STATE UNIVERSITY OF INTELLIGENT TECHNOLOGIES AND TELECOMMUNICATIONS

Faculty of Information Technology and Cybersecurity

Software Engineering Department

Final Project

on the topic of

"IT solution and services for Green Plant Market"

Completed by:

4th-year student, group SE 4.2.01TE

Yaroslav Holota

Head of the project Prof. Chepok A.O.

National scale _____

Number of points ___ ECTS grade ___

Odesa – 2023

CONTENTS

DEFINITIONS AND TERMS

MVU – Model View Update design pattern.

REST – Representational state transfer and an architectural style for distributed hypermedia systems.

JSON – Javascript object notation.

JWT – JSON web token.

SPA – Single page application.

CQRS – Command Query Responsibility Segregation.

DDD – Domain Driven Design.

INTRODUCTION

Informational Systems are a driving force behind the movement of automatization of business processes, since they allow businesses to streamline and greatly improve the efficiency of delivery of value and as such in increase in income.

The goal of the business is to sell and care for plants. Following from that, following tasks arise:

1. Care for the plants in preparation for their sale.
2. Put plants for sale and organize delivery through the postal service.
3. Provide customers and employees with instructions for plant care.
4. Track the history of orders and payments and present them in a form that would enhance management's decision making.

Following from the goal and tasks of the business the goal of this course work is to automate the process of plant selling and care. Following from this goal, the tasks of this project are as follows:

1. Analyze business domain and create a logical framework of this application, specified roles of actors, map business aggregates and the use cases that arise in-between them.
2. Select fitting software components.
3. Create Workflow descriptions that would solve implement those use cases.
4. Organize application architecture.
5. Create User Interface.

Additionally, due to auditability reasons, less-standard application architecture would be used, so the comparison between it and a more standard approach would be produced. The categories of performance, complexity of the implementation, and talent recruitment would be used for comparison.

1 REQUIREMENTS OF THE INFORMATIONAL SYSTEM

The business process that is being automated by this application has three main roles of actors: consumer, producer and manager. Table 1.1 includes the use cases as well as correspondence of input and output data related to them.

Table 1.1 – System Use Cases

Number	Use Case	Explanation	Input	Output
	Consumer, Producer, Manager			
S1	Access the system.		Login Password	Session
S2	Update your Password	User should only be able to update their own password and no other.	New Password	New Session
	Consumer, Producer			
A1	Search for plants that can be ordered.	Consumers have this task to be able to order plants. Producers have this task for analysis of posted plants.	Plant Groups Plant Soils Plant Regions Price Range Plant Name Plant Age	Plants that specify search requirements
A2	Search for instructions for plants.	If some input parameter has not been provided than there should be no filtering performed on that field.	Plant Group Instruction Title Instruction Description	Instructions: Title Description Cover Content

Continuation of Table 1.1

Number	Task	Explanation	Input	Output
A3	See detailed information for posted plant.	Consumer can do this to be able to perform more informed decision about ordering.	Plant Id	Plant Name Description Price Groups Soils Regions Plant Images Age Seller Credentials Caretaker Credentials
	Consumer			
B1	Order plant.		Post Id Delivery Address	Order Id
B2	See previously used addresses on order.	This would speed up delivery process and improve user experience.		Addresses: City Location
B3	Confirm order to be delivered.	This step may be automatically triggered when the postal system notifies package receival	Order Id	

Continuation of Table 1.1

Number	Task	Explanation	Input	Output
	Producer, Manager			
C1	Find plants that are being prepared for post.		Limit to Cared	Plants: Plant Name Plant Description Is cared flag
C2	Edit plant information.		Plant Id New Plant	New Plant
C3	Create plant.		Name Description Plant Regions Soils Groups Pictures Age	Plant Id
C4	See plant prepared for sale.	Seeing the plant as a client would see it before it is posted would allow producer to create better posts.	Plant Id	Plant Post with no price specified
C5	Post plant for sale.		Plant Id Price	Post Id
C6	Create Instruction.		Group Cover Title Description Content	Instruction Id

Continuation of Table 1.1

Number	Task	Explanation	Input	Output
C7	Find users.	This allows managers to manage producers and producers to manage	Name Phone Number	Users: Name Phone Number Roles
C8	Invite users.		Login Roles Email Name Phone Number	User created and email with temporary password send.
C9	Update user roles.	Only for roles with lesser priority than current user’s.	Login Role	
C10	Remove post.	For producers this is limited to their posts.	Post Id	
C11	Update instruction.		Instruction Id New Instruction	New Instruction
C12	Reject order.		Order Id	
C13	Start Order delivery.		Order Id Tracking Number	Delivery Id

Continuation of Table 1.1

	Manager			
D1	See popularity for plants based on their group.			Plant Groups: Income Stock Number Instructions
D2	See financial info for plant based on their group.		Time Range	Plant Groups: Income Sales Number Sold Percent
D3	See the history of changes performed to any item	This is needed for the reasons of transparency and auditing that is legally required from the business.		Changes list: User that performed changes Time Payload

2 INFORMATIONAL SYSTEM ARCHITECTURE

2.1 High-level overview

Requirements of the application that were provided before create a need for architecture that would allow them to be possible. In this case, we would be using three-tier architecture, whose diagram can be seen on fig. 2.1.

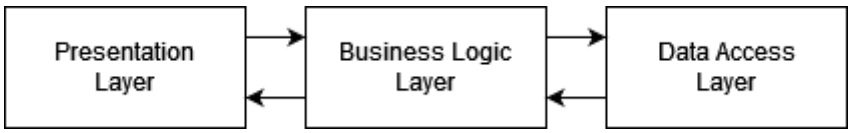

Figure 2.1 – Thee-tier architecture

Three-tier architecture allows us to segregate responsibilities of our architecture into parts in such a fashion that it is much easier to understand and modify them. The main advantage of such architecture over two-tier architecture is separation of client presentation and business logic, which allows us to create multiple versions of client presentations, such as mobile and desktop applications, that use the same business logic component.

2.2 Business logic layer architecture

Based on the requirements of having the projections of data in form of statistics and aggregations of data from many aggregates, the business logic would implement CQRS. It would allow the business logic to use both highly normalized data sets, which are beneficial for write operations, and highly de-normalized data sets, which are beneficial for read operations. While using this concept the command-query duality arises, where command is any operation that modifies the state of the system and query is an operation that requests the state of the system.

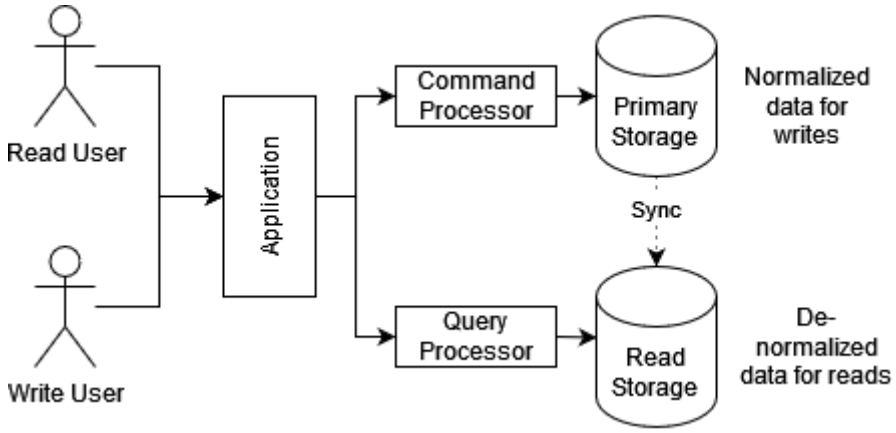

Figure 2.2 – CQRS implementation

Additionally, based based on the requirements for traceability and discoverability that were imposed upon the system we would be taking an approach of Event Sourcing [3]. The main idea of it consists in considering the events that led to the current state of the system as the source as opposed to considering the projection itself as the source of truth. Combining this idea with the CQRS, we may arrive at the architectural depicted on the fig. 2.3

