POLITECNICO
MILANO 1863

Design Document

Deliverable:	DD
Title:	Design Document
Authors:	Leonardo Gori, Marco Romanini, Yui Watanabe
Version:	1.1
Date:	February 15, 2022
Download page:	https://github.com/MarcoRomanini/GoriRomaniniWatanabe
Copyright:	

Contents

List of Figures

List of Tables

1 Introduction

DREAM is an easy-to-use application whose intent is [4] to design dynamic anticipatory governance models for food systems using digital public goods and community-centric approaches to strengthen data-driven policy making in the state of Telangana, India.

The state's main means of livelihood indeed heavily rely on agriculture, widely represented by smallholders farmers' activities. These ones are easily affected by complex problems such as climate change and the incoming raise of food demand due to the continuous population increase. In addition to that, the occurrence of `Covid-19` pandemic is recently causing further obstacles (such as food supply chains disruption) to the achievement of a resilient food system.

The final aim of Telengana's government is collecting and analyzing agriculture related real-time conditions in order to monitor and support smallholders' activity resilience capacity against the above mentioned problems.

1.1 Purpose

This document provides a detailed description of the software architecture chosen for the DREAM application described in the Requirement Analysis and Specification Document.

This includes architectural components and features, information about the deployment and the interfaces and a description of the adopted patterns.

Moreover, a runtime view of the core functionalities of the S2B is provided, together with some interaction diagrams between the different components.

Also some examples of user interface design are provided, in order to give a general idea of how the application will look like.

Finally, there is information about the implementation, integration and testing processes.

1.2 Scope

The system defines three main actors that will interact with the application: policy makers, farmers and agronomists.

Farmers can access the platform to see data relevant for them, such as weather forecasts and personalized suggestions, to insert data about their production and any problem they face, to request for help and suggestions by agronomists and other farmers and to create discussions with other farmers.

Agronomists can insert the area they are responsible of, they can receive and answer to request by farmers, they can see relevant data about their areas, such as weather forecasts, best/under-performing farmers and problems encountered by farmers, and they can also manage daily plans in order to visit periodically the farmers.

Finally, policy makers can visualize information and statistics about the overall process, they can see if the steering initiatives are producing significant results, they can ask well-performing farmers to write good practices and can assign incentives to them.

1.3 Definitions, Acronyms, Abbreviations

In order to introduce some sort of coherence in an environment— the physical world— that is informal, we present in this section the sets of **definitions**, **acronyms** and **abbreviations** used in the following sections. These are supposed to be as more generic as possible, in order to provide more flexibility for the following phases of the Waterfall software lifecycle.

It is important to agree in advance on the specific keywords and terms that signal the presence of a specific entity.

1.3.1 Definitions

Concept	Definition
EVENT BUS	With Event BUS we refer to all the system elements used to handle the database management process, in order to guarantee data consistency across the fragmented datasets. The Event BUS is the result of the implementation of different architectural patterns, such as Database per Service, Event Sourcing, Domain Event, CQRS and Saga.
CLIENT-SERVER PARADIGM	Distributed application structure that splits functionalities between the providers of a service, called servers, and the ones that ask for it, called clients
SERVICE	Indipendent lightweight application that implements narrowly focused functionalities
MICROSERVICE ARCHITECTURE	Architectural style that functionally decomposes an application into a set of services
SOFTWARE INTERFACES	Languages, codes and messages that programs use to communicate with each other and to the hardware
PATTERN	General reusable solution for the common problems that occur in software design

Table 1: Table of definitions

1.3.2 Acronyms

Acronym	Expansion
DREAM	Data-dRiven prEdictive fArMing
S2B	Software to be
DPGS	Digital Public Goods Standard
UML	Unified Modeling Language
DBMS	Data base Management System
API	Application programming interface
GUI	Graphical User Interface
CQRS	Command Query Responsibility Segregation
REST	Representational State Transfer
CRUD	Create, Read, Update, Delete
HTTPS	Hypertext Transfer Protocol Secure
URL	Uniform Resource Locator
DB	Data Base
DDD	Domain Driven Design
JSON	JavaScript Object Notation
TDD	Test Driven Development
SSL	Secure Sockets Layer

Table 2: Table of acronyms

1.4 Revision history

Date	Description
28/12	Architecure Structure (Microservices)

	Implementation, integration and test plan
29/12	Implementation, integration and test plan
	UI diagram
30/12	Architecture Overview
	component view
	Architecure Structure (Microservices)
	UI diagram
31/12	Architectural Styles and Patterns
	component/deployment diagram
	Sequence diagram
	UI diagram
01/01	component/deployment diagram
	Component view section
02/01	Functional requirements
	Architectural Styles and Patterns
	requirement treceability matrix
03/01	Architecture Overview
	Implementation, integration and test plan
	Architectural Styles and Patterns
	Other Design Decisions
	Sequence diagram
	requirements traceability matrix
04/01	Architecure Structure (Microservices)
	Introduction (Purpose, Scope)
	Requirements traceability (software system attributes)
	Other design decisions
	Sequence diagram
	Component diagram
05/01	Integration Plan
	Component Interfaces
	Component view
	Sequence diagram
06/01	Deployment view
07/01	Whole document review
	Components Interfaces
15/02	Requirement map refined
	UI diagram refined

Table 3: History table

1.5 Reference Documents

We present in this section the references we used to gather information about the good practices for the development of this document.

[1] B. Bruegge and A.H. Dutoit. *Object-oriented Software Engineering Using UML, Patterns, and Java*. Always learning. Pearson Education Limited, 2013.

[2] Digital Public Goods community. Digital public goods standard. https://digitalpublicgoods.net/standard/, 2021. Last accessed December 2021.

[3] ISO/IEC/IEEE. Iso/iec/ieee international standard - systems and software engineering – life cycle processes – requirements engineering. *ISO/IEC/IEEE 29148:2018(E)*, pages 1–104, 2018.

[4] LeoGori-MarcoRomanini-y1220. Requirement engineering and design project: goal, schedule, and rules. https://github.com/MarcoRomanini/GoriRomaniniWatanabe/blob/main/01.%20Assignment%20RDD%20AY%202021-2022.pdf, 2021. Last accessed December 2021.

[5] C. Richardson. *Microservices Patterns: With examples in Java*. Manning, 2018.

[6] Parvathy Krishnank Swetha Kolluri. Data4policy. https://github.com/UNDP-India/Data4Policy, 2021. Last accessed December 2021.

[7] Standardisation Testing, Quality Certification (STQC) Directorate of the Ministry of Electronics, and Government of India Information Technology. Standards for e-governance applications. http://egovstandards.gov.in/notified-standards-0, 2021. Last accessed December 2021.

[8] Hans van Vliet. *Software Engineering: Principles and Practice*. Wiley, 2007.

1.6 Document structure

The overall document is organized in 5 main sections. For each of them we provide a brief explanation of the contents of their subsections, except for the current one.

1 Introduction

Section 1.1 offers a brief description of the document content. In section 1.2 we briefly introduce the desired functionalities that the S2B should achieve, organized for each actor involved in the application's domain. Section 1.3 contains the list of definitions of terms and acronyms used along the document. Section 1.4 contains the history of reviewed section of the document during time passing. Finally, section 1.5 provides the list of useful sources that have been exploited to build the document.

2 Architectural design

Section 2 provides a complete description of the designed software architecture. In particular section 2.1 presents the architecture diagram from an high level perspective. Section 2.2 shows the structure of the components that compose the platform as well as the relationship between them through a component diagram. Section 2.3 provides a deployment diagram and describes the tiers of the S2B planned architecture. In section 2.4 we present the sequence diagrams showing the main functional requirements of the application, while section 2.5 introduces the interfaces that handle basic required operations. Finally in sections 2.6 and 2.7 we respectively introduce the main patterns that are useful to architect the application and the design decisions that could lead to improvements of the planned architecture.

3 User interface design

Where we propose a series of diagrams showing the application user interface and its transitions in different use cases.

4 Requirements traceability

This section provides a requirements traceability matrix that justifies the design choices of the platform. In addition, a description of the attributes the software system guarantees is presented in the same chapter.

5 Implementation, integration and testing plan

As the title suggests this chapter is splitted in different subsections that briefly describe the implementation (section 5.1), integration (section 5.2) and test (5.3) plans of the components that make up the architecture.

6 Effort spent

Here we provide the table of the overall efforts building the documents since last update for each team member.

2 Architectural design

2.1 Overview

DREAM will be implemented using a Microservice Architecture and all the architectural choices reflect this main idea. We chose a microservice-oriented approach because it perfectly addresses the necessity of supporting a variety of different clients, the necessity of exposing an API for 3rd parties to consume and the necessity of guaranteeing high scalability and availability in a complex distributed system.

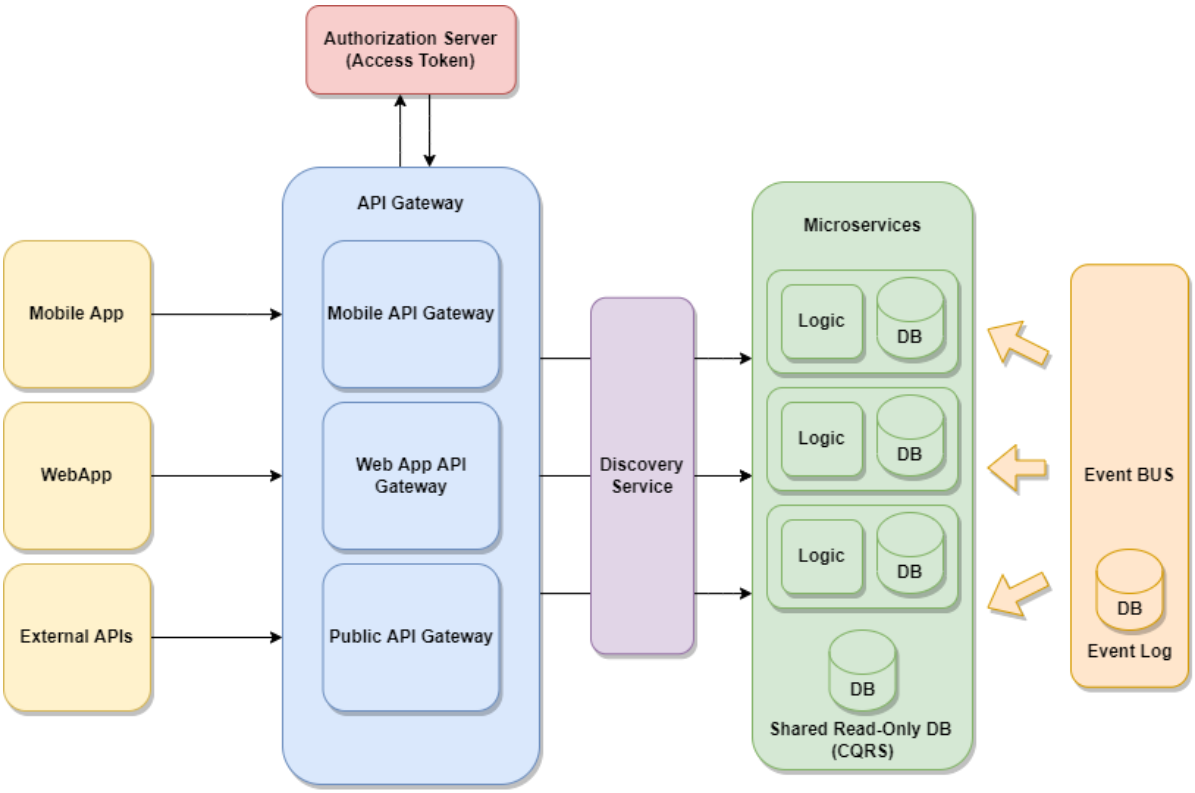

Figure 1: High-level Microservice Architecture

The system is logically divided in three high-level layers:

- presentation layer: it manages the presentation logic and all the interactions with the end user
- application layer: it manages the business functions that the S2B must provide
- data layer: it manages the safe storage and the relative access to data

According to these principles, the final architecture follows the client-server paradigm and is organized in 4 tiers (described in section 2.3).

There are two types of client: the web application and the mobile app. The former is a thin client since it offers only presentation functionalities and depends entirely from the server, which contains all the application business logic. However, the mobile app is a more thick client because it contains an internal database, in order to be more independent from the server.

Figure 1 provides a high-level idea of the structure of the system and the interactions between the components. More precise information will be given in the following sections.

The users can access the service through a web interface or a mobile application. From there, the API gateway contacts the Authorization Server in order to guarantee security across the platform.

After the identity has been checked, the Discovery Service is called to know the exact location of the microservices involved. The user's request is then redirected to the correct microservice in which the business application logic is implemented.

Each microservice has its own database, according to Microservice Architecture standards, and they all communicate through an Event bus (more details about the patterns used to keep these fragmented databases organized can be found in section 2.6). Finally, the system can interact with third party APIs and datasets through the API gateway.

