fulfill their motivations and needs. The following are just a few examples:

Alexandra wants to get rid of an incomplete book collection - Alexandra has an incomplete Tintin comic book collection that she started as a kid but never finished. While cleaning her book collections, she realizes she is ready to let go of this partial one so she opens the *reCollect platform* and logs in. She then proceeds to announce her collection, filling out the required information, uploading a couple of photos and deciding on a price before officially publishing it.

Alexandra sells her book collection - Alexandra has had her Tintin book collection up in the *Recollect platform* for about a week now. She received a few comments with questions and has interacted with them. Yesterday, she received a call from Francisco Faria, a gentleman who's from the north of Portugal, and is spending a week in the Algarve area. He told her he was interested in buying the collection. They set up a meeting at a café for the next day, and Francisco ends up buying her collection. As Alexandra gets home, she realizes she doesn't want other people to still think the collection is for sale, so she logs into the platform, accesses her profile, goes to her "products for sale" list, and marks the collection as sold.

Francisco has some incomplete book collections - Francisco is a big book collector but unfortunately still has some incomplete collections from his childhood that he just can't seem to find in any bookstore. Whilst on vacation in Faro, he was complaining to a friend about it and was told about the *reCollect platform*, a website where you can sell and buy objects but that only allows collectible items. That night, Francisco visits the platform, creates an account, and starts exploring the home page. He sees a bunch of categories displayed and clicks the "Books" category. Still not finding the

Tintin collection he is looking for, he enters a search term “tintin” to the search bar. Francisco then sees what he’s looking for as Alexandra is a user selling the books 4-10 of the collection, a part that Francisco is missing. He clicks the product, sees her email and number displayed, and decides to give her a call.

2.4 Project epics and priorities

After an initial brainstorming of features that needed to be achieved developing this platform, we then mapped them into user stories and started focusing on the ones that needed to be completed to achieve certain goals. For example, if we wanted that a user would be able to announce a product in reCollect, it must have registered, logged in, and only then be able to post details about a product it wants to sell. So, for that example, we had the following epic: “Publish products to sell as a user”. This epic was the broken down into smaller tasks, the user stories, as so:

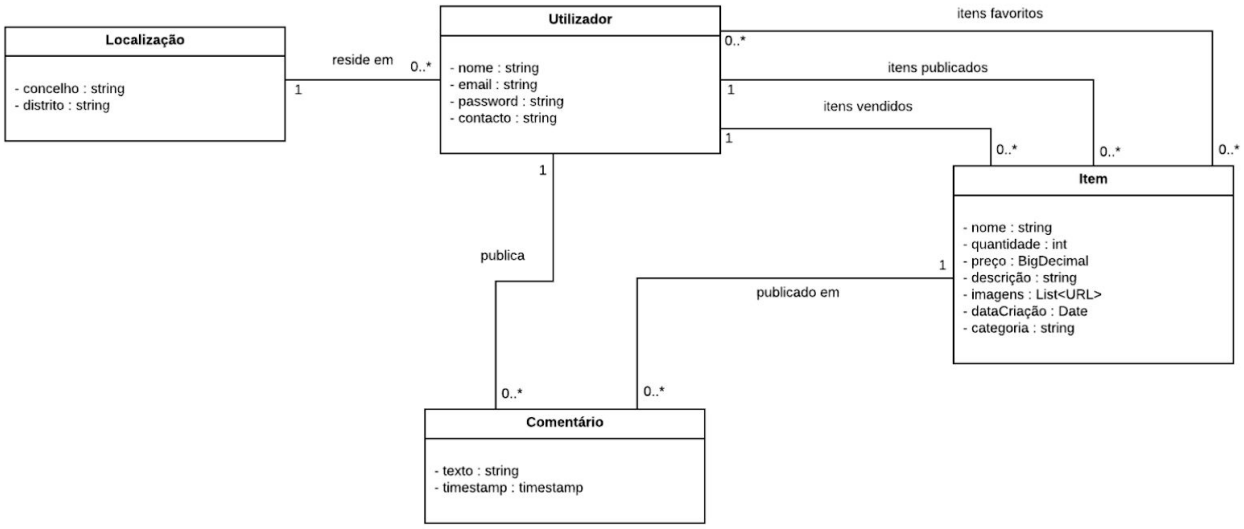
Epic: Publish products to sell as a user		
Story: Francisco wants to create an account in the platform	Story: Francisco, who is already registered, wants to log into his account	Story: Francisco wants to add a product to sell in the platform

This way, we were able to build a flow of development, focusing on what needed to be done in order to achieve a certain goal. By breaking epics into smaller user stories we were able to map user needs to tasks and assign them to a team member (or members) according to the effort estimated and priority, and integrate the results so that the main goal, defined by the epic, could be fully achieved.