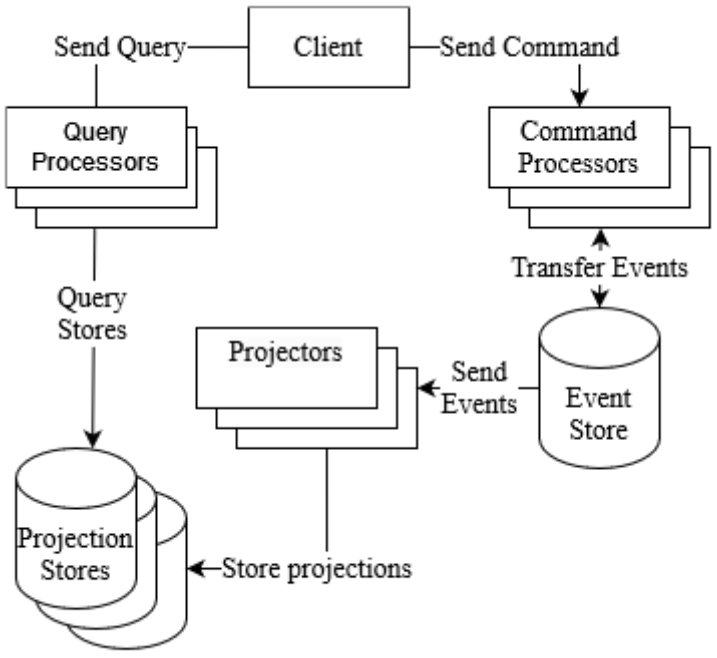

Figure 2.3 – Business layer architecture

Here, we separate reads and writes by the entry point, which results in us having a separate range of Command Processors and Query Processors. Command Processors are responsible for processing commands with the usage of the events stored with the Event Store. This is important due to concept from Event Sourcing that considers events as the Single Source of Truth. All projection that are needed to perform a command should happen on-the-fly from the set of events that appeared previously. So, after the command was processed some events may be produced in response to it. This would be picked up by the projectors, which would project the data into various Projection Stores for feature consumption. Due to this type of handling, we can use several projection stores that have their own benefits and drawbacks, so that we only use corresponding storage system in areas that are their strong points. As an example of that, we may use one data storage system for its search capabilities and use some other data storage system for its key lookup capabilities. Correspondingly, the query processor would identify proper projection store to query for the resource requested by the client. Correspondingly, the query processor would identify proper projection store to query for the resource requested by the client.

As we can clearly see there, we also have three deployable component groups:

- Command Processors
- Query Processors
- Projectors

Each of those also needs an internal architectural and compositional structure to use. For that the clean architecture [4] would be used as business layer architecture. Its main goal is to separate the actual business logic of the backend application from infrastructural logic that includes sending emails, querying database and interacting with file system. Its diagram can be seen on a fig. 2.4.

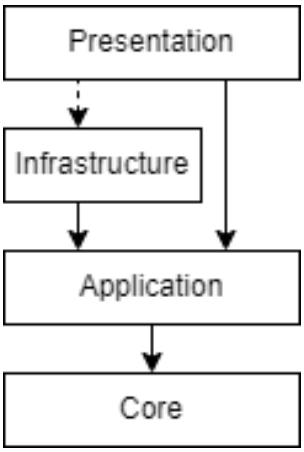

Figure 2.4 – Clean architecture

Here, we can see four structural layers:

- Core - the base of application and contains system-wide concerns and business entities representation
- Application - business logic
- Infrastructure - external dependencies
- Presentation - a medium for information transfer

Clean architecture has been chosen to allow for separation of business concerns and the actual infrastructure. This is achieved by Presentation layer components providing Infrastructure layer implementations for Application layer dependencies.

2.3 Presentation layer architecture

As a pattern for development of presentation layer MVU [5] pattern would be used. Here, model is an unambiguous and flat representation of all the information that is needed for the application, view is a function that renders model and convenes user interactions, update is a function that receives a model and a message and produces new model and optionally commands, side-effects externally processes commands and posts messages.

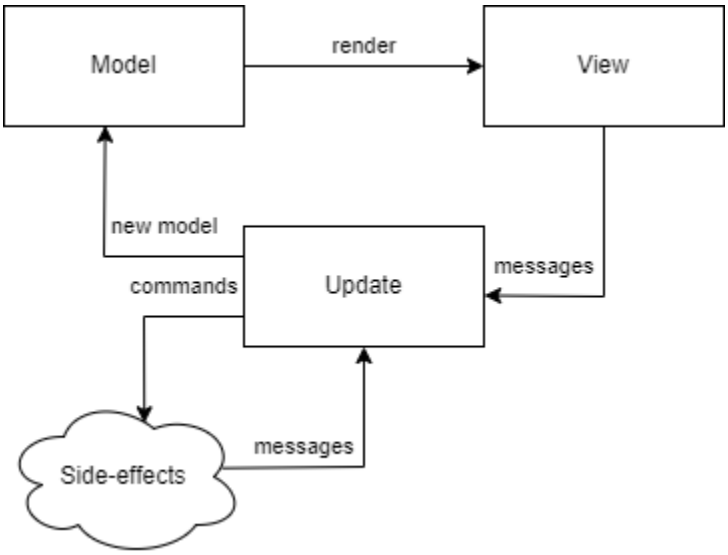

Figure 2.5 – MVU pattern

The MVU has been used to allow for predictable and deterministic user interface design, which allows us to save on doing testing.

Additionally, it is a form of the architectural structure of Event Sourcing that is already in use on the backend, which would cut down the number of concepts required from any developer working on the system.

3 MODELING OF THE DOMAIN OF INFORMATIONAL SYSTEM

3.1 Modeling data

To model the domain of informational system we would be using the concepts of aggregates, entities and values objects from the DDD [2]. Both aggregate and the entity are identifiable collections of fields that have a schema. The difference is in aggregates being considered as a Unit of Work which means that an entity exists within aggregate and may not be modified outside of it. On the other hand, value object are collections of data with some internal consistency logic that are not identifiable. Both aggregates and entities are identifiable by definition, so they would have an implicit surrogate id.

The full Domain Diagram is provided via Attachment A. Otherwise, aggregates, entities and their constraints in tabular form are presented in table 3.1.