Important note: the API gateway, the Authorization server and the Discovery Service are supposed to run on replicated machines, since they represent a single point of failure and can cause many problems to the overall service if they fail.

Moreover, all the application modules (especially the microservices) are expected to be stateless, according to the REST standard definitions (more details in section 2.6).

All the components will be described in depth in the following sections.

2.2 Component view

In order to reduce the complexity of the **solution domain** (cfr. [1], §6.3.2), the required functionalities of the software are here decomposed by solution domain classes. A domain consists of multiple sub-domains: replaceable parts of the system with well-defined interfaces that encapsulates the state and behavior of their contained classes. Each sub-domain corresponds to a different part of the business. By decomposing the system into relatively independent subsystems, concurrent teams can work on individual subsystems with minimal communication overhead.

Some business functionalities require the aggregation of microservices functionalities (i.e. for processing complex queries about farmer's information). As described in section 2.6, the system should make use of the CQRS pattern to fulfill the above mentioned requirement. Here we provide the internal structure that the interested microservice components should implement to achieve this task:

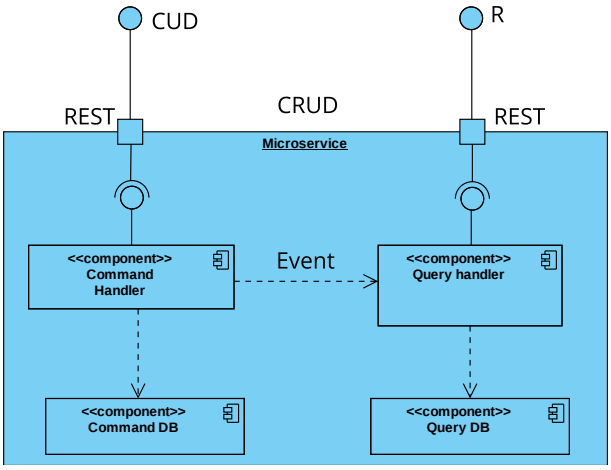

Figure 2: CQRS-based microservice

In a CQRS-based service (ref.[5], §7.2.2), the command-side domain module handles CRUD operations and is mapped to its own database. It handles simple queries, such as non join, primary key-based queries. The command side publishes domain events whenever its data changes. These events might be published using event sourcing.

A separate query module handles simple queries. It's much simpler than the command side because it's not responsible for implementing the business rules. The query side uses whatever kind of database

makes sense for the queries that it must support. The query side has event handlers that subscribe to domain events and update the database.

The overall component diagram is shown below:

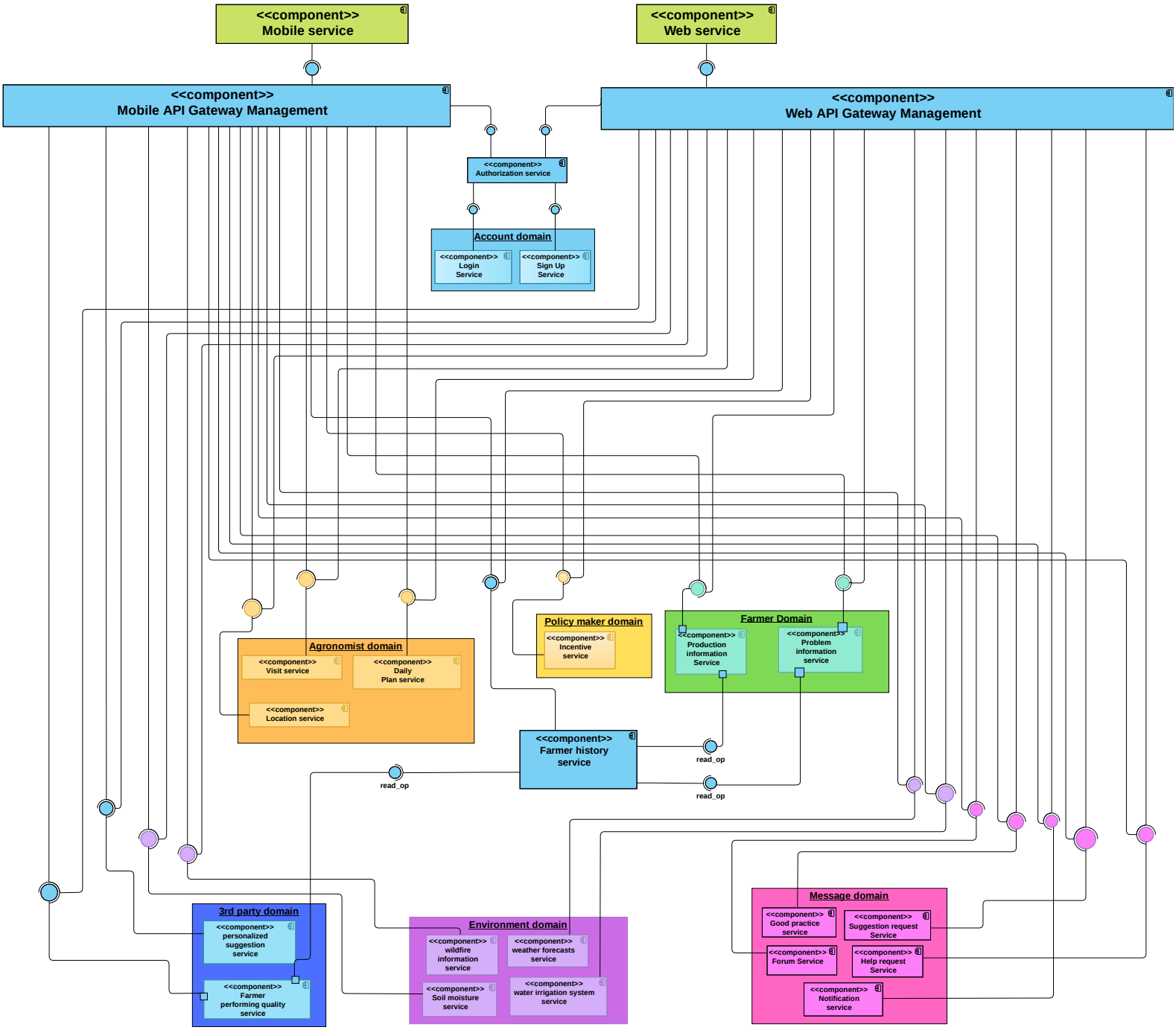

Figure 3: Component diagram

In figure 3 the platform is decomposed in 7 main domains:

- The ACCOUNT domain contains all the features in order to manage the user side. In fact, it includes the log in and sign up microservices. Moreover, the component provides different interfaces for different users, in order to provide additional distinct information.
- The MESSAGE domain handles all different kind of message information exchanged between

users, from simple suggestion and help requests to forum and good practice document requests. These microservices are completely used by farmers, and partially used by both agronomists and policy makers.

- The ENVIRONMENT domain contains microservices that handle all the available environment information gathered by sensors along the state of Telangana. The four microservices contained in this domain manage the information of the water irrigation system, of the soil moisture, of the weather forecasts and the wildfire ones.
- The 3RD PARTY domain provides statistics mainly based on farmers information and the integration with the environment service information. This module is composed by two microservices responsible to provide reliable farmer's performance quality information and personalized suggestion.
- The FARMER domain contains farmer related CQRS-based microservices providing CRUD operations for production and problem information.
- The AGRONOMIST domain contains those microservices that fulfill agronomist requirements. In particular they provide utilities for managing the area they are responsible of, the daily plan and the visits confirmation.
- The POLICY MAKER domain is composed by a single main microservice that is responsible to provide the incentive assignment functionalities.
- Finally, the Farmer history microservice provides READ only operations. This component uses event handlers that subscribe to events published by Farmer and 3rd party domains' services in order to keep its database updated.

2.3 Deployment view

In this section we describe the environment in which the system is executed from the perspective of the deployment diagram. Figure 4 shows the structure of the hardware components that execute the software, and presents the architecture in a more detailed way.

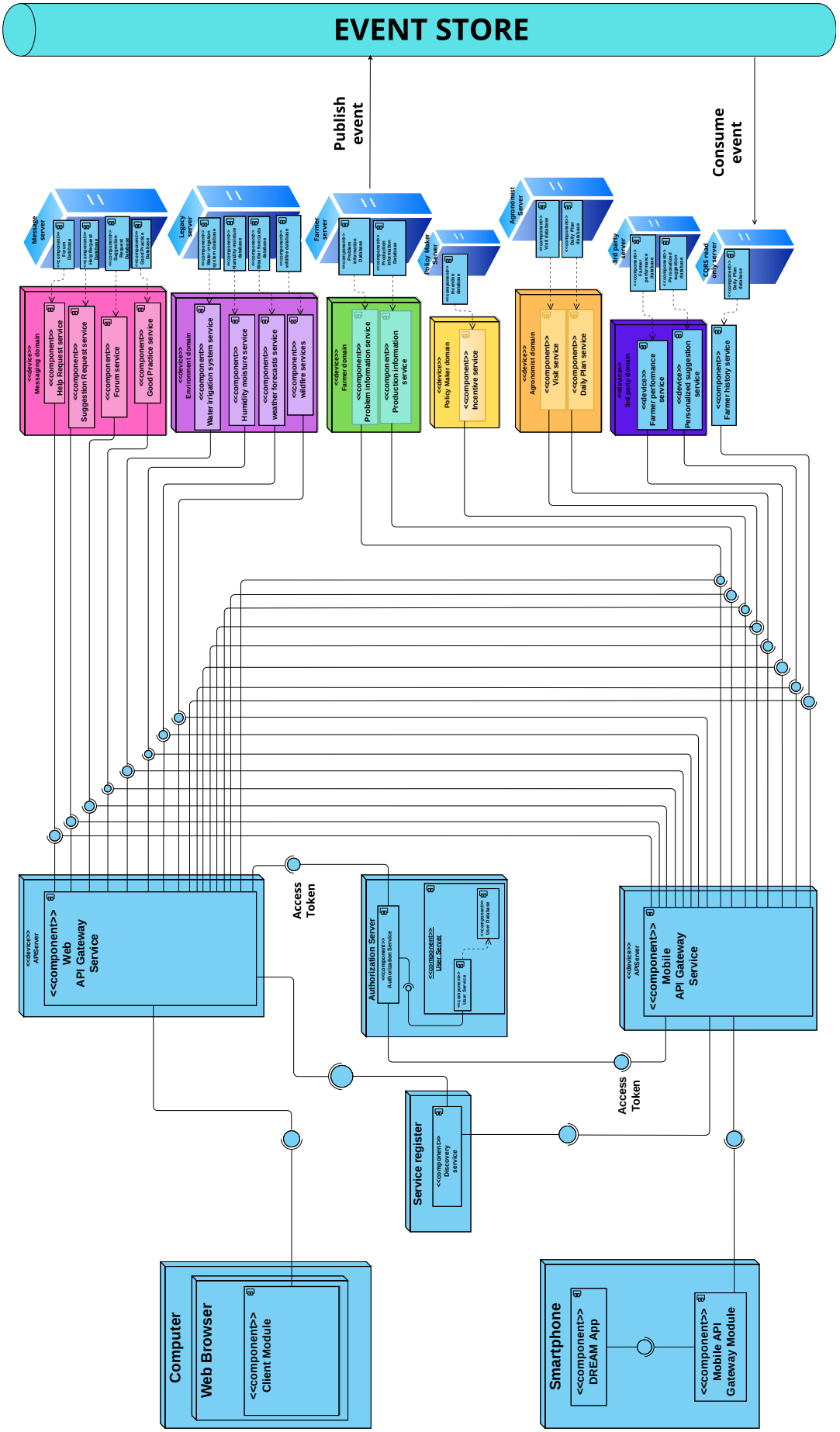

Figure 4: Deployment diagram diagram

The above diagram highlights the four tiers of the architecture, and the relationship between components. In particular:

- The PRESENTATION TIER is the front layer of the architecture and provides the final user interface (more details in section 3). As described in the RASD, the platform will be achievable through either an online web application or an installable mobile application. The web page's UI is supposed to provide both mobile and desktop visualization modes. This layer communicates with the application tier through API calls on HTTPS.

- The ROUTING TIER is composed by the API gateway pattern, which acts as the entry point of the application from the outer world. It is responsible mainly for request routing and protocol translation. It is similar to the Facade pattern from object-oriented design Like a facade, an API gateway encapsulates the application's internal architecture and provides an API to its clients [5]. In addition to these functionalities, it is common practice to introduce in the API gateway pattern edge functionalities such as: authentication (Verifying the identity of the client making the request) and authorization (Verifying that the client is authorized to perform that particular operation) services, made possible through the introduction in the architecture of the authorization server component. Finally, in order to provide the service discovery pattern, a service registry component is required.

- The BUSINESS TIER contains the cloud of microservices that runs the core functionalities of the S2B. These ones are the same described in section 2.2 and communicate with the routing layer through RESTful APIs. Each microservice is responsible of a portion of the business logic and refer to its own database. Furthermore, microservices contained in the same domain (defined in the decomposition presented in section 2.2) can be stored in the same server.