To prioritize tasks, we analyzed the backlog and discussed which user stories would be the most critical for the core concept of our application and what would be the effort to implement and test them, addressing in the first place the more critical ones for our context and others that would take more time to develop and could influence the development of the remaining (such as implementing authentication - registration and login - dictates if the user will be able to publish a product for sale or not).

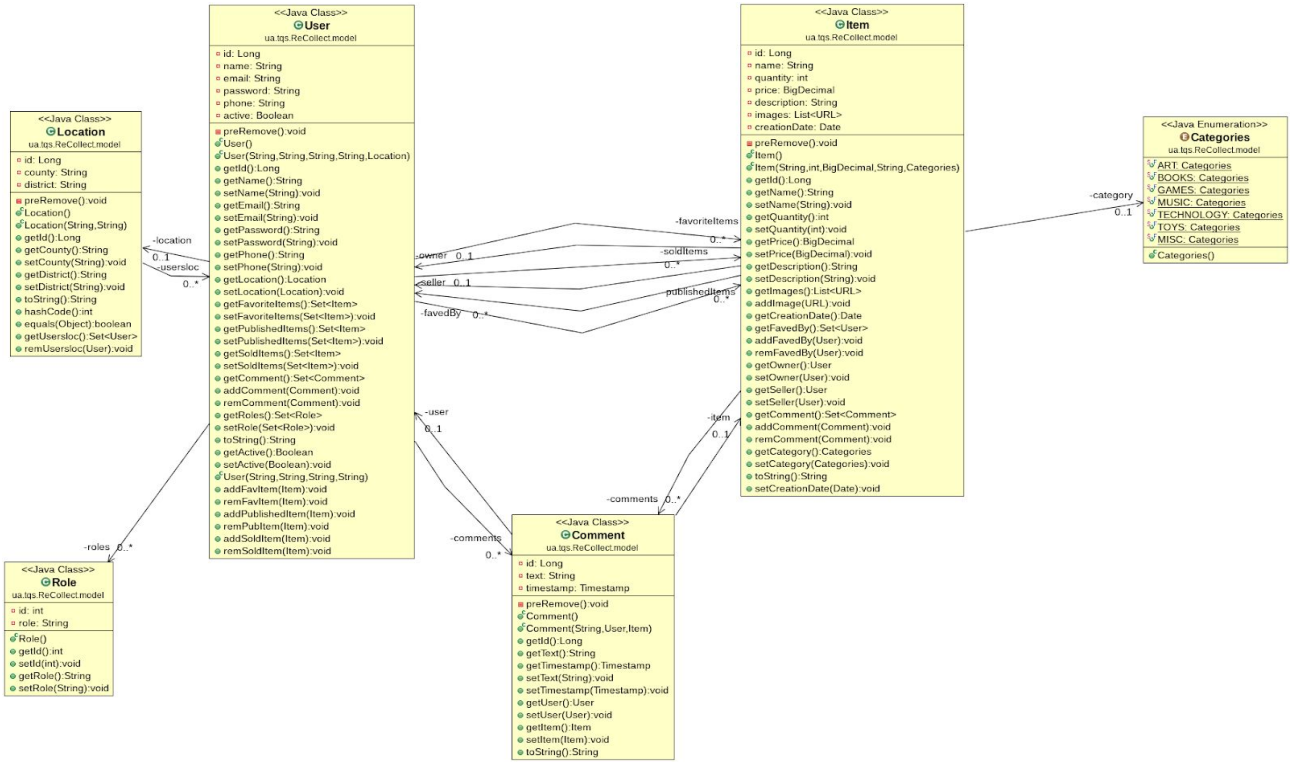
3 Domain model

The project’s domain model consists of the four main entities represented in the below diagram.