Table 3.1 – Aggregates and their constraints

Field	Description	Constraints
“User” aggregate		
FirstName	First name of the user	NOT NULL
LastName	Last name of the user	NOT NULL
PhoneNumber	Phone number of the user	NOT NULL
Login	User-friendly identifier of the user	NOT NULL, UNIQUE
Roles	Role that the user has permissions for	NOT NULL, NOT EMPTY
PlantsCared	Number of plants cared for by the user	NOT NULL, NOT NEGATIVE
PlantsSold	Number of plants sold by the user	NOT NULL, NOT NEGATIVE

Continuation of Table 3.1

InstructionsCreated	Number of instructions created by the user	NOT NULL, NOT NEGATIVE
UsedAddresses	Delivery addresses previously used by the user	NOT NULL
“Delivery Address” value object		
City	City of the delivery	NOT NULL
MailNumber	Number of postal location	NOT NULL
“PlantStock” aggregate		
CaretakerId	Identitfier of User that is the caretaker	NOT NULL
PlantName	Name of the Plant	NOT NULL
Description	Description of the Plant	NOT NULL
RegionNames	Names of the regions	NOT EMPTY
SoilNames	Names of the soils	NOT EMPTY
GroupNames	Names of groups	NOT EMPTY
Pictures	Pictures of the plant	NOT NULL
CreatedTime	Time the plant was created in the real world	NOT NULL
“Picture” entity		
Location	Url from which the picture may be downloaded	NOT NULL
“PlantPost” aggregate		
StockId	Identifier of the Stock	NOT NULL
SellerId	Identifier of the User that is the Seller	NOT NULL

Continuation of Table 3.1

Price	Price of the posted item	NOT NULL, NOT NEGATIVE
“PlantOrder” aggregate		
PostId	Identifier of the Post	NOT NULL
BuyerId	Identifier of the User that is the buyer	NOT NULL
DeliveryAddress	Address of the delivery	NOT NULL
OrderTime	Time of order being requested	NOT NULL
DeliveryStartedTime	Time at which the delivery was started	
TrackingNumber	Tracking number for the delivery	
DeliveredTime	Time at which the order was delivered	
“Plant Instruction” aggregate		
GroupName	Name of the plant group for which the instruction is created	NOT NULL
Text	Content of the instruction	NOT NULL
Title	Title of the instruction	NOT NULL
Description	Description of the instruction	NOT NULL
Cover	Cover image	NOT NULL
“PlantsInformation” aggregate		
GroupNames	Used group names so far	NOT NULL
RegionNames	Used region names so far	NOT NULL
SoilNames	Used soil names so far	NOT NULL

Continuation of Table 3.1

TotalStats	Stats so far	NOT NULL
DailyStats	Day to the stats	NOT NULL
“PlantStats” value object		
GroupName	Name of group for which stats are collected	NOT NULL
PlantsCount	Number of stock items added	NOT NULL, NOT NEGATIVE
InstructionsCount	Number of instructions created	NOT NULL, NOT NEGATIVE
PostedCount	Number of posts created	NOT NULL, NOT NEGATIVE
SoldCount	Number of orders delivered	NOT NULL, NOT NEGATIVE
Income	Combined income for period	NOT NULL, NOT NEGATIVE

Additional constraints are presented in table 3.2

Table 3.2 – Additional constraints

Aggregate	Constraints
PlantStock	Cannot be updated if post record exists that references it. Age of the plant cannot be edited under any condition.
PlantPost	Cannot be deleted by anyone except a manager or the producer that created it.
PlantOrder	Cannot be deleted by anyone except a manager or the producer that created underlying post. Cannot be created for plants that are in a planning stage – their creation dates are after current date. Cannot be confirmed to be received by anyone outside of customer that ordered it.

There are two types of relationships between aggregates, entities and value objects:

- One-to-one
- One-to-many

“One-to-one” relationship exists between following items:

- PlantStock and User
- PlantPost and Stock
- PlantPost and User
- PlantOrder and PlantPost
- PlantOrder and DeliveryAddress

“One-to-many” relationship exists between following items:

- User and DeliveryAddress
- PlantStock and Picture
- PlantsInformation and PlantStats

3.2 Modeling Workflows

Workflows, by their definition, combine use cases and aggregates, defining proper relations between those and introducing limits for order of execution. As the first step to modeling the workflows we would create a correspondence between aggregates and the use cases. This correspondence would use the numbers for use cases from table 1.1. The User contains use cases S1-2, B2, C7-10; the PlantStock contains use cases C1-5; the PlantPost contains use cases A1, A3 and B1; the PlantOrder contains use cases B3, C12-13; the Instruction contains use cases A2, C6 and C11; the PlantsInformation contains use cases D1-2.

Once we have separated out the subdomain, we can map out their interactions, limitations and order of execution. This would be defined in the following figures: fig 3.1, fig 3.2, fig 3.3.

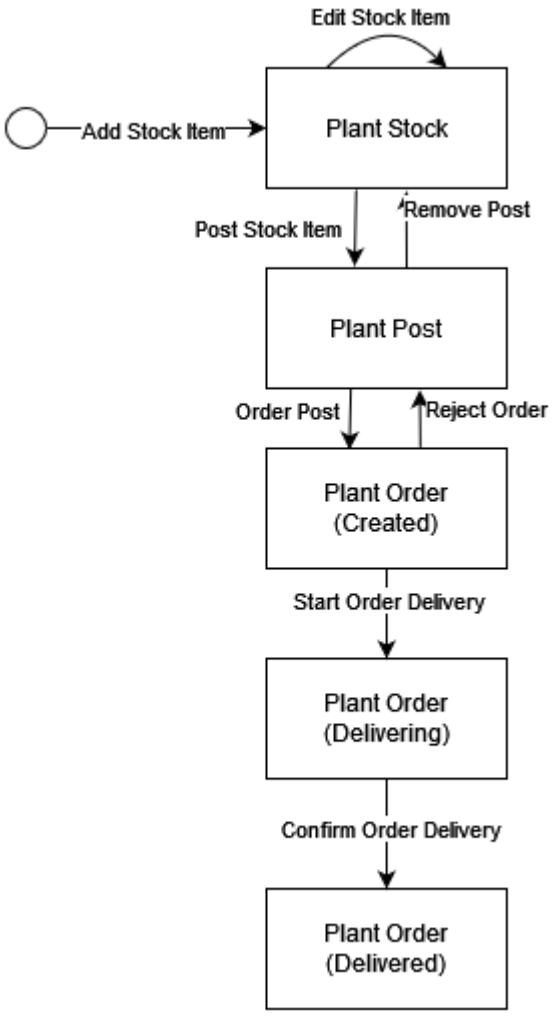

Figure 3.1 – Plant workflow

Plant Stock can be added via Add Stock Item use case, optionally edited via Edit Stock Item use case, and then posted via Post Stock Item use case after which Edit Stock Item Use Case becomes unavailable.

Plant Post may be removed via Remove Post use case by seller, going back to Plant Stock, or ordered via Order Post use case by a buyer.

Plant Order may be rejected, going back to Plant Post, or delivered which happens in two stages – start delivery from seller and configured delivery from buyer, which should appear in that order.

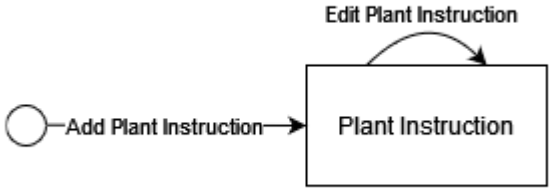

Figure 3.2 – Instruction workflow

Instruction may be added and then edited with no limitations outside of aggregates limitations.

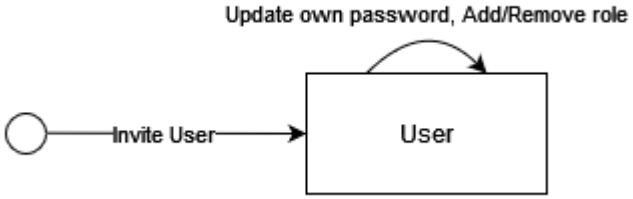

Figure 3.2 – User workflow

User may get invited, change their own password and have a role granted to them or revoked from them by a Producer or Manager.

