1. Introduction

This report was developed in the scope of the ARQSOFT Course of the Master's Degree of Software Engineering at ISEP. The main goal of this report is to present the architectural redesign and migration of the ACME software application into a microservice architecture.

This work was done by the following student:

- 1200613 - Joao Lucas Roesner

This report is divided into 4 sections:

- Section 1 - Architecturally Significant Requirements: This section presents the architecturally significant requirements, the quality attributes and the constraints of the system.
- Section 2 - Attribute-Driven Design: This section presents the architectural redesign and migration of the ACME software application into a microservice architecture.
- Section 3 - Design and analys: This section presents the design and analysis of the system with the new architecture and drivers discovered in the previous section.
- Section 4 - Implementation: This section presents some of the implementation work done for the system.

2. Architecturally Significant Requirements

For the scope of the ADD approach that this iteration of the project is following, the ASRs will be used as Drivers of the archtecture, later being referenced as such in other sections of this report.

2.1 Use Case Model

Use Case	**Description**
UC1 - Get catalogue of Products	A user can get a list with all Products that were inserted into the system.
UC2 - Get Product from SKU	A user can insert the SKU of an specific product and that should return the product if it exists.
UC3 - Create a Product	A user can create a new Product to be sold.
UC4 - Get Reviews	It's possible to see a list of reviews.
UC5 - Add vote for a Review	In a review, the user should be able to send a vote, the vote can be either upvote or downvote.
UC6 - Post a Review	A user can post a review related to a product.
UC7 - Accept or reject Review	When a review is created, it is presented in a 'Pending' status, the user then can approve or deny the review to publish it.

Use Case	Description
UC8 - Create User	The admin can create new Users and add them into the system.
UC9 - Get User	The admin can get a specific user information by inserting their Id.
UC10 - Publish product after validation	As product manager, I want to publish a Product (i.e. available for clients, reviewers, voters) only aVer two other product managers accept it.
UC11 - Publish review after validation	As a reviewer, I want my Review is published (i.e. available voters) only if it is accepted by two other users that are recommended with that review

2.2 Quality Attributes

In addition to Use Cases, the group also identified the most relevant quality attributes for the system, and the most relevant ones are listed below:

ID	Quality Attribute	Scenario	Associated Use Cases
QA-1	Modifiability	A new SKU generation algorithm is implemented as part of a new client requirement. The system must successfully integrate the new algorithm without breaking the existing functionality and without changing any other component of the system.	UC-3
QA-2	Modifiability	A new Review listing algorithm is implemented as part of a new client requirement. The system must successfully integrate the new algorithm without breaking the existing functionality and without changing any other component of the system.	UC-4
QA-3	Security	A user creates a new review, it's possible to know who created the review 100% of the time.	UC-6
QA-4	Performance	During busy hours, the system can hit peak loads, when that happens, 100% of reviews posted must be successfully captured and stored.	UC-6
QA-5	Availability	If an error occurs with the persistence mechanism, the system must be back to normal within less than a minute.	All
QA-6	Usability	User can display the list of pending reviews and select which one to update, the update should consist of both approve or reject the review.	UC-7
QA-7	Modifiability	Implementation of new functionality must be easily done during development	All
QA-8	Security	Only a logged in user can do action inside the application	All

2.3 Constraints

Some Constraints were also identified, and are listed below:

- CON-01: The system must support the integration of multiple persistence mechanisms, such as: MongoDB, Redis, Neo4j and MySQL.
- CON-02: The system must support the integration of multiple SKU generation algorithms.
- CON-03: The system must support the integration of multiple Review listing algorithms.
- CON-04: The choice of which implementation to use regarding persistence, SKU generation and Review listing must be defined at the application properties.
- CON-05: Injection of dependencies must be done through the use of Spring Framework.
- CON-06: Arquitectural Style: Microservices
- CON-07: Incremental Migration
- CON-08: Function calling inside the microservice must be done through REST API
- CON-09: The system must be deployed using containers.
- CON-10: Data must be migrated between databases.

2.4 Architectural Concerns

Considering the scope of the project, only a few concerns were raised, and they are listed below:

- CRN-01: The work is being done by only one student, so the workload is not well distributed.
- CRN-01: Complexity in implementing the microservices architecture.
- CRN-01: Difficulties in implementing the API Gateway.
- CRN-01: Difficulties in drawing the scope of the project alone.