UML Class Diagram

Regarding the database itself, the next diagram includes two additional classes, a Role class, needed in order to implement the Spring Security mechanisms, and a Java Enumerate Class, where the different product categories are stored.



Database Entities Class Diagram

The first class, the User, consists of a name, an email (which has to be unique in the database), a password, a phone number (for other users to contact when interested in the User's products), and a Location. The Location, which is a separate class, is composed simply by two strings, one for the county (concelho) and other for the district ("distrito"). This location is important because, since all payments and product delivery are handled manually, by the users, outside the platform, it is useful to query products by location, and know in advance the owner's general location. In the current version, all the 308 Portugal county/district combinations are loaded to the database on application initialization.

The User also possesses three Item lists, the first one for favorited products, in which the user might be interested, a second one for currently on sale/published products, and a third list which contains the already sold user's products.

The Item (product) class, is modeled by a name, description (in which the user can include all the product's specifications), category (corresponding to the Enumerate Class, such as Art, Toys, Technology), the quantity, price, creation/publication date, and a set of image URLs.

The fourth main class represents the comments that can be made by the platform's users on a published product. The Comment consists of the input text, a creation timestamp, the associated user who posted it, and the item in which the comment was made.

The backend-frontend interaction is done by the related repositories, services, and controllers, following a typical Spring MVC Framework.

4 Architecture notebook

4.1 Key requirements and constraints

At the time of writing, the main constraints conceived for reCollect's system architecture consist of:

- Since it aims to provide service to everyday consumers, good system performance is a key requirement. In other words, no client should be kept waiting for a page to load for more than 5 seconds.
- Given that there are currently no hardware dependencies planned, there is not a particular key requirement to ensure strong microservices compatibility. We are also currently not expecting mobile support since at the time of writing the goal is to provide a single user-interfacing medium: a web app.
- In order to ensure high availability and maximize uptime, the system must be robust. Furthermore, since long-term maintenance is also a goal, this requirement is of utmost importance.
- The system should also be scalable. To meet undefined customer needs, such as posting a variable number of items or an undefined number of clients being served simultaneously, we need to ensure the system is able to scale without being too technically expensive.
- We are currently not expecting random peak user loads, hence dynamically handling system performance for such occasions is not a priority.
- The system must ensure there is no unauthorized data access. There can be no data leaks and users must be able to trust the platform.
- Users must be authenticated to post items to the platform but not to browse already available and posted items.
- Being a client-server application, reCollect must provide a User Interface that will be accessed on the client's behalf and provide a server that will be unknown to the user, operating on a Virtual Machine provided by UA.
- Given that the application will run on a typical system, no complex deployment overhead should be necessary.
- Being a Spring Web App, the system architecture must comply with Spring Framework standards.