4 USED TECHNOLOGIES AND SOFTWARE

The informational system is composed of three parts:

- Data Access layer is built with the usage of EventStore for Event Store system and MongoDb combined with ElasticSearch for Projection Stores
- Application Layer is built with ASP.NET Core framework [1]
- Presentation Layer is built with Elm, React and Bootstrap 5.

The EventStore was chosen as the Event Store for the following reason:

- Supports user-level access control.
- Throughout documentation.
- Actively supported and developed.
- Support for arbitrary data.

Two Projection Store of MongoDb and ElasticSearch were chosen for their performant key lookup and search queries and their support for user-level access control.

The ASP.NET Core framework for backend application has been selected for well-crafted database access packages, advanced support for creation of REST-full APIs and Microsoft support.

The frontend uses Bootstrap 5 for cross-platform support, accessibility and consistency of the user interface, Elm for its support of zero exception runtime and the guarantee of impossibility of undefined state of User Interface and React for its support for Single Page Application development. All of those frameworks are used within Node JS environment that uses Parcel bundler as a build tool for its support for minimization of static files. Build application is being distributed using Nginx web-host through nginx alpine docker image for its support for caching of static files.

5 STRUCTURE OF THE APPLICATION

The frontend application would be structured as one homogeneous application, where all users use one and the same application. However, only options that they would be able to execute are visible to them. This would not be used for defining access as the client would still be able to call all of those options through the Web API, where the access that the client should have would be forwarded to the corresponding event or projection store.

5.1 Backend architecture

The communication between frontend and backend would be organized through the REST-full API. It may be represented by many forms, but we would be taking a JSON HTTP api approach, whose diagram is presented in fig. 5.1.

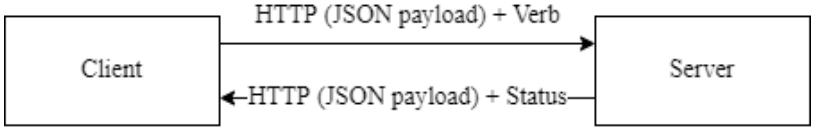

Figure 5.1 - REST API diagram

The authorization would be organized through the usage of JWT, which would be using two-way encryption to encode the data so that only the server that has private key is able to read it. Each of the three deployable components would use the Dependency Injection system to implement the abstraction replacement defined in the Clear Architecture. That means that business logic deals with abstractions, which would mean that the service itself would operate with the usage abstractions to implement its business logic, while the infrastructural dependencies, such as storage systems, file system, http servers, etc would be supplied through the infrastructure ports. A diagram of such interaction may be seen on fig. 5.2 and an example of it from the Projection component may be seen on the fig 5.3

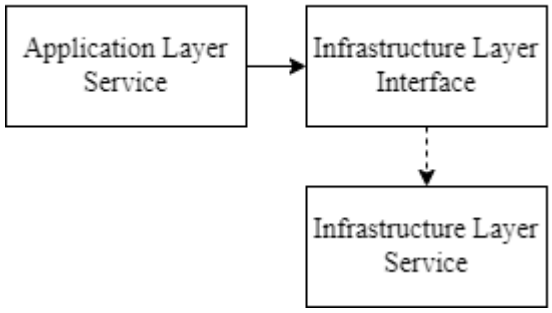

Figure 5.2 – Dependency Injection diagram

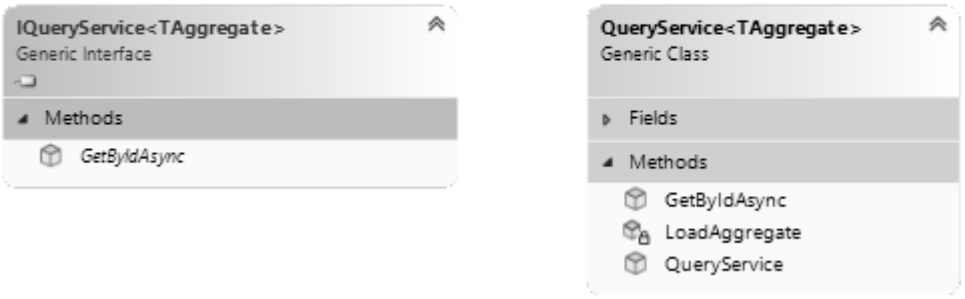

Figure 5.3 – Query service example

The events and commands should work with the strongly-types system, deriving from base Event and Command classes respectively. All of the aggregates should derive from AggregateBase, which would allow the support of high-level event handlers, command handlers and access controls. The event handlers should be attached through IEventHandler interface directly to an aggregate. They should be discovered on startup by the Domain library and be invoked when loading aggregates using the domain library. Command Handlers should be separated in pure and impure ones. Pure ones should be defined directly on the aggregate and use IDomainCommandHandler interface, whilst impure ones should use ICommandHandler and be defined externally, but both should also be automatically discovered.

5.2 Frontend architecture

The frontend application is structured as many MVU applications that represent a singular page of the application that acts as a SPA by wrapping pages with a SPA router.

There would be base application for MVU applications that would handle cases of unauthorized access and no login info being available.

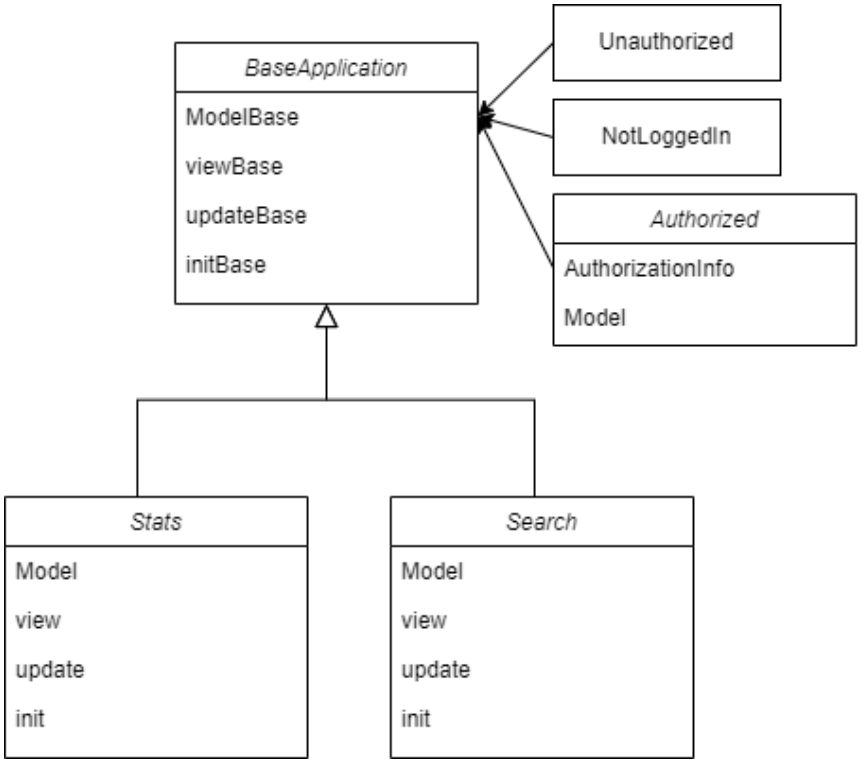

Figure 5.6 – Frontend architecture

6 APPLICATION IMPLEMENTATION

6.1 Backend

The backend takes a heavy advantaged from the Domain library that contains all of the basic definitions and abstractions. The most important interface would be considered here.