- Finally the DATA TIER contains the DBMS servers that store the data and provide SQL queries from the related microservices that communicate through an *event store* for the implementation of the database management patterns (more details in section 2.6). This final tier could be implemented according to the **single service per container** or the **serverless** patterns.

2.4 Runtime view

In order to make simpler diagrams, we decided to show only the main flow of the events, without considering all the possible alternatives and exceptions (every actor inserts the data correctly and satisfies their authorization requirements).
Additionally, we also note that the communication between the API gateway and the authorization server to generate the access token (to allow users to access only the resources they have permission for) has been left implicit.

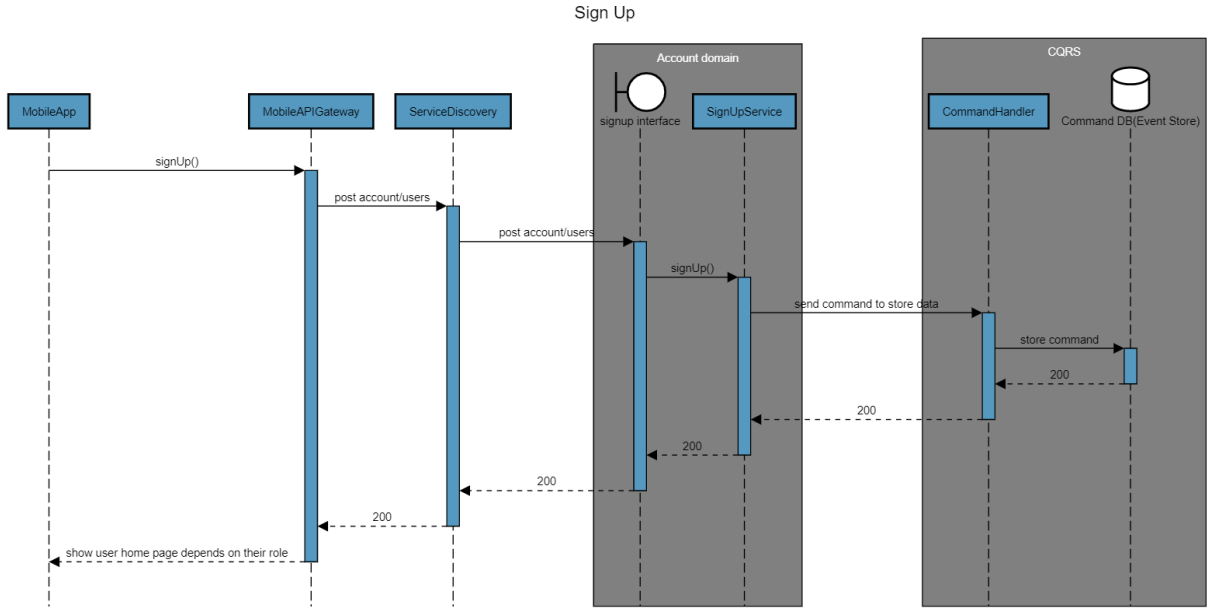

Figure 5: Sign Up

Actor: All type of user
UI flow: 3.1 Sign Up, Login

Firstly, this diagram explains the flow of sign up for all type of user. Due to CQRS pattern, there are two different database according to their purpose (2.2).

FLOW OVERVIEW
Mobile App sends a method ”postUser()” to Mobile API Gateway, then service discovery finds a routing path by URL of REST API. Then SignUpService receives post request. After that, to update the database, it sends command to Command handler to insert the data into Command DB, since it is a write operation.

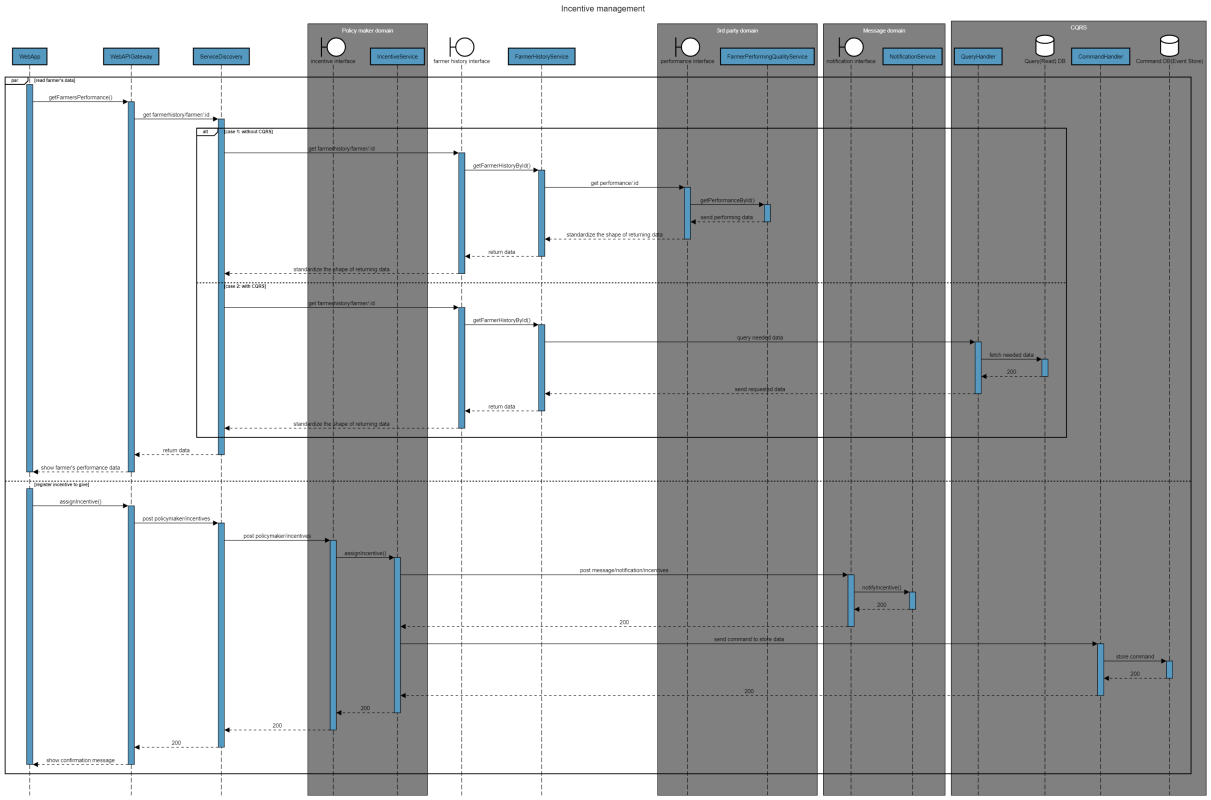

Figure 6: Incentive management

Actor: Policy maker
UI flow: 3.1.1 Farmer's performance data

Secondly, we would like to highlight the reasoning of our design decision about this diagram. In the upper sequence flow we display how the interactions would be without the CQRS pattern, while in the lower flow we show the interactions with the CQRS pattern. For the first part, by the fact we consider the analysis taken cared by 3rd party systems, it is also suitable to have replica of database to fetch aggregated data when it is needed, regardless of condition of their system.

FLOW OVERVIEW

Above part describes the flow of read operation: search and collect the data from the database. It is required to access Query DB. Then, the below part explains the flow of write operation. Incentive service needs to call Notification service to let the selected farmer receive it.

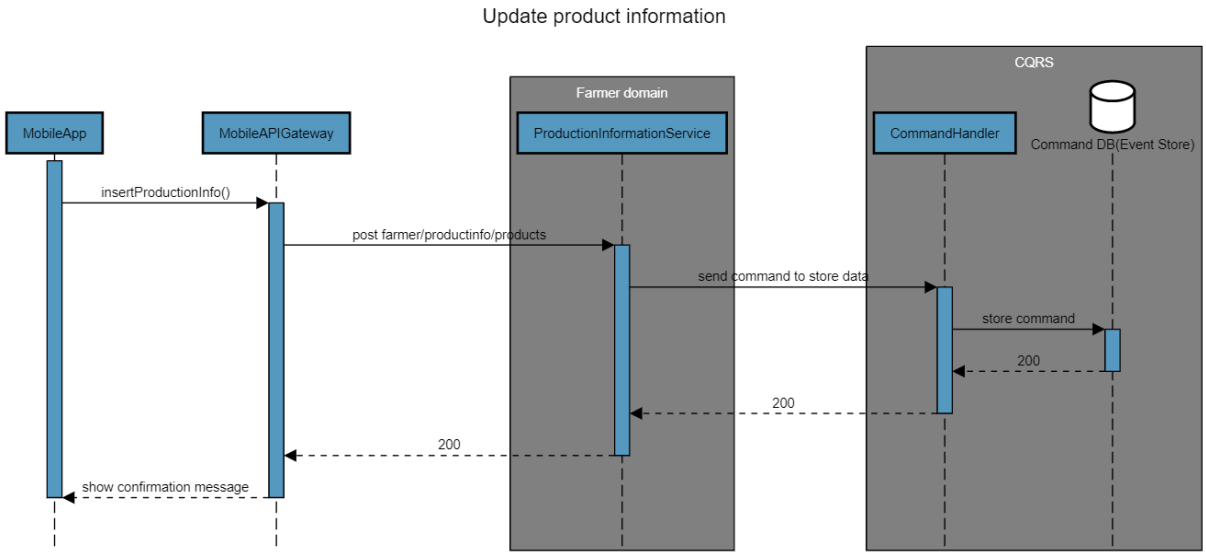

Figure 7: Product information registration

Actor: Farmer
UI flow: 3.1.2 Production data registration, Help/Suggestion request

From now on, we consider that the communication between the API gateway and the discovery service, in order to find the exact service location, is implicit inside the API Gateway instance. Moreover, the interface of each microservice is no longer explicitly put in the diagram, to avoid redundant and less relevant information.

FLOW OVERVIEW

Farmer inserts needed information and sends the production data to register it. API gateway contacts to authorization server to assure the permission; if it is verified, then discovery service finds ProductInformationService and executes the method. Since it is a write operation, it sends data to Command handler to insert it into Command DB.

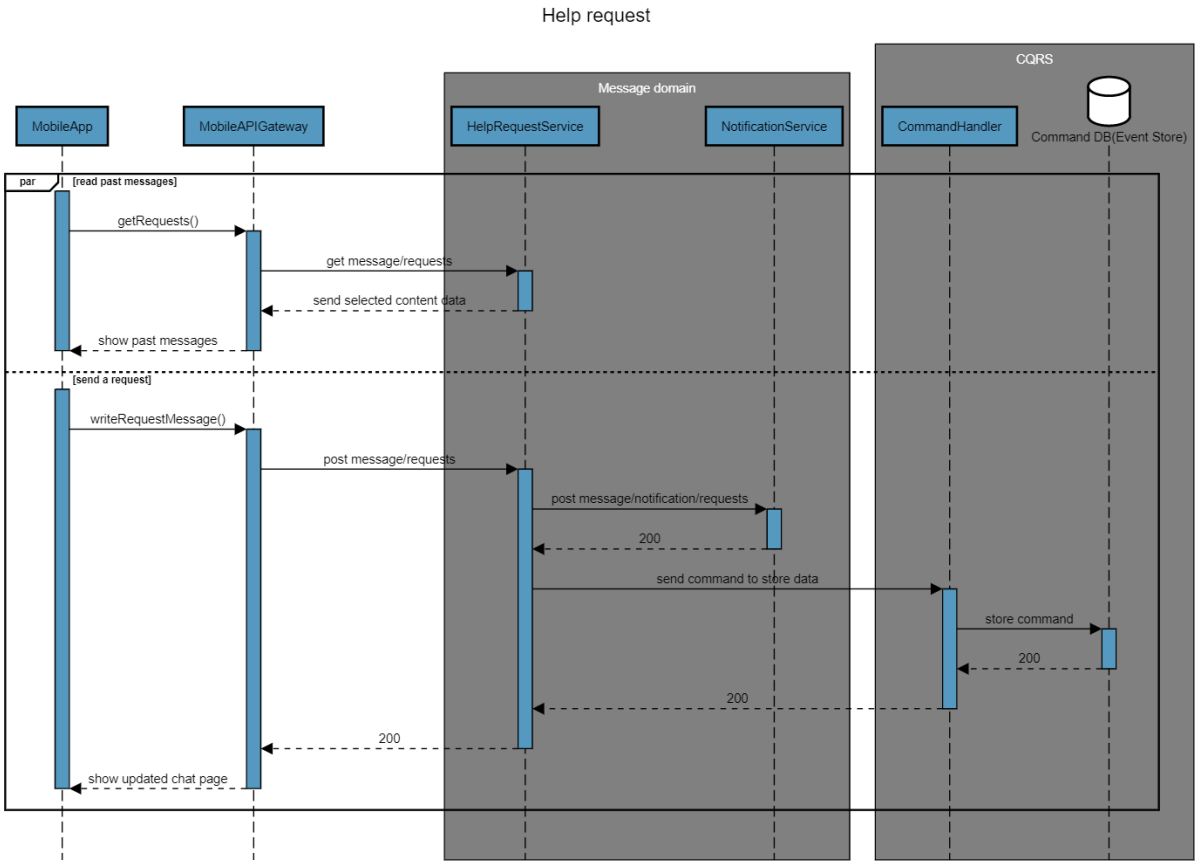

Figure 8: Help request registration

Actor: Farmer
UI flow: 3.1.2 Production data registration, Help/Suggestion request

This diagram explains the flow of asking help request. As a previous step, it requires the farmer to select whom they would like to ask the request, though we assumed this operation has been done in advance and the diagram above describes the phase afterwards.