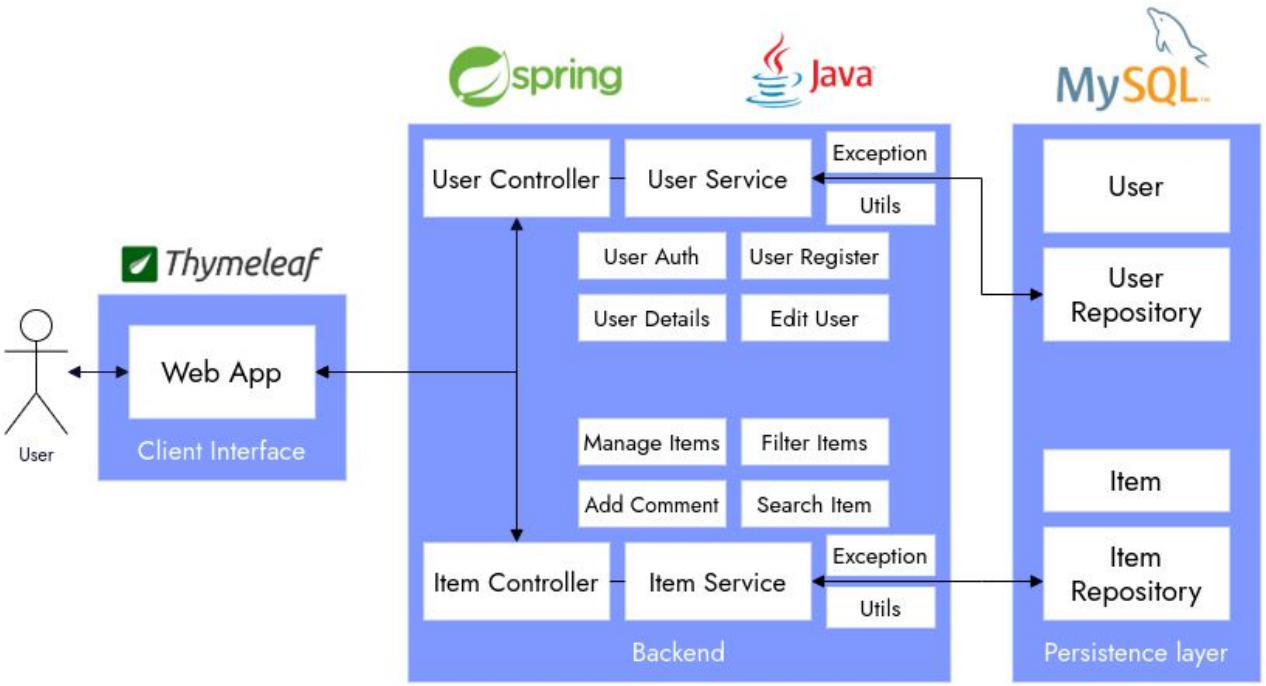
4.2 Architectural view

The software solution should follow the MVC architecture pattern. In other words, the Web App should comply with the standards demanded by such. These include: the Model, responsible for handling and managing data, the Controller that should expose and receive data to/from users or other developers, and the View component that should provide a means of interaction between the user and the system.
The architecture of a generic Spring-based solution can be broken down into 6 main modules:

- **Entity** - contains the relevant POJO's for the context of the application. In this case, it will contain the User and Item classes, which will be stored persistently.
- **Service** - contains the classes that will handle the business logic.
- **Repository** - contains the interfaces needed to interact with the persistence layer.
- **Controller** - the controllers necessary to map endpoints and build the API.
- **Exception** - contains the exceptions that can be thrown and error messages.
- **Utils** - misc classes needed to perform different operations that might be necessary.

As to the modeling of entities, our system aims to provide service to two main models. The user, which is a client that is allowed to browse and publish products on the platform; and the item. The item consists of an abstract class that encapsulates the common attributes that different collectibles own, such as title, description, price, number of units, and so forth. This type of architecture will prove to be extremely useful since it provides a convenient layer of abstraction through polymorphism. Each collectible item will inherit and extend this class and add its own attributes, for instance, for a Coin maybe it is relevant to add the date it remounts to, as well as the press, edition, or country. This means a direct implementation of each class that will be supported on the platform. However, after discussion, we have concluded this would not be an overly daunting task, since at the time our platform aims to serve only the mainstream collectible items.