Basic definition for the base aggregate may be seen in the listing 6.1.

```
public abstract class AggregateBase
{
    public AggregateBase(Guid id)
    {
        Id = id;
        Metadata = new(GetType().Name, new());
    }

    public Guid Id { get; }

    public AggregateMetadata Metadata { get; private set; }

    public void Record(OneOf<Command, Event> newRecord)
    {
        Metadata.Record(newRecord);
    }
}
```

Listing 6.1 – AggregateBase definition

Definitions for Command and Event base classes may be seen in listing 6.2

```
public abstract record Command(CommandMetadata Metadata);
public sealed record CommandMetadata(Guid Id,
AggregateDescription Aggregate, DateTime Time, string Name,
string UserName, /*this is needed for
subscriptions*/AggregateDescription? InitialAggregate = null);

public abstract record Event(EventMetadata Metadata);
public sealed record EventMetadata(Guid Id, AggregateDescription
Aggregate, Guid CommandId, DateTime Time, string Name, ulong
EventNumber = ulong.MaxValue);
public record FailEvent(EventMetadata Metadata, string[]
Reasons, bool IsException) : Event(Metadata);
```

```
public record CommandProcessedEvent(EventMetadata Metadata) :
Event(Metadata);
```

Listing 6.2 – Command and Event definition

Command and event handler definitions can be seen in listing 6.3

```
public interface ICommandHandler<T> where T : Command
{
    Task<CommandForbidden?> ShouldForbidAsync(T command,
IUserIdentity userIdentity, CancellationToken token = default);
    Task<IEnumerable<Event>> HandleAsync(T command,
CancellationToken token = default);
}

public interface IDomainCommandHandler<T> where T : Command
{
    CommandForbidden? ShouldForbid(T command, IUserIdentity
userIdentity);
    IEnumerable<Event> Handle(T command);
}
/// <summary>
/// This is supposed to be applied to the aggregate
/// </summary>
public interface IEventHandler<T> where T : Event
{
    void Handle(T @event);
}
```

Listing 6.3 – Command and Event Handler

All of the use cases are defined are defined as types with the usage of Command base class for Commands and IRequest interface for queries. The full set of use case definitions may be seen in the Appendix B. However, and example of PlantPost use cases would be presented in the listing 6.4.

```
// Commands

public record AddToStockCommand(CommandMetadata Metadata,
PlantInformation Plant, DateTime CreatedTime, byte[][] Pictures)
: Command(Metadata);
public record StockAddedEvent(EventMetadata Metadata,
PlantInformation Plant, DateTime CreatedTime, Picture[]
Pictures, string CaretakerUsername) : Event(Metadata);
```

```
public record EditStockItemCommand(CommandMetadata Metadata,
PlantInformation Plant, byte[][] NewPictures, Guid[]
RemovedPictureIds) : Command(Metadata);
public record StockEdditedEvent(EventMetadata Metadata,
PlantInformation Plant, Picture[] NewPictures, Guid[]
RemovedPictureIds) : Event(Metadata);

public record PostStockItemCommand(CommandMetadata Metadata,
decimal Price) : Command(Metadata);
public record StockItemPostedEvent(EventMetadata Metadata,
string SellerUsername, decimal Price, string[] GroupNames) :
Event(Metadata);

// Queries

public record GetStockItems(PlantStockParams Params,
QueryOptions Options) :
IRequest<IEnumerable<StockViewResultItem>>;
public record GetStockItem(Guid StockId) :
IRequest<PlantViewResultItem?>;
public record GetPrepared(Guid StockId) :
IRequest<PreparedPostResultItem?>;

// Types

public record PlantInformation(
    string PlantName, string Description, string[] RegionNames,
    string[] SoilNames, string[] GroupNames
    );

public record PlantViewResultItem(string PlantName, string
Description, string[] GroupNames,
    string[] SoilNames, Picture[] Images, string[] RegionNames,
DateTime Created)
{
    public string CreatedHumanDate => Created.Humanize();
    public string CreatedDate => Created.ToShortDateString();
}
```

Listing 6.4 – PlantPost use cases

A PlantPost aggregate that uses all those abstractions may be seen in the listing 6.5

```
[Allow(Consumer, Read)]
[Allow(Consumer, Write)]
[Allow(Producer, Read)]
[Allow(Producer, Write)]
[Allow(Manager, Read)]
[Allow(Manager, Write)]
public class PlantPost : AggregateBase,
IEventHandler<StockItemPostedEvent>,
```

```
    IDomainCommandHandler<RemovePostCommand>,
IEventHandler<PostRemovedEvent>,
    IDomainCommandHandler<OrderPostCommand>,
IEventHandler<PostOrderedEvent>,
    IEventHandler<RejectedOrderEvent>
{
    public PlantPost(Guid id) : base(id)
    {
    }

    public decimal Price { get; private set; }
    public PlantStock Stock { get; private set; }
    public User Seller { get; private set; }
    public bool IsRemoved { get; private set; }
    public bool IsOrdered { get; private set; }

    public void Handle(StockItemPostedEvent @event)
    {

Metadata.Referenced.Add(new(@event.Metadata.Aggregate.Id,
nameof(PlantStock)));

Metadata.Referenced.Add(new(@event.SellerUsername.ToGuid(),
nameof(User)));
        IsRemoved = false;
        IsOrdered = false;
        Price = @event.Price;
    }

    public CommandForbidden? ShouldForbid(RemovePostCommand
command, IUserIdentity user) =>

(user.HasRole(Manager).Or(user.HasRole(Producer).And(IsSeller(us
er))))
        .And(IsNotRemoved)
        .And(IsNotOrdered);

    private CommandForbidden? IsNotRemoved() =>
        (IsRemoved is false).ToForbidden("Already removed");

    private CommandForbidden? IsNotOrdered() =>
        (IsOrdered is false).ToForbidden("Already ordered");

    private CommandForbidden? IsSeller(IUserIdentity user) =>
        (user.UserName == Seller.Login).ToForbidden("Cannot
remove somebody elses post");

    public IEnumerable<Event> Handle(RemovePostCommand command)
=>
        new[]
        {
```

```
            new
PostRemovedEvent(EventFactory.Shared.Create<PostRemovedEvent>(co
mmand))
        };

    public void Handle(PostRemovedEvent @event)
    {
        IsRemoved = true;
    }

    public CommandForbidden? ShouldForbid(OrderPostCommand
command, IUserIdentity user) =>
        user.HasAnyRoles(Manager,
Consumer).And(IsNotRemoved).And(IsNotOrdered);

    public IEnumerable<Event> Handle(OrderPostCommand command)
=>
        new[]
        {
            new
PostOrderedEvent(EventFactory.Shared.Create<PostOrderedEvent>(co
mmand), command.Address, command.Metadata.UserName)
        };

    public void Handle(PostOrderedEvent @event)
    {
        IsOrdered = true;
    }




    public void Handle(RejectedOrderEvent @event)
    {
        IsOrdered = false;
    }
}
```

Listing 6.5 – PlantsPost aggregate

Client-side usage of Command and Query service from the Domain library may be seen in the listing 6.6 and listing 6.7 correspondingly

```
  var result = await _command.SendAndNotifyAsync(
            factory => factory.Create<PostStockItemCommand,
PlantStock>(id),
            meta => new PostStockItemCommand(meta, price),
            token);
```