3. Attribute-Driven Design

Attribute-Driven Design (ADD) is an architecture-centric, iterative, and incremental approach to software design. ADD is based on the identification of a set of architectural drivers, which are the quality attributes that have a major influence on the architecture of a system. ADD uses drivers to guide the design of the software, these drivers were identified in the previous section, being them the Use Cases, Quality Attributes and Constraints and Concerns. New drivers were added but it was decided to keep the old ones as well, as one of the main objectives of this project is that the system should not break any existing functionality, and the API available for outside users should remain the same.

The implementation of the ADD design was done using the brownfield approach.

3.1 Brownfield Approach

Brownfield development is a term used in software developement for the process of developing new software from an existing system, where any implementation of new features is done on top of existing code. This can produce some advantages, such as less work and time spent on developing a new system, and the reusability of existing code. But it can equally produce some disadvantages, such as the difficulty of understanding the existing code, and the difficulty of implementing new features without breaking the existing ones.

Brownfield Approach

It was decided that the work would be separated into 3 iterations, where the first one would be an analysis of the scope of the project and how to implement the necessary changes to accomodate for the new drivers, the second one is for detailing the use of microservices, API Gateway and the database migration, while the third and final iteration is for addressing some final drivers and the 2 new use cases.

3.2 Iteration 1

The goal of this iteration is to analyze the legacy code, identify the changes needed and draw the new changes and appraches to the architecture.

3.2.1 Legacy and Scope Review

Taking into consideration that the project is being built on top of an existing system, it was necessary to review the scope of the project and the legacy system, to understand what changes are necessary. These changes were added as drivers in addition to the ones identified in the previous project.

3.2.2 Architectural Drivers

The Drivers related to the first iteration:

- CON-06: Arquitectural Style: Microservices
- CON-07: Incremental Migration
- CON-08: Function calling inside the microservice must be done through REST API
- CRN-01: The work is being done by only one student, so the workload is not well distributed.
- CRN-01: Complexity in implementing the microservices architecture.
- CRN-01: Difficulties in implementing the API Gateway.
- CRN-01: Difficulties in drawing the scope of the project alone.
- QA-07: Implementation of new functionality must be easily done during development
- All Use Cases

3.2.3 Architectural Decisions

The system will become decoupled into multiple microservices, and each of those must follow a single responsability and be independent from each other. So the idea is to stick with the Single responsability principle, and divide the system that way.

There is two approaches to this method, the first one is to divide the system by domain, and the second one is to divide the system by business logic.

After consideration, the best idea seemed to be to divide it by domains, as the other option can lead into a lot of unwanted complexity that would not be feasible to manage working alone.

3.2.4 Migration Plan

The main concern with migrating the system is that it needs to be always available, which can be difficult considering that changes are done incrementally.

To achieve this then, the use of a Strangler Pattern is used, where a Strangler Facade is put on top of the legacy system, which then can redirect the requests to either the legacy system or the new system, depending on the functionality that is being requested.

That way, the system can be migrated incrementally, because whenever a new microservice is created, it can be added to the Strangler Facade and replace the legacy code for that responsability.

Strangler Pattern

In the context of the Strangler Pattern, the Strangler Facade is the API Gateway.

3.2.5 API Gateway Plan

API Gateway is a design pattern that basically provides an entry point for all microservices, the way it works is that the gateway becomes the main URL for the system, and each request it receives is then redirected to any of the microservices that are behind it depending on the type of the request and the domain it relates (in this case presented by the suffix of the URL).

Additionally, the API Gateway can also be used to implement some security features, such as authentication and authorization.

API Gateway

The API Gateway will be implemented using Spring Boot Gateway, which is a Spring Boot project using the gateway maven dependency that allows the project to function as a Gateway for the microservices.

3.2.5 Database Plan

Databases will also be segregated by domain, with each microservice having a different database. Additionally, the databases then can be separated either logically or physically, depending on the persistence mechanism used. In this case it was opted for a logical separation, where each database is part of MongoDB, but each database has a different collection.

Database Plan

It's important to notice that ideally, each database would be using one of the different persistence mechanisms implemented last iteration, however, due to time constraint and due to the personal lack of experience of the student (which only worked with MongoDB before), it was decided that all databases would be implemented using MongoDB.

CQRS Remark

CQRS Pattern separates the database into two different databases, one for reading and one for writing. This pattern is used to improve performance, as the reading database can be optimized for reading, and the writing database can be optimized for writing.

CQRS

However, due to lack of time and the complexity of the work when working alone, this could not be implemented in time.

3.2.6 System-to-be

After the analysis, it's possible to draw the new system, with responsabilities alocated, interfaces defined and the communication between them.