FLOW OVERVIEW

First half shows the flow of read operation to visualize the past messages; on the other hand, the below part represents the write operation to send the message. For read operation, it does not access to Query DB since the data is not aggregated data, but it simply accesses the database inside of its own service.

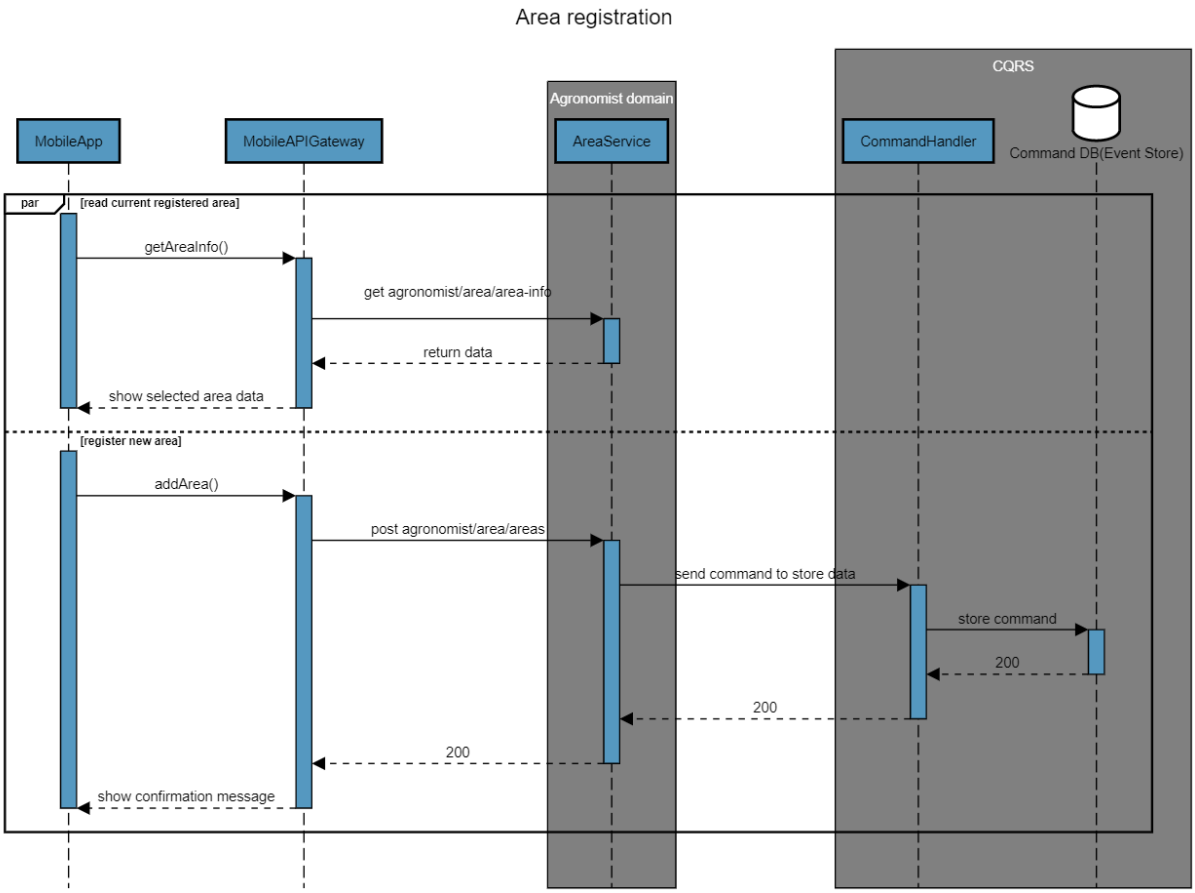

Figure 9: Area registration

Actor: Agronomist
UI flow: 3.1.3 Area management

This diagram shows the flow of registering a new area to supervise. As like other examples, this diagram contains read and write operation individually.

FLOW OVERVIEW

Firstly, by calling getAreas(), the system will fetch current area information about the place (name of farmers, agronomists, ...). After the agronomists have observed area information, if they are interested in, they could register a new area to supervise. Write operation requires updating Command DB to let new registered data be reachable by other services.

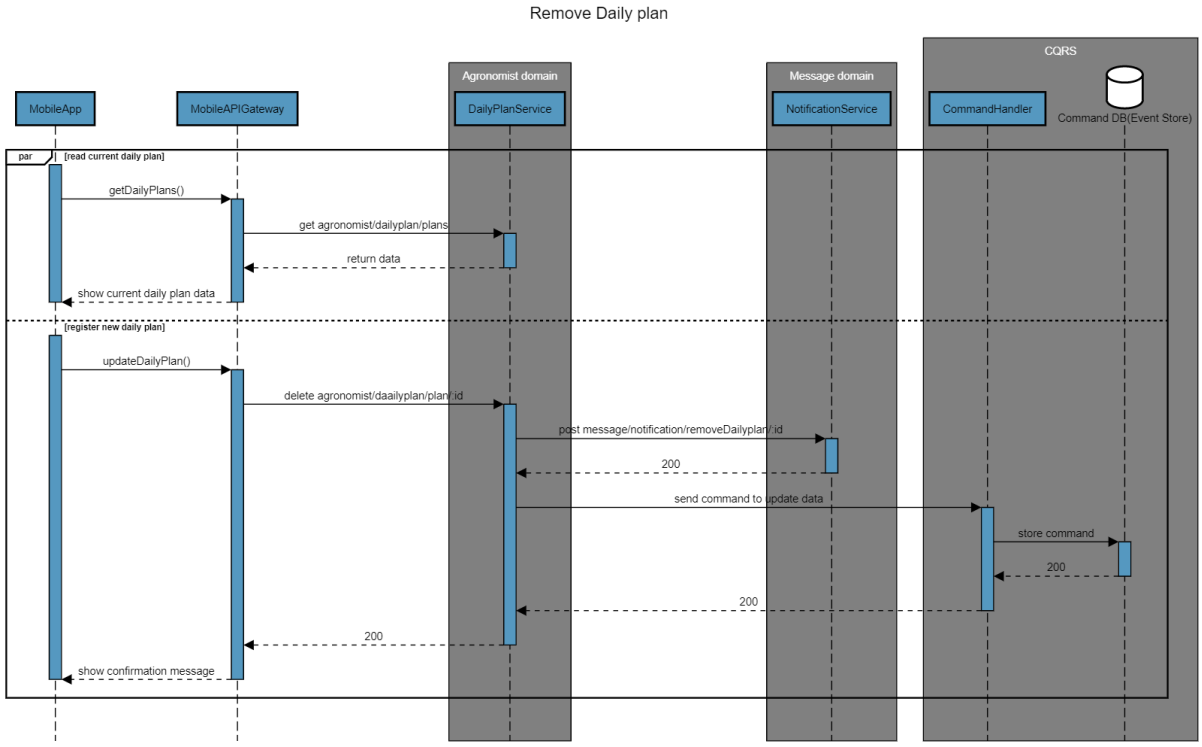

Figure 10: Remove Daily plan

Actor: Agronomist
UI flow: 3.1.3 Daily plan management

As the last example, this diagram shows the flow of updating daily plan of agronomist. Unlike other insertion examples, here it deletes the record from database, so it requires DELETE method.

FLOW OVERVIEW
On the upper part, the system lets agronomists know the daily plan they have. Instead, the bottom part explains the flow of deletion operation. By specifying the ID in the URL, Command Handler searches the record that owns that id and deletes it.

2.5 Component interfaces

In this section we present the main interfaces used in the system. Given the microservice-based approach, the architecture is composed of many simple components, each one with its own interface (as shown in section 2.2) that handles the basic operations on that specific module.

For simplicity, we decided to explicitly list only the external methods, provided by the API gateway, that ensure the satisfaction of the high-level goals of the different actors.

In particular:

- UserInterface: it handles the operations of a generic user, such as login, registration and visualizing and updating account information;
- PolicyMakerInterface: it handles the operations of a policy maker, such as visualizing different types of data and statistics, assigning incentives and asking farmers to write good practices;

- FarmerInterface: it handles the operations of a farmer, such as visualizing relevant information about his area and production, inserting details about production and problems faced, interacting with other actors through requests and forums and writing good practices;
- AgronomistInterface: it handles the operations of an agronomist, such as managing the areas he is responsible of, visualizing different types of information about the areas and the farmers, answering farmers' requests and visualizing, updating and confirming the execution of daily plans.

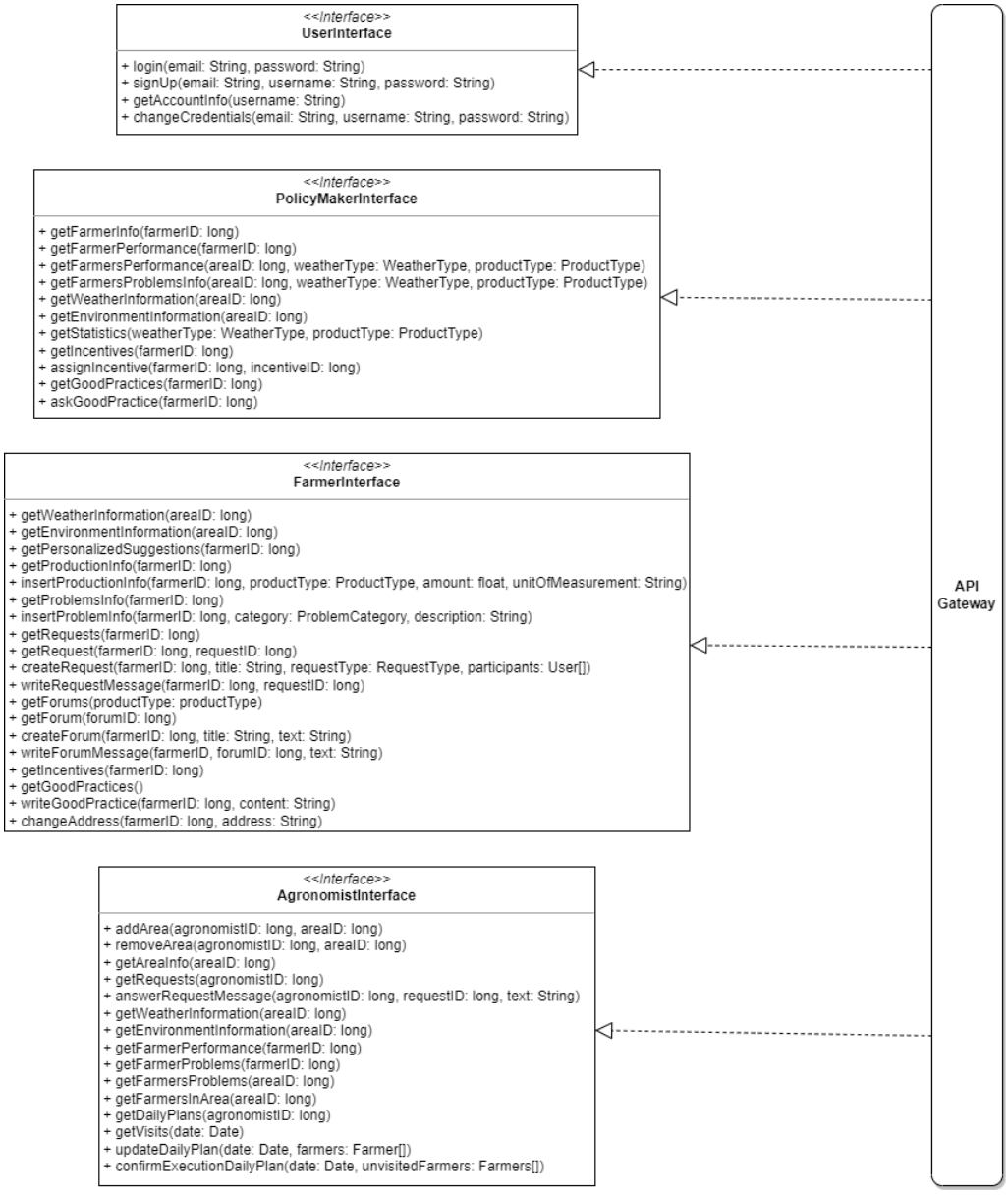

Figure 11: Interfaces Diagram

2.6 Architectural styles and patterns

2.6.1 Microservice Architecture

The S2B will be implemented using Microservice architecture, an architectural style that breaks the business logic into small and independent modules.