Listing 6. – Post Stock Item command invocation

6.2 Frontend

The base functions for creating an MVU application can be seen in the listing 6.8.

```
initBase : List UserRole -> model -> (AuthResponse -> Cmd msg) -
> Maybe AuthResponse -> ( ModelBase model, Cmd msg )
initBase requiredRoles initialModel initialCmd response =
    case response of
        Just resp ->
            if intersect requiredRoles resp.roles then
                ( Authorized resp initialModel, initialCmd resp
)

            else
                ( Unauthorized, Cmd.none )

        Nothing ->
            ( NotLoggedIn, Cmd.none )

type ModelBase model
    = Unauthorized
    | NotLoggedIn
    | Authorized AuthResponse model

viewBase : (AuthResponse -> model -> Html msg) -> ModelBase
model -> Html msg
viewBase authorizedView modelB =
    case modelB of
        Unauthorized ->
            div [] [ text "You are not authorized to view this
page!" ]

        NotLoggedIn ->
            div []
                [ text "You are not logged into your account!"
                , a [ href "/login" ] [ text "Go to login" ]
                ]

        Authorized resp authM ->
            authorizedView resp authM
```

Listing 6.8 – base initialization, view functions and model structure

The definition of base application can be seen on listing 6.9.

```
mainInit : (Maybe AuthResponse -> D.Value -> ( model, Cmd msg ))
-> D.Value -> ( model, Cmd msg )
```

```
mainInit initFunc flags =
    let
        authResp =
            case D.decodeValue decodeFlags flags of
                Ok res ->
                    Just res

                Err _ ->
                    Nothing
    in
    initFunc authResp flags

type alias AuthResponse =
    { token : String
    , roles : List UserRole
    , username : String
    }

type alias ApplicationConfig model msg =
    { init : Maybe AuthResponse -> D.Value -> ( model, Cmd msg )
    , view : model -> Html msg
    , update : msg -> model -> ( model, Cmd msg )
    , subscriptions : model -> Sub msg
    }

baseApplication : ApplicationConfig model msg -> Program D.Value
model msg
baseApplication config =
    Browser.element
        { init = mainInit config.init
        , view = config.view
        , update = config.update
        , subscriptions = config.subscriptions
        }
```

Listing 6.9 – base application

The router of routing single page application can be seen on listing 6.10.

```
const App = () => (
  <BrowserRouter>
    <Routes>
      <Route path="/login" element={<Login
Page isNew={false} />} />
      <Route path="/login/new" element={<LoginPage isNew={true}
/>} />
      <Route path="/stats" element={<StatsPage />} />
      <Route path="/search" element={<SearchPage />} />
      <Route path="/notPosted" element={<NotPostedPage />} />
      <Route path="/plant/:plantId" element={<PlantPage
isOrder={false} />} />
      <Route
```

```
        path="/plant/:plantId/order"
        element={<PlantPage isOrder={true} />}
      />
      <Route path="/notPosted/:plantId/post"
element={<PostPlantPage />} />
      <Route path="/notPosted/add" element={<AddEditPage
isEdit={false} />} />
      <Route
        path="/notPosted/:plantId/edit"
        element={<AddEditPage isEdit={true} />}
      />
      <Route path="/orders" element={<OrdersPage
isEmployee={false} />} />
      <Route
        path="/orders/employee"
        element={<OrdersPage isEmployee={true} />}
      />
      <Route path="/user" element={<UsersPage />} />
      <Route path="/user/add" element={<AddUserPage />} />
      <Route path="/instructions"
element={<SearchInstructionsPage />} />
      <Route
        path="/instructions/add"
        element={<AddInstructionPage isEdit={false} />}
      />
      <Route
        path="/instructions/:id/edit"
        element={<AddInstructionPage isEdit={true} />}
      />
      <Route path="/instructions/:id" element={<InstructionPage
/>} />
      <Route path="/profile" element={<ProfilePage />} />
      <Route path="*" element={<NotFound />} />
    </Routes>
  </BrowserRouter>
);
```

Listing 6.10 – application router

All of the pages are applications that use base application template, as shown in the listing 6.11 with the model, view, update and main function of Login page.

```
type alias Model =
    { username : String
    , password : String
    , status : Maybe (WebData CredsStatus)
    }

view : Model -> Html Msg
view model =
    Grid.containerFluid [ style "height" "100vh" ]
```

```
        [ Grid.row [ Row.attrs (fillParent ++ flexCenter) ]
            [ Grid.col [] []
            , Grid.col [] []
            , Grid.col [ Col.middleXs ]
                [ viewForm model, viewBackground
                ]
            , Grid.col [] []
            , Grid.col [] []
            ]
        ]
type Msg
    = UsernameUpdated String
    | PasswordUpdate String
    | Submitted
    | SubmitRequest (Result Http.Error AuthResponse)

update : Msg -> Model -> ( Model, Cmd Msg )
update msg model =
    case msg of
        UsernameUpdated login ->
            ( { model | username = login, status = Nothing },
Cmd.none )

        PasswordUpdate pass ->
            ( { model | password = pass, status = Nothing },
Cmd.none )

        Submitted ->
            ( { model | status = Just Loading }, submit model )

        SubmitRequest (Ok response) ->
            ( { model | status = Just <| Loaded GoodCredentials
}, notifyLoggedIn <| encodeResponse response )

        SubmitRequest (Err err) ->
            ( { model | status = Just <| Loaded BadCredentials
}, Cmd.none )

main : Program D.Value Model Msg
main =
    baseApplication
        { init = init
        , view = view
        , update = update
        , subscriptions = subscriptions
        }
```

Listing 6.11 – login page components

The main module and other cross-page modules can be found in Appendix C. The navigational diagram for pages can be found in Appendix D.

7 USER GUIDE WITH ILLUSTRATIONS

This section explores achievement of user tasks through the application user interface.

7.1 Consumer

The initial page of the application is the login page. Its illustration can be seen on fig.7.1. It contains two fields for login and password. There is no way of performing registration, because the system is invite-only. Your credentials should be passed to you through email.

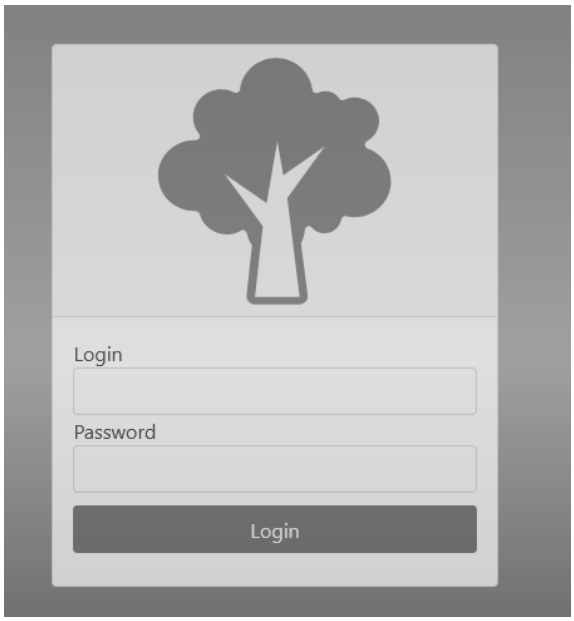

Figure 7.1 – Login Page

The page that you would be forwarded to is the search page. Its illustration can be seen on the fig 7.2. This page contains left-sided navigational bar that is used for the majority of navigation within the application. On the top of the page there are a few inputs for various properties for a plant you are looking for. Upon selecting any of them found list that is displayed below selectors would get updated. From this page you can navigate to order and plant pages by selecting specified buttons of the search result item accordingly.