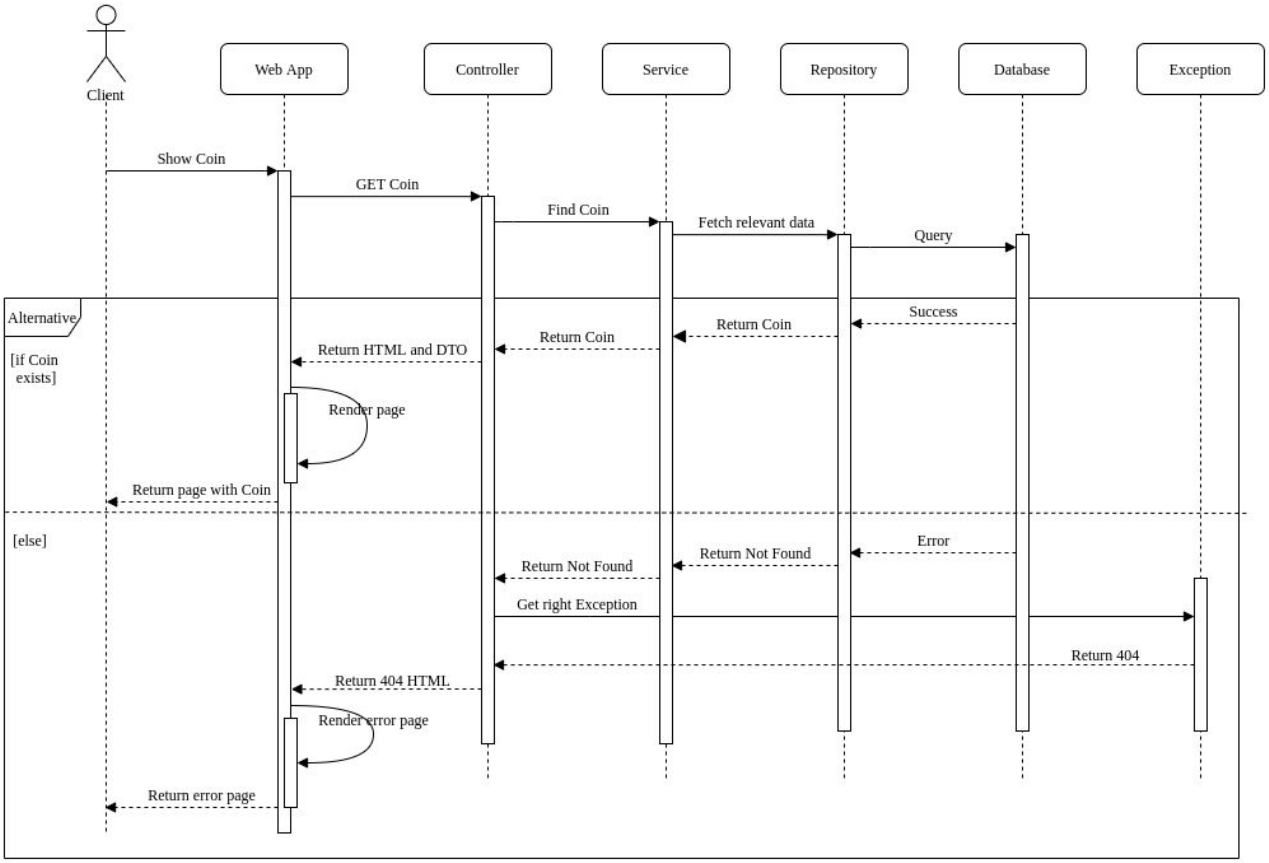
Our architecture follows this structure with a couple of tweaks. Since we will have 2 core entities in our application, we have decided to split the system into 2 cores: one responsible for managing users that register themselves in the app and are able to publish products and another one responsible for managing the items that are stored in our application. Each entity is mapped and managed by its own Controller which interacts with a Service that will have the implementation of the features that regard a specific entity. At the time of writing, for the user side, these include user authentication and registry as well as managing user details; whereas for the product side of the system, the main features to be implemented include managing (such as retrieving and saving) items, filtering and searching for products by category or name and adding comments to published items. It is worth mentioning that both cores will interact with the MySQL database using their own specific repository interface.



Architecture Diagram

A normal workflow of the application would consist of a client wanting to see information on the website or to add new information to the website. That being said, since the primary way of interacting with the system will be through the web application, a user will start off by making a request via the website. In other words, the website will lead the user to the right endpoint depending on the information they want to see. This request will be mapped to an endpoint by the controller module which will take into account the type of HTTP request and the endpoint name. If the request isn't mapped to any available endpoint, a call to the Exception module is made in order to properly handle the error. Afterward, the Controller will make the appropriate call to the Service module, which is responsible for the business logic. The service layer will then interact with the Repository in order to fetch the relevant information that is to be made available or to persistently store new information. Depending on the type of request, calls to Utils can be made for intricate operations or handling misc situations. Once the request reaches the Repository, a proper call to the Database is made using an interface and the Entity object is retrieved. The database will consist of instances of Entity POJOs. If the database operation was successful, then its output is returned to the Service, which returns it to Controller, which returns it to the Web App, which will render an appropriate HTML page using Thymeleaf. If, on the other hand, the object requested is not found or the database operation is unsuccessful, then another interaction with the Exception module is made, in order to properly handle the error.