Adopting a microservice-based approach grants many benefits:

- modularity: this makes the software easier to understand, develop, test, and become more resilient to architecture erosion; this aspect is very important for complex systems such as DREAM.
- scalability: since microservices are implemented and deployed independently of each other (i.e. they run within independent processes), they can be monitored and scaled independently; scalability is a crucial aspect for DREAM, considering the wide user base and the possibility of future expansions.
- distributed development: development can be parallelized, enabling small autonomous teams to develop, deploy and scale their respective services independently; it also allows the architecture of an individual service to emerge through continuous refactoring; in general, microservice-based architectures facilitate continuous integration, continuous delivery and deployment.
- digital public good fitting: since DREAM tries to follow the Digital Public Good principles (being the product of a United Nations initiative), a microservice architecture enables the system to be well-structured and to adopt open APIs with great flexibility.

2.6.2 RESTful Architecture

The S2B will adopt the REST (Representational state transfer) architecture, which is based on the concept of stateless sessions and connections. Since each application is designed as an independent service, REST is a valuable architectural style for microservices, thanks to its simplicity, flexibility, and scalability.

One of the strongest advantages of REST for microservices is that services can communicate without requiring internal knowledge of one another. Furthermore, in RESTful microservices, APIs are standardized according to the OpenAPI Specification, which provides a documented contract for how services are expected to communicate across ongoing development.

2.6.3 API Gateway

The API gateway pattern is very often used in Microservice architecture to handle the complexity of a distributed and fragmented system. The API gateway represents the single entry point for all clients. It handles requests in one of two ways: some requests are simply proxied/routed to the appropriate service, while other requests are fanned out to multiple services.

Moreover, rather than provide a one-size-fits-all style API, the API gateway can expose a different API for each client (i.e. Web app API gateway, Mobile API gateway, Public API gateway). This pattern allows to simplify the client's logic by hiding the microservice structure internally and by handling multiple service calls server-side.

2.6.4 Event Sourcing

Given the fragmented nature of microservices, a service must atomically update the database and send messages in order to avoid data inconsistencies and bugs. Event sourcing persists the state of a business entity as a sequence of state-changing events. Whenever the state of a business entity changes, a new event is appended to the list of events. The application can reconstruct an entity's current state by replaying the events.

Applications persist events in an event store, which is a database of events. The store has an API for adding and retrieving an entity's events. The event store also behaves like a message broker. It provides an API that enables services to subscribe to events. When a service saves an event in the event store, it is delivered to all interested subscribers.

This particular pattern is required from other patterns used in the overall database management process.

2.6.5 Domain Event

Since a service often needs to publish events when it updates its data, the Domain Event pattern is used to organize the business logic of a service as a collection of DDD (Domain-Driven Design) aggregates that emit domain events when they are created or updated. The service publishes these domain events so that they can be consumed by other services.

This pattern is required from other patterns used in the overall database management process, such as CQRS and Saga.

2.6.6 Database per Service

The Database per Service pattern is used to keep each microservice's persistent data private to that service and accessible only via its API. In this way, the service's database is effectively part of the implementation of that service and it cannot be accessed directly by other services.

Depending on the resources available for the real deployment of the system, there are different ways to keep a service's persistent data private:
- Private-tables-per-service: each service owns a set of tables that must only be accessed by that service
- Schema-per-service: each service has a database schema that's private to that service
- Database-server-per-service: each service has its own database server

It is a good idea to create barriers that enforce this modularity. Without some kind of barrier to enforce encapsulation, developers may be tempted to bypass a service's API and access its data directly.

In this document we assumed an hybrid approach, as shown in section 2.3: different "domains" have different machines, but inside each machine a logical separation is used.

2.6.7 Saga

This pattern implements each business transaction that spans multiple services as a saga. A saga is a sequence of local transactions. Each local transaction updates the database and publishes a message or event to trigger the next local transaction in the saga.

If a local transaction fails because it violates a business rule, then the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions.

To coordinate these sagas, a choreography approach may be chosen: each local transaction publishes domain events that trigger local transactions in other services.

The Saga pattern enables the application to maintain data consistency across multiple services without using distributed transactions.

2.6.8 CQRS (Command Query Responsibility Segregation)

Given the fragmented nature of microservices, it is no longer straightforward to implement queries that join data from multiple services.

The Command Query Responsibility Segregation (CQRS) pattern defines a view database, which is a read-only replica that is designed to support that query. The application keeps the replica up-to-date by subscribing to Domain events published by the service that owns the data.

This pattern works together with other patterns in the overall database management process.

2.6.9 Access Token

The Access Token pattern is used for security reasons, in order to communicate the identity of the requestor to the services that handle the request. The API Gateway authenticates the request through

an Authorization server and passes an access token (e.g. JSON Web Token) that securely identifies the requestor in each request to the services.

A service can include the access token in requests it makes to other services. In this way, the identity of the requestor is securely passed around the system and the services can verify that the requestor is authorized to perform a specific operation, especially when accessing data resources.

2.6.10 Circuit Breaker

The Circuit Breaker is used to efficiently handle synchronous calls to microservices in order to prevent a network or service failure from cascading to other services. A service client invokes a remote service via a proxy. When the number of consecutive failures crosses a threshold, the circuit breaker trips, and for the duration of a timeout period all attempts to invoke the remote service will fail immediately. After the timeout expires, the circuit breaker allows a limited number of test requests to pass through. If those requests succeed, the circuit breaker resumes normal operation. Otherwise, if there is a failure the timeout period begins again.

This is due to the fact that, when one service synchronously invokes another, there is always the possibility that the other service is unavailable or is exhibiting such high latency it is essentially unusable, causing the caller to waste resources in the meanwhile.

2.6.11 Server-side Discovery

The Server-side Discovery pattern is used to manage the virtualized or containerized environment in which the microservice-based application runs, where the number of instances of a service and their locations changes dynamically.

With this pattern, when making a request to a service, the caller makes a request via a router (a.k.a load balancer) that runs at a well known location. The router queries a service registry, which might be built into the router, and forwards the request to an available service instance.

This pattern allows to keep the client code simpler, since it does not have to deal with discovery. In addition, the possibility of using virtualization inside the system's modules increases the availability and the scalability of the software since it is possible to dynamically generate new service instances when needed.

2.6.12 Monolithic front-end

For the first implementation and deployment of the system, we decided to adopt a monolithic front-end that handles the calls to the system and puts together all the information gathered by querying the platform, thanks to its simplicity and easy deployment. A future implementation of micro front-ends is suggested in order to guarantee scalability also on the client side, increasing however the complexity of the software, but assuring easier updates for long-term operations. For this particular transition, the user experience could be retrieved through some sort of feedback mechanism in order to develop the most suitable application possible.

2.7 Other design decisions

2.7.1 Scale-out

This method consists of cloning the nodes in which we expect to have a bottleneck, in order to increase the overall availability and scalability of the system. In particular, elements such as the API gateway, the Authorization Server, the Discovery Service and the CQRS database should necessarily be replicated since they represent a single point of failure.

Also the microservices can be replicated: the managing process of redirecting the incoming requests to the node with the lowest workload could be done in the load balancer already implemented in the Discovery Service (see 2.6.11 for more details).

2.7.2 Thin and thick client

The web application will be the thin client. This means that it will have only presentation duties, with no business logic inside. In this way, the machines that run the web application are not required to have high computational power. However, a stable internet connection is required to guarantee the proper usage of the application.

The mobile application, instead, will be the thick client. In fact, it is reasonable to save useful information on the local memory of the device (for example chats) in order to avoid continuous requests to the platform (less computational load) and to guarantee also some functionalities when the Internet connection is not available.

2.7.3 Client-side UI composition (for the future)

As mentioned in the Monolithic front-end paragraph (section 2.6.12), we suggest a future transition to micro front-ends, which can be achieved by the Client-side UI composition pattern. This pattern will be used to implement a UI screen or page that displays data from multiple services.

Different teams will develop a client-side UI component (i.e. AngularJS directive) that implements the region of the page/screen for their service. A UI team will be responsible for implementing the page skeletons that build pages/screens by composing multiple, service-specific UI components.

2.7.4 External APIs and Datasets

The system will interact with external interfaces and services, in particular with:

- Identity providers (such as Google, Facebook, etc.) to simplify the process of user registration
- open datasets that collect information from the territory about weather forecasts, soil moisture, water irrigation, humidity, wildfires, etc. through sensors and satellite technologies
- evaluating algorithms to understand the performance of farmers (positive or negative deviance)
- incentives companies to manage the definition and collection of the vouchers that are assigned to farmers
- weather analysis algorithms to categorize the different weather types and characteristics of the areas, in order to support more specific personalized suggestions and statistics