3.2.6.1 Domain Model

Domain Model

The domain model was separated into 4 different aggregates, each with only one root, and these will each be turned into a microservice.

- Product: This aggregate is responsible for all the functionality related to products, such as creating, updating, deleting and getting products.
- Review: This aggregate is responsible for all the functionality related to reviews, such as creating, updating, deleting and getting reviews.
- User: This aggregate is responsible for all the functionality related to users, such as creating, updating, deleting and getting users.
- AggregatedRating: This aggregate is responsible for all the functionality of aggregated rating.
- Additionally, one microservice for authentication was created, to separate it from the user domain, as it was decided that it was different enough to be its own microservice.

3.2.6.2 System Context

An updated version of the component diagram was created to reflect the microservice implementation.

System Context

It's possible to see what components where created and how they relate to each other, and the responsability of each can be found in the table below.

Component	Description
API Gateway	It works as an API Composer and the sole endpoint of the application, it's the gateway between all microservices, once a request is sent to this API, it will verify te path of the request and then select which microservice to send the request to. Used by acme products, acme reviews and acme aggregated reviews. Uses all microservices.
acme Authentication	In charge of authentication and authorization, receives login invites and returns a jwt token. Used by the API Gateway.
acme User	In charge of all endpoints related to users. Used by the API Gateway.
acme Products	In charge of all endpoints related to products. Used by the API Gateway. Uses the API Gateway.
acme Review	In charge of all endpoints related to reviews and rating. Used by the API Gateway. Uses the API Gateway.
acme Aggregated Rating	In charge of all endpoints related to aggregated ratings. Used by the API Gateway. Uses the API Gateway.

3.2.6.3 Deployment View

The application should use containers to deploy the microservices, the tool of choice was docker, and each microservice was built as an image and deployed as a container. Also worth noting that each microservice has its own database.

That way, the new deployment view is as follows:

Deployment View

One of the problems of this approach, is that, because each docker acts as its own machine, if the API Gateway itself is deployed as a container, it will not be able to communicate with the other microservices, as they are not in the same host.

In a real world scenario, the best approach would be to deploy each of the microservices in a server host, and not running locally on docker.

3.3 Iteration 2

The goal of this iteration is to show how the migration of the database will occur and how the structure of the microservices will look like.

3.3.1 Architectural Drivers

The Drivers related to the second iteration:

- CON-06: Arquitectural Style: Microservices
- CON-07: Incremental Migration
- CON-09: The system must be deployed using containers.
- CON-10: Data must be migrated between databases.
- CRN-01: Complexity in implementing the microservices architecture.
- CRN-01: Difficulties in drawing the scope of the project alone.
- All Use Cases

3.3.2 Database Migration

The database migration needs to be done incrementally, with the API always available, to achieve this, the new micro service was first pointing at the old database, with that pointer being setup in the application properties of the app, then, a new database was created and the data was migrated into it, that way, the application properties were now able to reflect the new database, without any changes to the code.

Database Migration Database Migration2

3.3.3 Data Migration

Because of time constraints and working alone, it was decided that the data migration would be done in the simplest way, by bulk insert, which consisted of exporting all the data from one domain in the old database and manually inserting it into the new database pertaining to that domain.

Other approaches that are much better in term of security and performance (in large scale projects) would be to use a bespoke migration tool or through a data validating service that guarantees that junk data would not get in the new database.

With the bulk approach there are some advantages however, which is for example, that we can guarantee that both databases would be in sync in regards to data for that model.

3.3.4 Microservice structure

The structure used by the microservices can be seen in the image below:

Microservice Structure

Worth noting that this implementation permits the abstraction of the persistence mechanism, in a similar way that was implemented in the previous project.

3.3.3 Technologies

The project used spring boot for the microservices, and spring boot gateway for the API Gateway, and the database used was MongoDB.

3.4 Iteration 3

The goal of this iteration is to address the expandability, authentication and the implementation of the new use cases.

3.4.1 Architectural Drivers

The Drivers related to the third iteration:

- UC10 - Publish product after validation |
- UC11 - Publish review after validation
- CRN-01: The work is being done by only one student, so the workload is not well distributed.
- QA-07: Implementation of new functionality must be easily done during development

3.4.1 Expandability

Taking into consideration the abstraction of the persistence mechanism and the use of the API Gateway for routing requests, we can conclude that the Quality Assurance QA-07 was achieved.

No changes to the code will be needed to change functionalities, databases, or to add new microservices, as long as the new microservice is added to the API Gateway.

3.4.2 Authentication

Authentication is done through the use of JWT tokens, which are generated in the authentication microservice, and then stored locally on the API Gateway, to be used in request headers for other requests as a form of authentication.