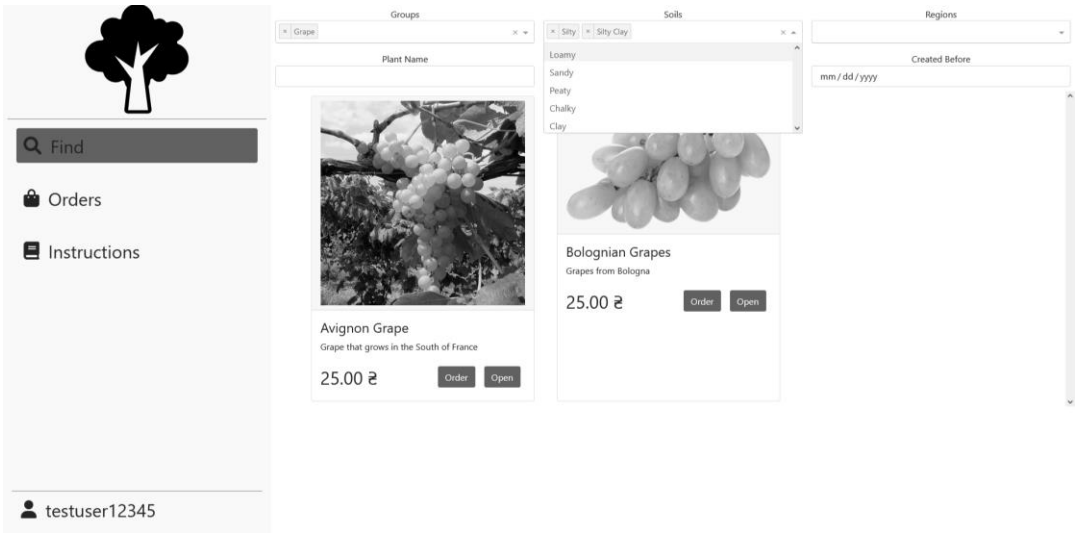

Figure 7.2 – Consumer Search page

Page with the detailed plant information can be accessed through search page. Its diagram can be found on fig 7.3. It displays information about plants region, group, age and soil as well as information about its caretaker and seller. From this page you can navigate to ordering page.

Figure 7.3 – Plant page

Order page displays most important information about plant and allows customer to select payment method as well as delivery address. Its illustration can be found in fig. 7.4. Delivery can be selected out of the list of existing or created on the fly. Upon selecting confirm order an order would be created. The order can found on Orders page that can be accessed through left navigational bar.

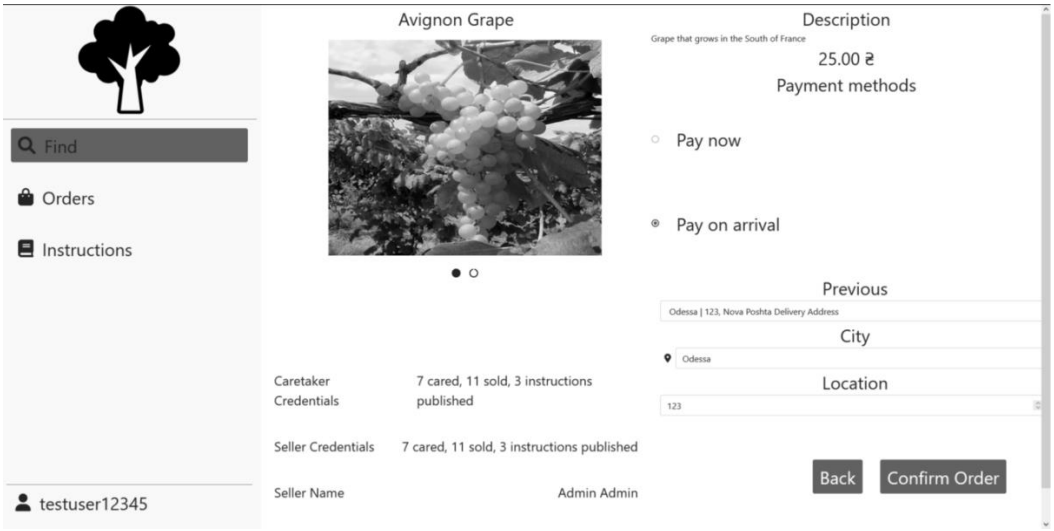

Figure 7.4- Order page

Orders page displays all of the orders that have been made by current customer and allows the customer to confirm the delivery of some order. Its illustration can be seen on fig. 7.5. The status of the plant can have following values:

1) Created – order have not started the delivery
2) Delivering – order have started delivery.
3) Delivered – order have been delivered.

An interaction of confirming delivery can only be performed on delivering status orders. This page allows you to hide delivered orders by checking top-left checkbox.

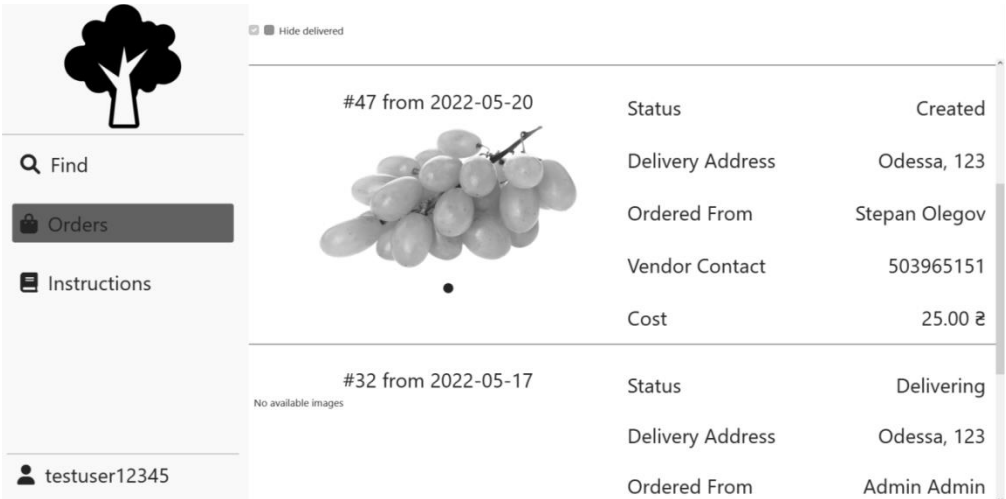

Figure 7.5 – Consumer Orders page

The instructions page is accessible through the left navigational bar and it displays a search page for instructions that acts the same way as plants search page does. Its illustration can be found on fig. 7.6. This page allows you to change filtering options and then open one for the full view.