3 User interface design

3.1 User mobile interface

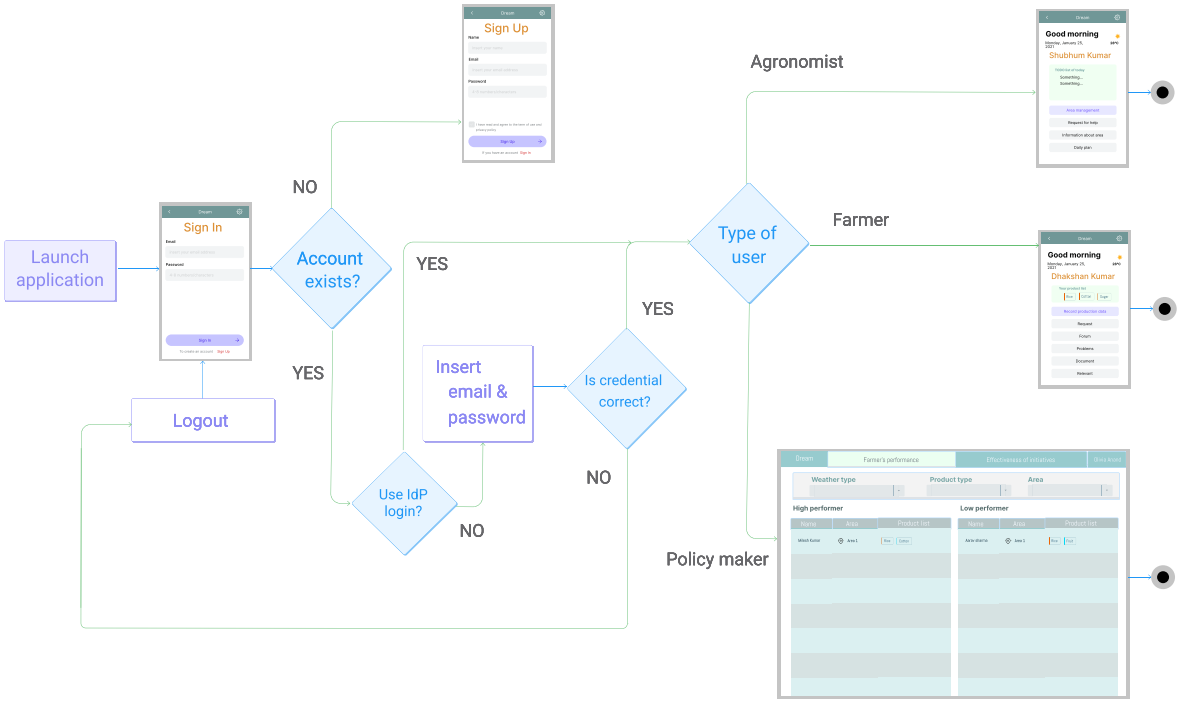

Figure 12: Sign Up, Login

3.1.1 Policy maker web interface

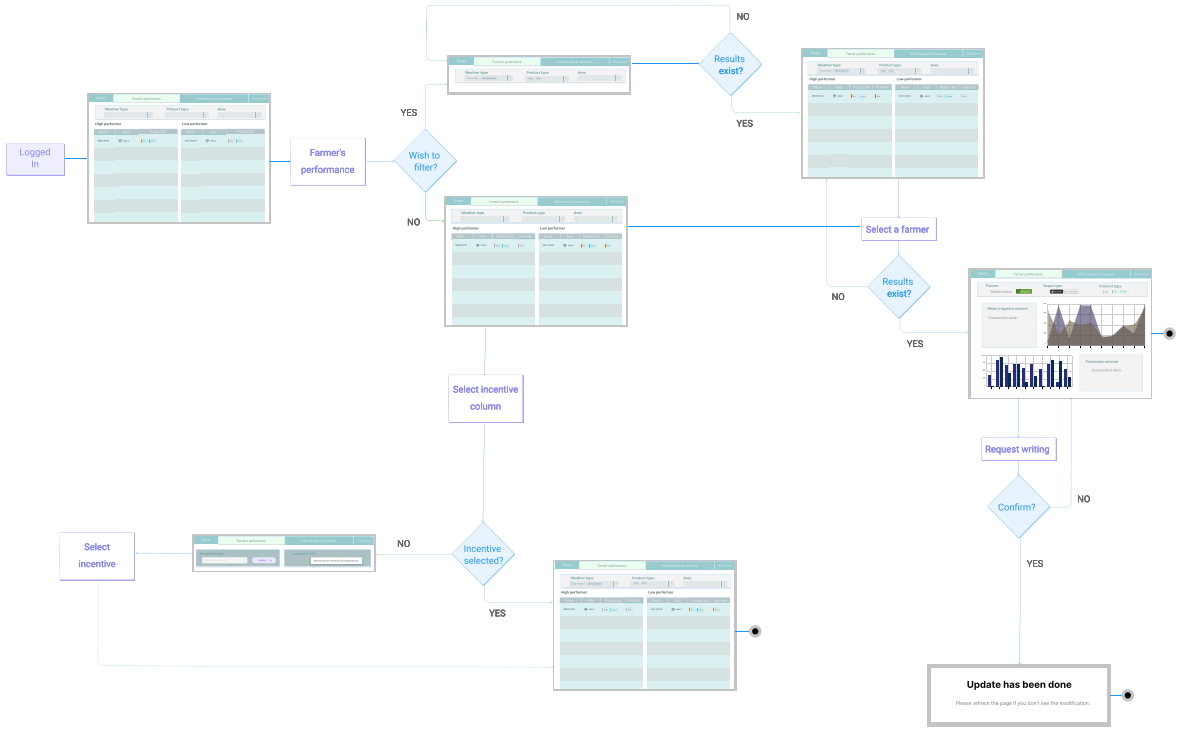

Figure 13: Farmer's performance data

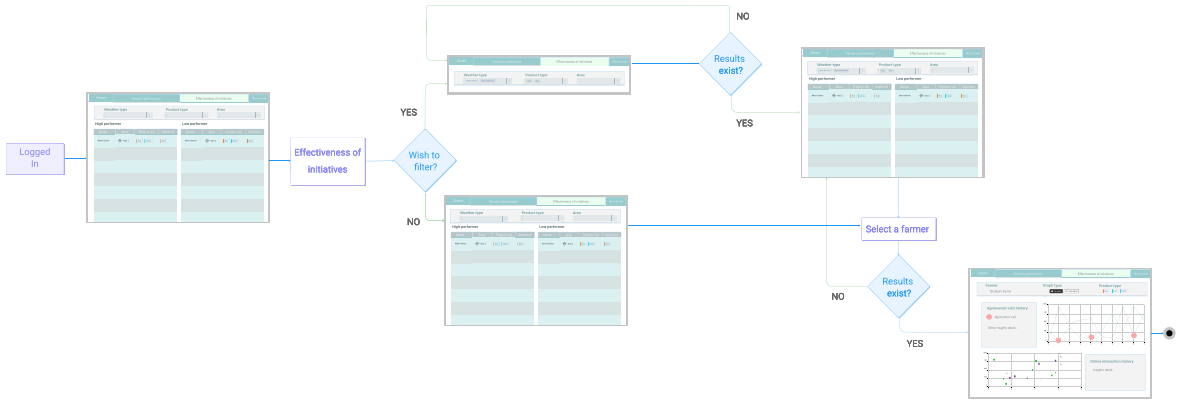

Figure 14: Effectiveness of initiatives

3.1.2 Farmer mobile interface

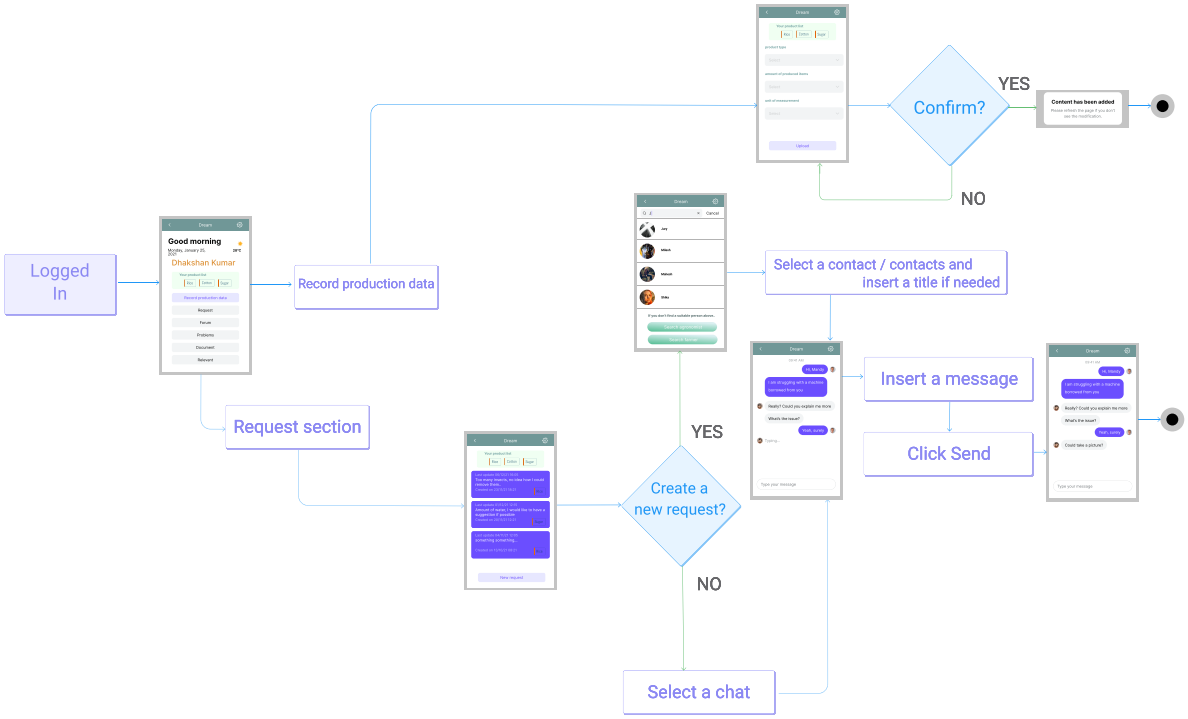

Figure 15: Production data registration, Help/Suggestion request

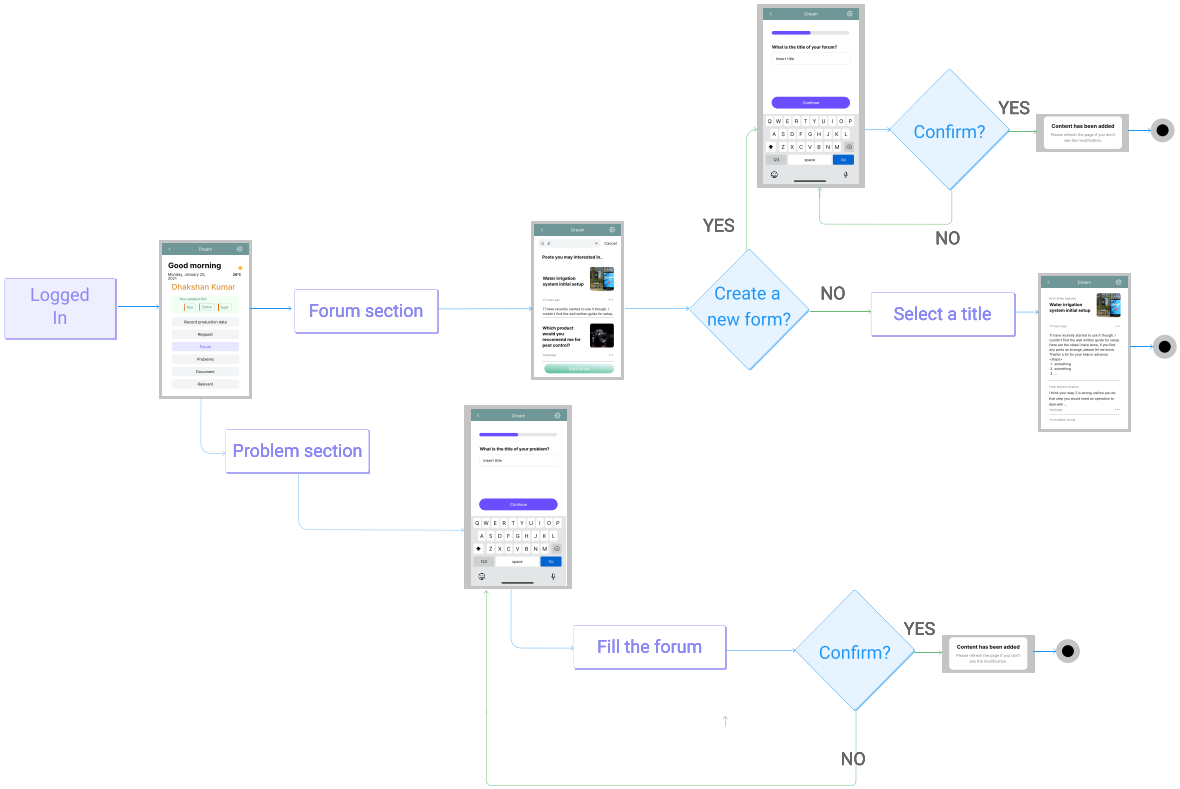

Figure 16: Forum, Problem information

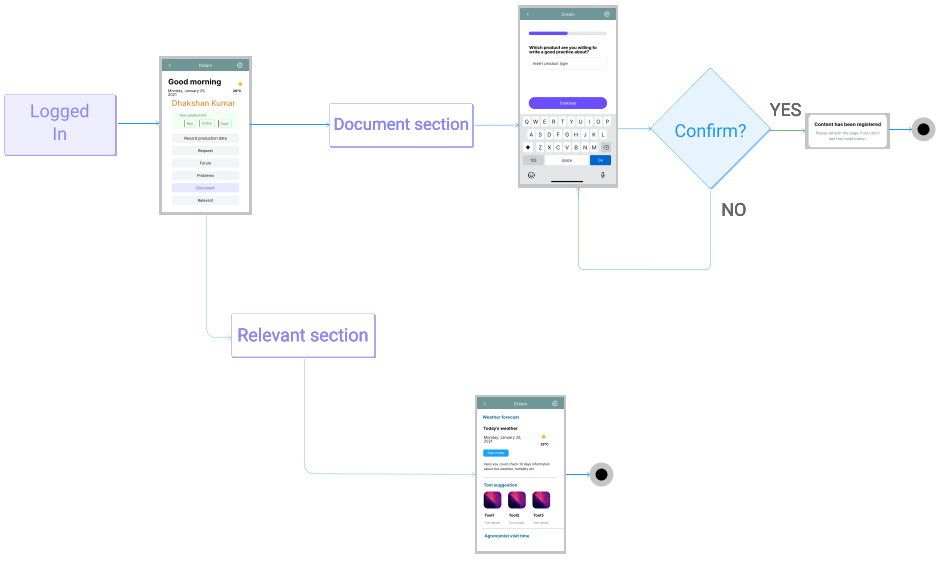

Figure 17: Good practice, relevant data

3.1.3 Agronomist mobile interface

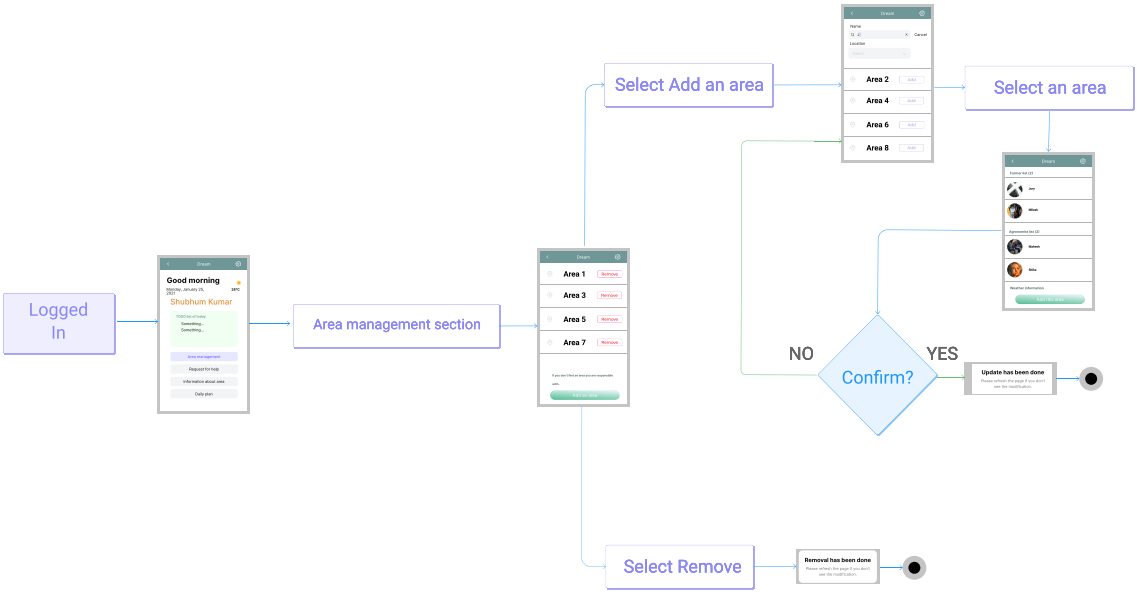

Figure 18: Area management

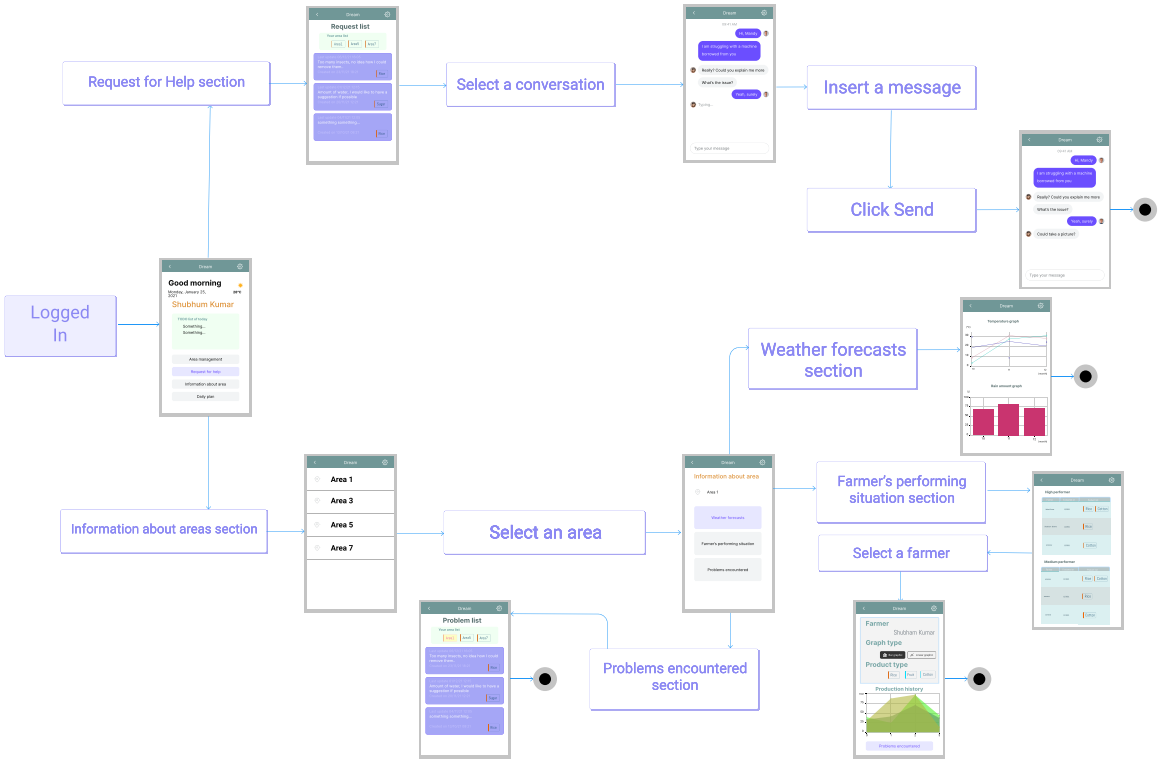

Figure 19: Answer to request, data of area

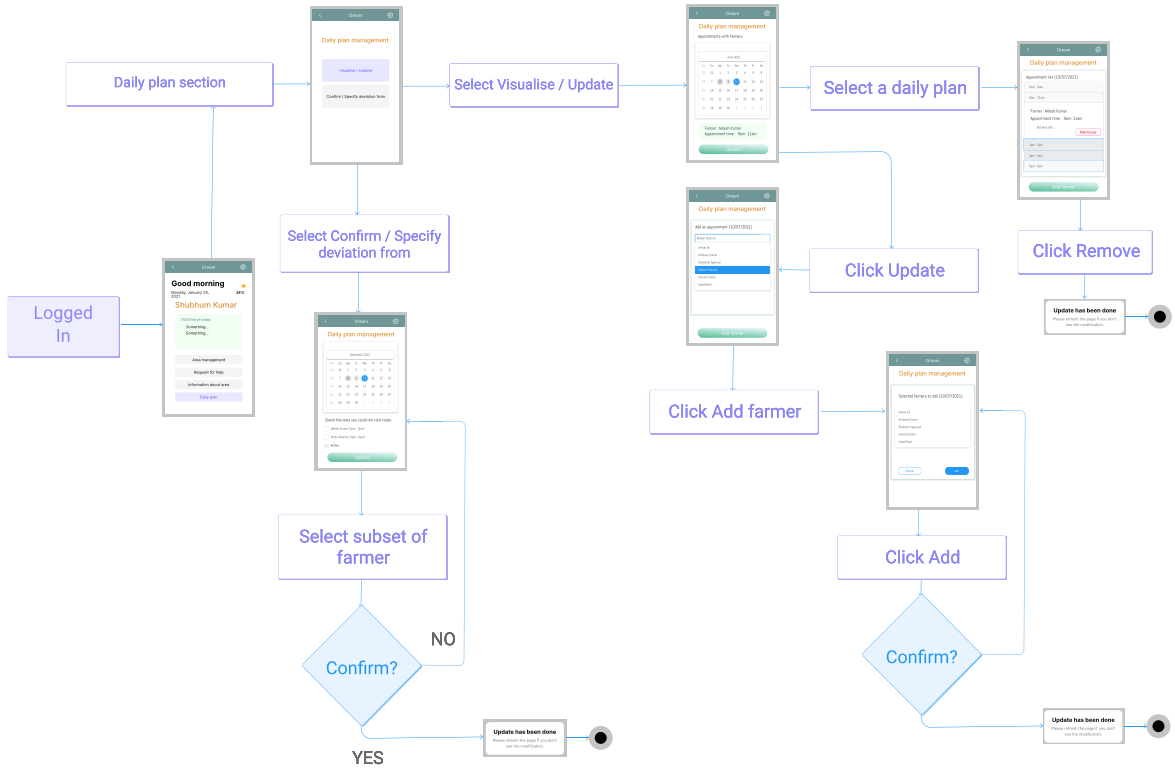

Figure 20: Daily plan management

4 Requirements traceability

This section provides a visual representation of the relationship between microservices (defined in section 2.2) and requirements presented in the RASD document (§3.2) (please refer always to the latest version). In particular, the traceability matrix in table 5 maps each requirement to the set of microservices that are involved in its satisfaction.

MX	Microservice
M1	Login
M2	Sign up
M3	Daily plan
M4	Help request
M5	Suggestion request
M6	Forum
M7	Good practice
M8	Problem information
M9	Production information
M10	Soil moisture
M11	Water irrigation system
M12	Wildfire
M13	Farmer performance
M14	Personalized suggestion
M15	Weather forecasts
M16	Incentive
M17	Visit
M18	Location
M19	Notification
M20	Farmer history

Table 4: Microservices table

	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	M11	M12	M13	M14	M15	M16	M17	M18	M19	M20
R1	✓	✓																		
R2													✓							
R3																				✓
R4															✓					
R5																✓				
R6																✓				
R7									✓											
R8																	✓			
R9							✓													
R10							✓												✓	
R11				✓	✓	✓	✓	✓	✓											
R12									✓											
R13									✓											
R14									✓											
R15								✓												
R16								✓												
R17								✓												
R18																		✓		
R19				✓	✓														✓	

	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	M11	M12	M13	M14	M15	M16	M17	M18	M19	M20
R20				✓	✓															
R21	✓	✓																		
R22	✓	✓																		
R23																	✓			
R24						✓														
R25						✓														
R26						✓													✓	
R27				✓	✓														✓	
R28				✓	✓															
R29															✓			✓		
R30													✓							
R31								✓												
R32			✓																	
R33			✓																	
R34																	✓			
R35			✓																	
R36																		✓		
R37																		✓		
R38			✓														✓			
R39										✓	✓	✓		✓						

Table 5: Requirements traceability matrix

In addition, also the software system attributes (described in the RASD) are guaranteed by the architectural design choices made and explained in this document.

In particular:

- **Usability**: it is achieved through a simple and intuitive interface, as shown in section 3. The possible choices are clearly presented to the user, both in the mobile app and in the web application. With this simple type of interface, the application could be used by all types of users, for example also by elders and by people that are not very experienced with electronic devices.

- **Reliability and Availability**: they are accomplished through replication and the subdivision in microservices. Thanks to modularity and low coupling, possible problems are limited to a single microservice, without affecting others. Moreover, the replication of nodes allows the system to be even more fault tolerant, especially in critical elements such as the API gateway, the Authorization server, the Discovery Service and the shared read-only database.