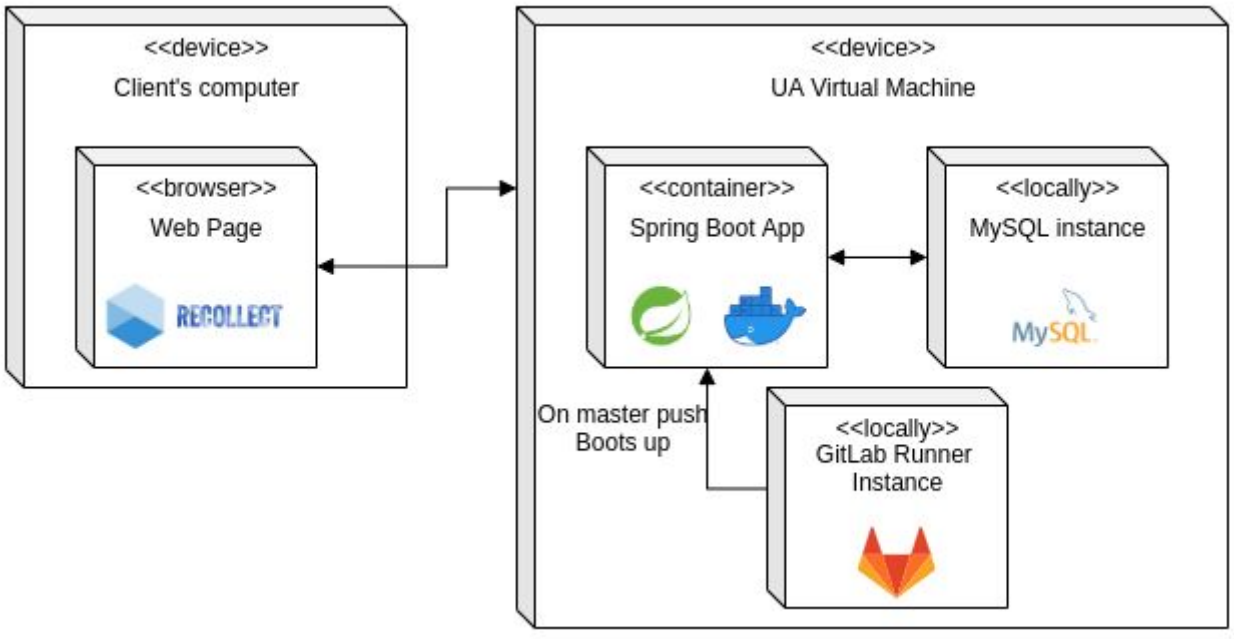
To understand better how these modules interact with each other, let us consider the sequence diagram regarding a Use Case where our client wants to see information on a given product. Let us call this product Coin.



Sequence Diagram

4.3 Deployment architecture

In order to deploy the application, the system will be split across Docker containers, which will communicate with each other in the VM production environment. In later stages, this deployment will be done continuously and automatically for every update on the master branch.



Deployment Diagram

5 API for developers

The API we developed for this project focuses on supporting services and applications that aim to serve clients of a specific collector's niche, for instance, vintage technology or book collectors. The Interface does not aim to allow users to leverage off of our Database or hosted services, it focuses on helping other developers enriching the content displayed on their platforms. For these reasons, the endpoints made available support only read-access to the resources on ReCollect.

Our API currently supports three top-level collections: items, sales, and users; each of which is preceded with the conventional /api/ path. The automatically generated Swagger documentation for the API can be accessed here.

Item Collection

The Item Collection is composed of two endpoints. The /api/items endpoint allows querying the database based on five parameters:

- the **category** of the item: since every item is assigned to one category, it is possible to access all the items from a given one; for instance, to fetch every book posted on the platform the developer can issue a request to /api/items?category=BOOKS
- the **owner** of the item: the user's email whose items the developer wants to fetch, delimited by apostrophes. For example, /api/items?owner='mail@mail.com'
- the **ordering parameter** for the fetched results: currently, the API only supports sorting in ascending order. The retrieved items can be sorted according to their price or creationDate. This sorting is done using a custom implementation of a Pageable object, which proved to be extremely useful in order to allow sorting, limiting, and paging through results using a single argument. (source)
- the **limit** which serves as a custom maximum of results parameter. This parameter has a default value of 25 which is used when no limit is specified or when the passed limit surpasses the default one, in order to avoid overly large requests at once. This limit is imposed by the OffsetBasedPageRequest.
- the **offset**: the number of results to skip. This allows users to page through results and iterate over them whilst establishing a reasonable limit. The above implementation for the Pageable object also allowed this by passing on an offset attribute.
- **sold** is a boolean value passed on to specify whether or not the retrieved items are on sale or have already been sold. The introduction of this parameter deprecates the sales collection, but we considered it relevant to maintain the documentation about it.