Authentication

3.4.3 UC10 and UC11

UC10 and UC 11 were two new functional requirements added into the project, and were both implemented:

- UC10 - Publish product after validation: This use case was implemented by adding a new field to the product model, which is the approve Count, then, a new endpoint 'approve product' was created, which takes the product sku and role of the user and increments the approve amount if the role given is of type "Product Manager". Finally, a new endpoint was created to get the products, which only returns the products that have an approve count of 2 or more. That way, only products that were approved by 2 product managers are available.
- UC11 - Publish review after validation: This use case was implemented in a similar way as UC 10, a new endpoint was created to approve a review, which receives the review id and user id, the function runs the review algorithm using the userId parameter, and checks if any of the returned reviews have the id equal to the review id parameter, if it does, then it increments the approve count of that review, and if the approve count is equal to 2, then the review also has it's status changed to 'Approved'.

4. Implementation

This section presents some examples of the implementation of the system, and how it works.

4.1 API Gateway

The API Gateway was done with a RouterLocator function which is part of the Gateway maven dependency, and allows for setup of each route.

API Gateway

4.2 Authentication

Authentication is done in the authentication service and stored in a local variable.

```
.route(r -> r.path("/auth/public/login")
      .filters(f -> f
       .addResponseHeader("X-Powered-By","DanSON Gateway Service")
        .modifyResponseBody(String.class, String.class, (exchange, s) -> {
         String authToken =
exchange.getResponse().getHeaders().getFirst("Authorization");
         headers.put("authToken", authToken);
         return Mono.just(s);
        })
      )
      .uri(acmeAuthentication)
```

and then passed as a header to the other services.

```
.route(r -> r.path("/products/**")
      .filters(f -> f.addRequestHeader("authToken", String.valueOf(new
StringBuilder("Bearer ").append(headers.get("authToken"))))
        .addResponseHeader("X-Powered-By", "Response")
      )
      .uri(acmeProducts)
```

4.3 Abstraction

Abstraction was done for the gateway, deciding the port for each service:

Abstraction

and aswell as for the persistence mechanism:

Abstraction2

4.4 UC10 and UC11

UC10 and UC11 were implemented in the product service and review service respectively.

The logic for UC10:

```
    @Override
    public ProductDTO approveProduct(String sku) {
        final Optional<Product> productToUpdate = repository.findBySku(sku);

        if( productToUpdate.isEmpty() ) return null;

        productToUpdate.get().setApproved(productToUpdate.get().getApproved()+1);

        Product productUpdated = repository.save(productToUpdate.get());

        return productUpdated.toDto();
    }
```

The logic for UC11:

```
        @Override
    public ReviewDTO recommendReview(Long reviewID, Long userID) {

        Optional<Review> rev = repository.findById(reviewID);
        if (rev.isEmpty()){
            return null;
        }
        Review r = rev.get();
        List<ReviewDTO> reviews = getReviewAlgorithm(userID);

        if (reviews == null) return null;

        for (ReviewDTO review: reviews) {
            if (Objects.equals(review.getIdReview(), reviewID)) {
                int count = r.getApprovedCount() + 1;
                r.setApprovedCount(count);
                if(count >=2) {
                    r.setApprovalStatus("approved");
                }
                repository.save(r);
                return ReviewMapper.toDto(r);
            }
        }

    return null;

    }
```

4.5 Communicating between microservices

As mentioned before, the communication between microservices is done through the API Gateway, so in the code, when a service needs logic from another service, a new HTTP request is done pointing at the gateway, and then the result is serialized into the object that is needed.

Here is an example of a request done in the Review service to get a product:

```


    private Optional<Product> getProduct(String sku) {
        try{

            URL endpointUrl = new URL(url + "/products/" + sku);
            HttpURLConnection connection = (HttpURLConnection)
endpointUrl.openConnection();
            connection.setRequestMethod("GET");
            int responseCode = connection.getResponseCode();
            System.out.println("Response Code: " + responseCode);

            BufferedReader reader = new BufferedReader(new
InputStreamReader(connection.getInputStream()));
            String line;
            StringBuilder response = new StringBuilder();

            while ((line = reader.readLine()) != null) {
                response.append(line);
            }
            reader.close();
            connection.disconnect();

            ObjectMapper objectMapper = new ObjectMapper();

            Product product = objectMapper.readValue(response.toString(),
Product.class);

            return Optional.of(product);
        } catch (IOException e) {
            e.printStackTrace();
        }

        return Optional.empty();

    }
```