- **Security**: it is guaranteed through the Authorization Server and the Access Token pattern. Since the API gateway represents a single point of entrance, the identity of a user can be checked before forwarding his request to the microservices. Moreover, since the access token is passed around the system together with the request, a Role Based Access Protocol (RBAC) could be adopted because the information about the role of a user can be put inside the token. In addition, communication running on HTTP is encrypted thanks to protocols such as HTTPS and SSL.

- **Portability**: it is achieved through the implementation of a web application and the existence of an API gateway. The lightweight web application allows the system to be accessed from any device provided with an internet browser. The API gateway, instead, allows the development of a mobile app for different operating systems (mainly Android and iOS) thanks to the fact that it offers a dedicated interface for each client type, guaranteeing compatibility and providing additional elasticity to the developers.

- **Maintainability and Scalability**: they are accomplished through the Microservice architecture itself. A microservice-based approach grants high modularity and low coupling, with independent components that are easier to maintain, test and fix. Microservices also grant high scalability because new modules can constantly be added to the system for future upgrades, introducing new functionalities, without affecting the behaviour of the other components of the application.

5 Implementation, integration and testing plan

5.1 Implementation overview

In this section we present all the preliminary considerations needed to implement and test the S2B from the beginning of its life.

The S2B will be divided as follows:

- Client (Web App and Mobile App)
- API gateway (with Discovery Service)
- Authorization Server
- Microservices
- Event BUS
- External services

A bottom-up approach is suggested to implement these elements, in order to avoid stub structures that would be more difficult to implement and test. An incremental integration facilitates bug tracking and generates working software quickly during the software life cycle.

Given the Microservice Architecture, each microservice can (and should) be developed independently, starting from the ones that are more critical.

Unit testing should be done before integrating a component with the rest of the system, in order to ensure that the offered functionalities work correctly in an isolated environment. After the integration, integration tests should follow to guarantee the correct behavior of the system at each step. These tests should not only check the interactions of the new component with the old elements, but should also guarantee that previous checked behaviours between old components have not been changed or compromised.

5.2 Integration plan

In this section, we describe how the components are integrated in order to build the S2B. The integration is divided into different phases.

First of all, the first components to build are the microservices, since they are the core of the system. It is possible to prioritize the integration of the most critical services at first, and later on add the other services.

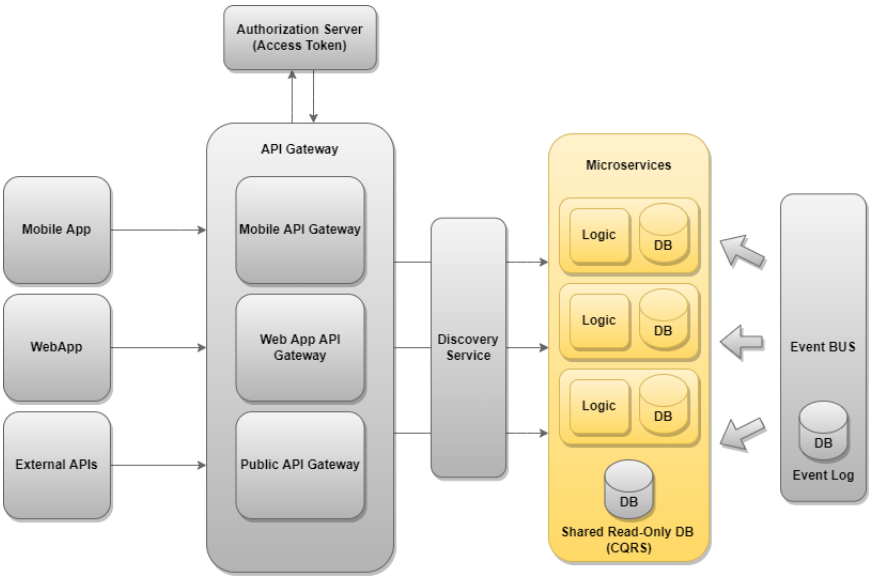

Figure 21: Integration - Step 1

After this, the Event BUS is the next thing to add, together with the Shared Read-Only DB. The Event BUS will implement all the design patterns related to database communication between microservices, in order to guarantee a correct transmission of information. The Shared Read-Only DB, crucial in the CQRS pattern, will be kept up-to-date by every single microservice.