It is worth mentioning that this endpoint supports any combination of these parameters or none at all.

On the other hand, the /api/item/id endpoint allows developers to access a more detailed view of the items found when using the above endpoint to search for them. The id path variable, as suggested, is the ID of the item that the developer wants to access.

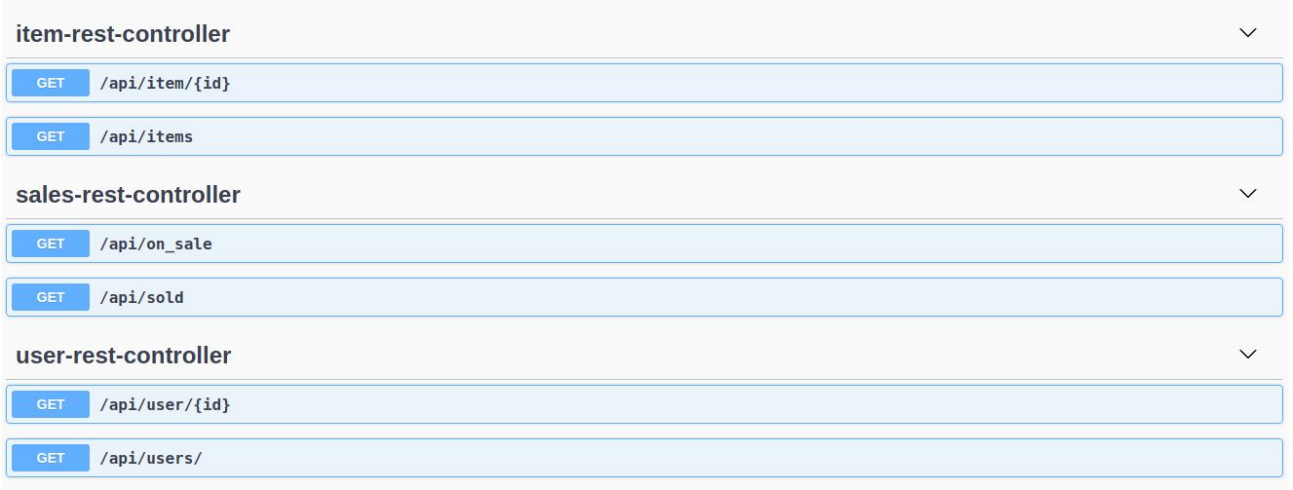
Sales Collection

We considered it would also be relevant to develop endpoints that provide access to only sold items or only items that are available for sale. Both of these endpoints accept only the limit and offset parameters that behave similarly as the ones described on the /api/items endpoint. As to the

introduction of the sold parameter on /api/items, this collection is redundant and hence deprecated, since the said endpoint also enables this sort of results whilst allowing additional filters. If a developer needs access to every item on sale, they need only pass along every parameter as null except **sold**, which should assume false. We considered it would be relevant to document and keep it, however, to show the evolution of the project through time.

Users Collection

Lastly, the users collection is composed of two endpoints, similarly to the items collection. The /api/users/ endpoint is composed of 2 parameters besides the offset and limit ones, that work as already explained. The developers are free to query for users at a given location. It is worth mentioning that a location is identified by a district and a county. This search can only occur if both are supplied. The /api/user/id exposes further information about a given user when passed along their ID, which can be found via platform or via the other endpoint.



6 References and resources

- Spring
- Spring Initializr
- MySQL
- GitLab CI/CD
- Docker
- Thymeleaf
- Tutorial: Thymeleaf + Spring
- Thymeleaf + Spring Security integration basics
- Swagger
- Spring Security (medium)
- Gitlab CI/CD using Docker (vid)
- Sorting and Paging through DB registries (StackOverflow answer)
- Swagger and Spring Boot auto generated documentation (Baeldung article)
- ThemeFisher: Bootstrap and HTML5 Templates
- Agile Epics: Definition, Examples, & Templates