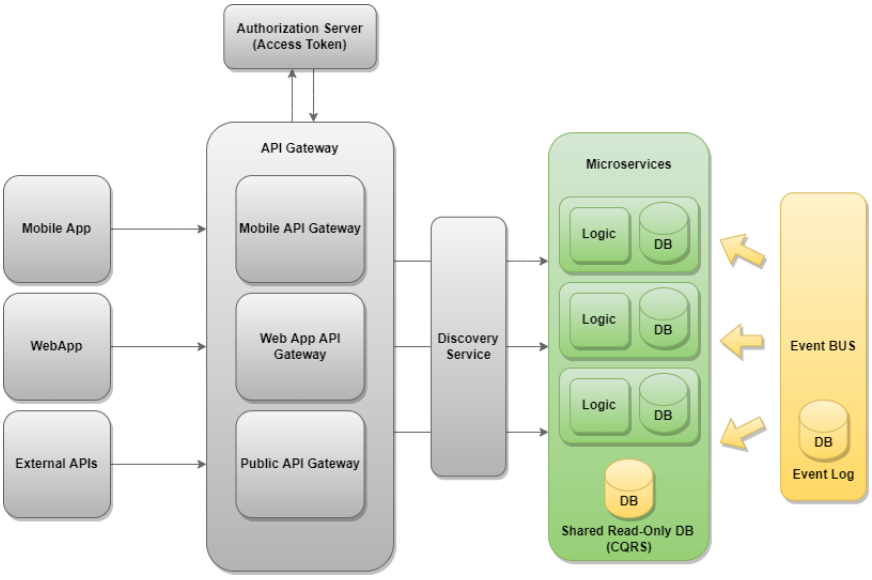

Figure 22: Integration - Step 2

Now comes the API gateway, that will assure the correct invocation of the methods and functionalities

of the microservices, together with the Discovery Service, used for routing the requests to the correct microservice instance and for load balancing.

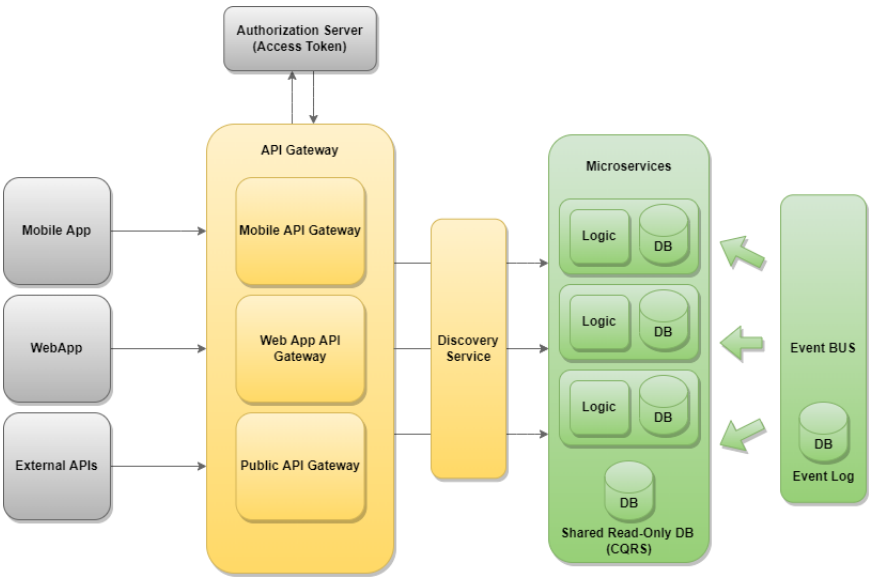

Figure 23: Integration - Step 3

At this point, it is possible to integrate the authorization server, connected to the API gateway, that will handle all the security aspects, especially those related to authentication and authorization.

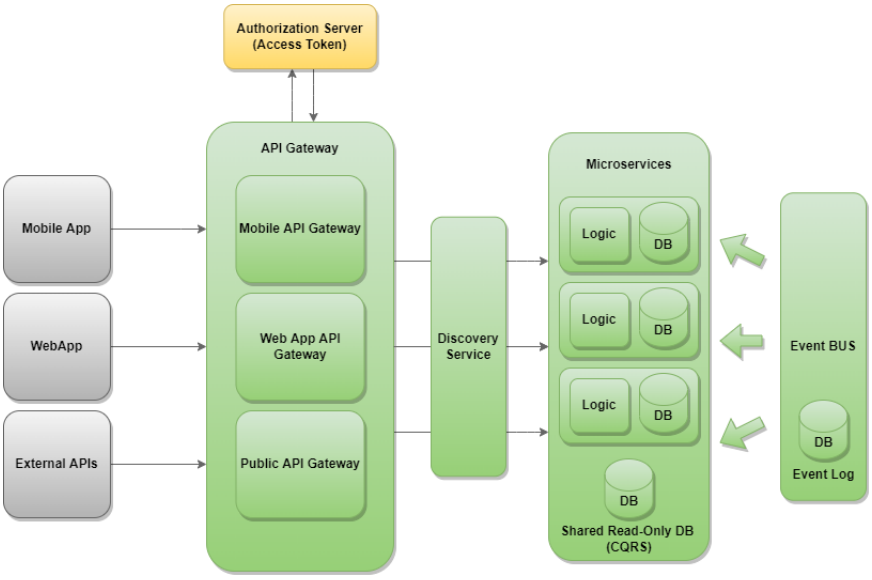

Figure 24: Integration - Step 4

Then, it is time to integrate the external APIs and Datasets. Since we are assuming them as given and already implemented, we just need to correctly connect them with the rest of the system.

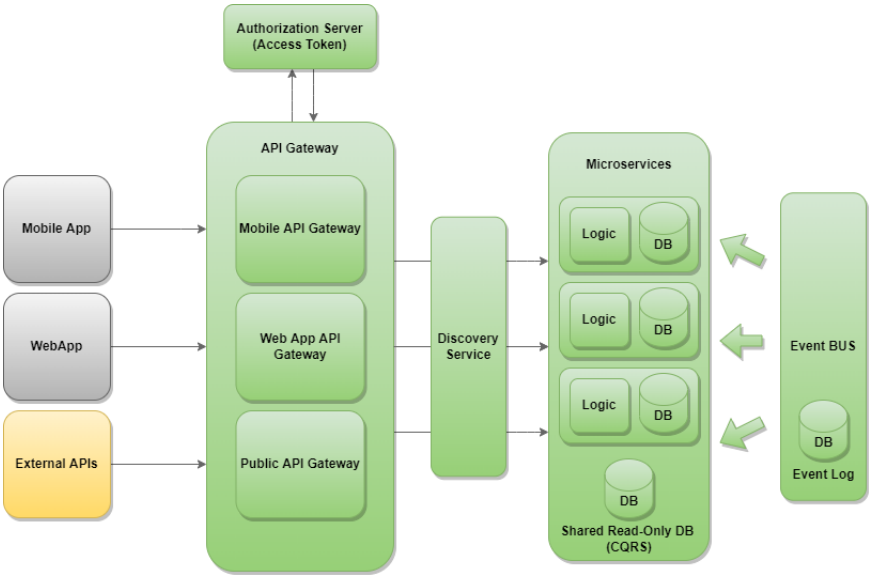

Figure 25: Integration - Step 5

Finally, it is possible to integrate the mobile application module and the web application module, together with the web browser, that will handle the user-side aspects of the S2B.

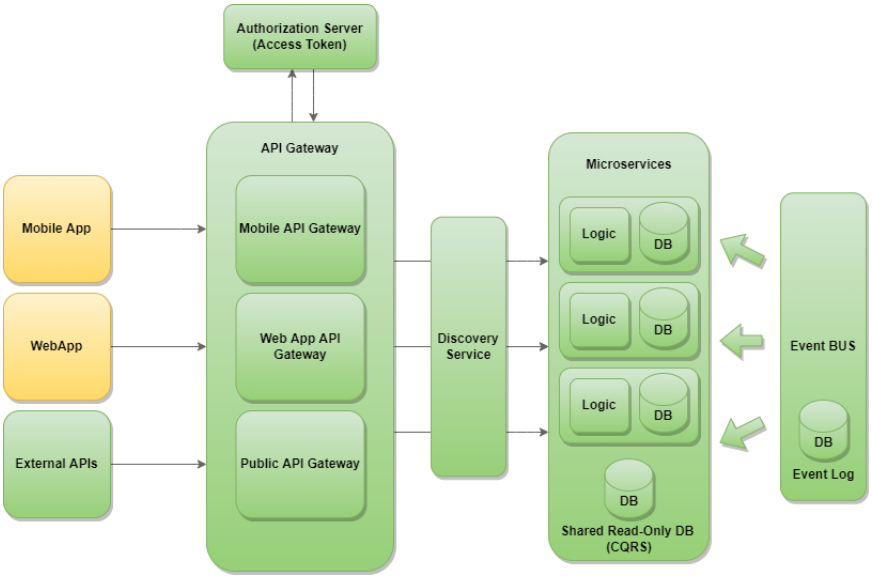

Figure 26: Integration - Step 6

5.3 Test plan

In Microservice Architecture, testing is a critical aspect of software development since the fragmented structure increases the overall complexity of the system. To have a better understanding of the problem, we can refer to Mike Cohn's test pyramid (Fig. 27): as we move towards the top layers of the pyramid, the scope of the tests increases and the number of tests that must be written decreases.

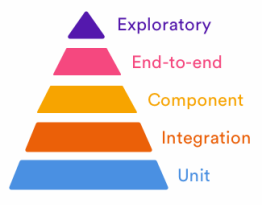

Figure 27: Mike Cohn's test pyramid

Following this structure, we can identify different testing types:

- unit testing
- integration testing
- component testing
- end-to-end testing
- performance testing

Unit testing is very important in this context to validate each business logic (aka microservice) separately, since it provides the coverage of each module in the system in isolation. It is better to keep the testing units as small as possible: it becomes easier to express the behavior if the test is small since the branching complexity of the unit is lower. Also, Unit testing is a powerful design tool when combined with Test-Driven Development (TDD).

Integration testing verifies the communication path and interactions between the components to detect interface defects. Integration test collects microservices together to verify that they collaborate as intended to achieve some larger piece of business logic. One of the most important aspects of service-to-service testing is tracing, especially when multiple services are touched by a single test. Therefore, it becomes imperative to have observability and monitoring of requests across services.

Component testing is used to test single microservices in isolation, once we have executed unit tests of all the functions within microservices. Component tests should be implemented within each microservice's code repository. By writing the test at the granularity of the microservices layer, the API behavior is driven through the test from the consumer perspective. At the same time, the component tests will test the interaction of microservices with the database, all as one unit.

End-to-end testing treats the system as a black box since its intention is to verify that the system as a whole meets business goals, irrespective of the component architecture in use. It provides coverage of the gaps between the services and, additionally, allows testing the correctness of message passing between them. End-to-end testing is crucial to verify that the system is able to achieve the high-level functionalities it has been designed for.

Performance testing is the most complex testing strategy because of the high number of moving parts and supporting services/resources. However, performance load tests are crucial to understand how

well the system can manage a wide set of users, especially in order to guarantee high availability and scalability.

In addition, it is highly suggested the adoption of Contract Testing every time an element relies on another element's interface, in this case microservices. Having well-formed contract tests makes it easy for developers to avoid many problems in such a complex type of architecture, assuring easy development and deployment.

Moreover, for client-side applications (web app, mobile app), it is important to mention that some sort of acceptance testing should be taken into account, for example using the user experience feedback to do a validation of the design activity. This aspect is particularly important if a future transition to micro front-ends is going to take place: in fact, the user experience feedback will be a crucial element to consider when testing the client-side functionalities.

Finally, since the S2B will follow the Digital Public Good principles, special attention must be given to security aspects. In fact, they should be carefully tested in order to satisfy the constraints defined by India's Personal Data Protection Act (PDPA), a set of regulations derived from the European General Data Regulation Protection (GDPR).

6 Effort Spent

In this section we provide detailed information about how much effort each group member spent in working at this document. Further information about commits and updates is stored in the project GitHub repository and on the others online tools we used respectively for diagrams design and teamwork's communication.

Section	Gori	Romanini	Watanabe
Architecure Structure (Microservices theory)	10	7,5	11,5
1.1		0,5	
1.2		0,5	
1.3	0,5		
1.4	2		
1.5			
1.6	1,5		
2.1		4,5	
2.2	8		
2.3	8		
2.4			14
2.5		4	1
2.6		7	
2.7		1,5	
3.0			12
4.0	6	1	
5.1		2	
5.2		3	
5.3		2	
6.0			
Whole document review	3,5	3,5	3,5
Total	39,5	37	42

Table 6: Table of efforts