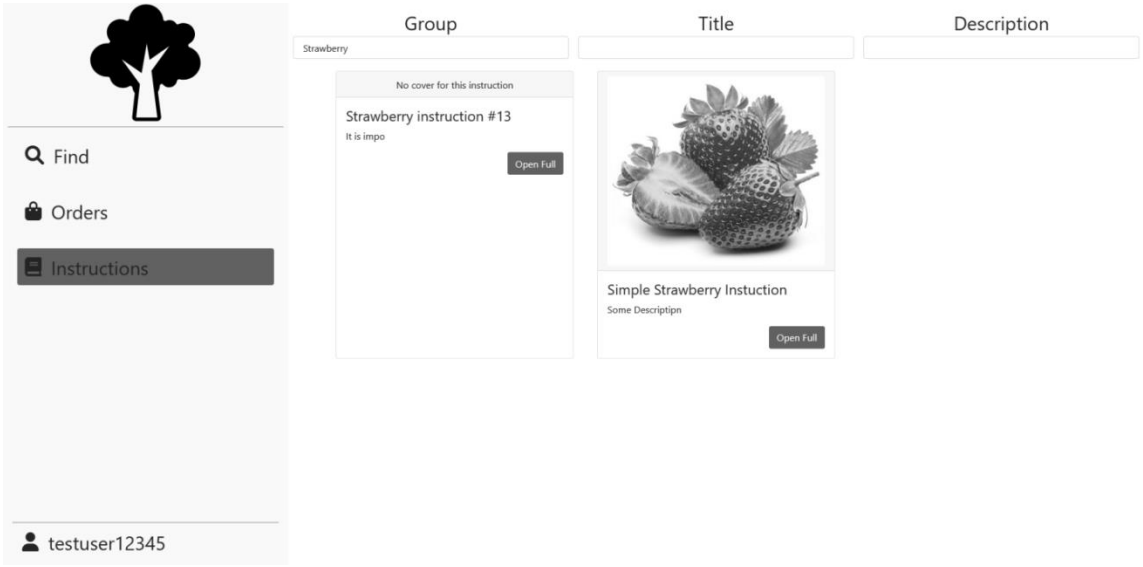

Figure 7.6 – Instructions page

Upon opening instruction for the full view you would see Instruction page that displays all of the relevant information about instruction including its main text that is richly formatted. Its illustration can be seen on fig 9.7. The only interaction is going back to the search page.

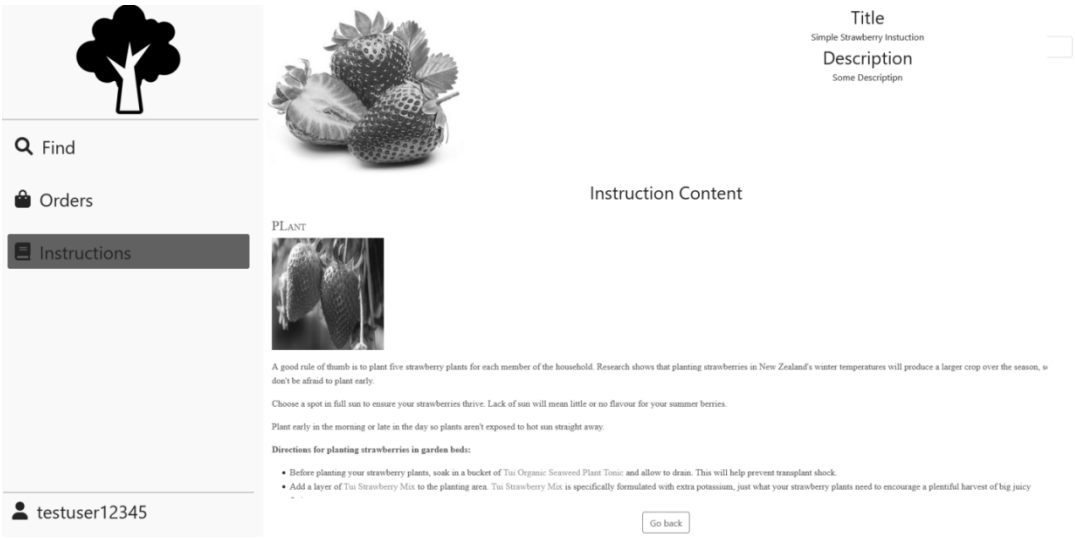

Figure 7.7 – Instruction page

Profile page can be accessed through left-sided navigational bar and it allows the user to change their password or logout of the system. Its illustration can be seen on fig 7.8.

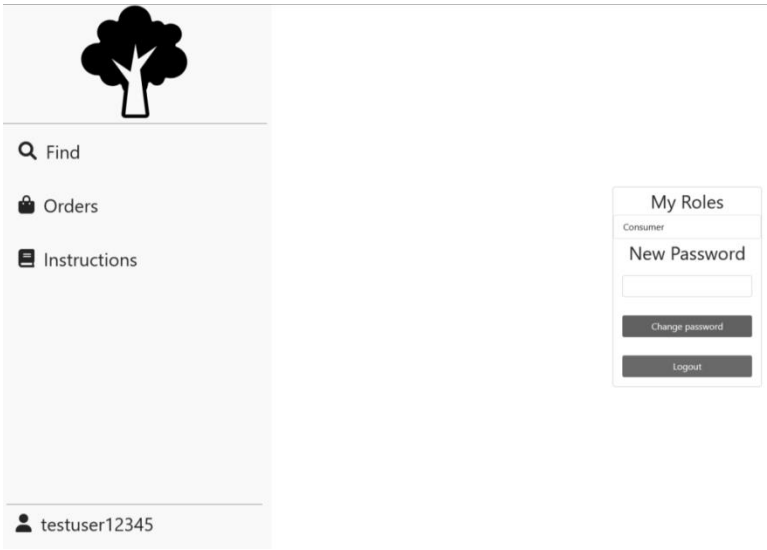

Figure 7.8 – Profile page

7.2 Producer

Producer can access the search page alongside consumer, but the producer would not be able to order the plant. Instead of that producer has interaction to remove the post. This can only be performed for posts that have been created by current producer or by manager. Its illustration can be seen on fig 7.9.

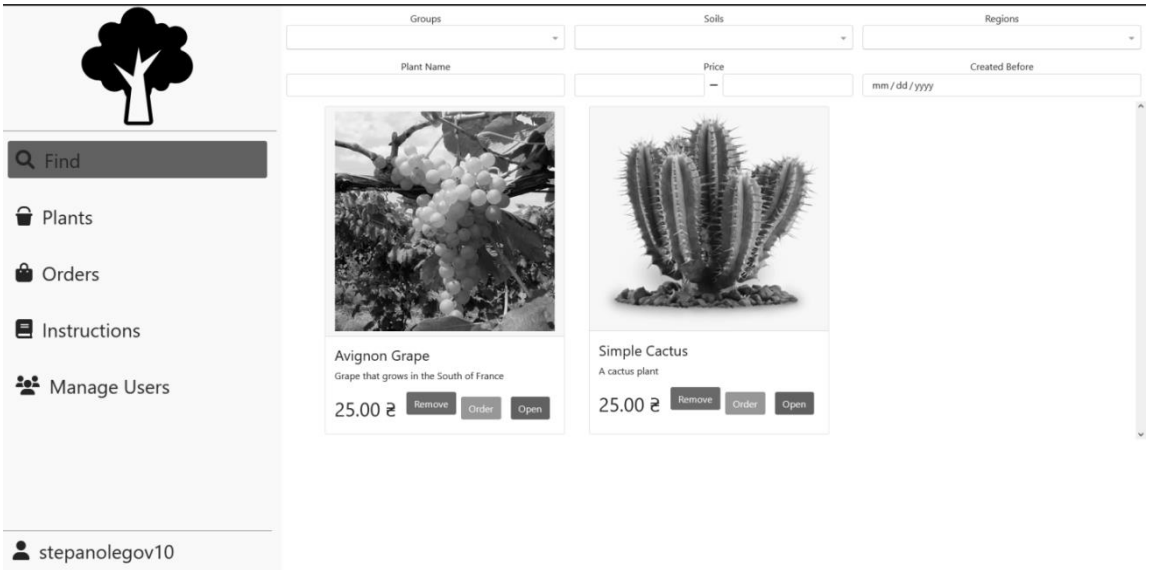

Figure 7.9 – Producer Search page

Plants page can be accessed through the left-sided navigational bar. It allows the producer to find all of the plants that are being current cared for before they are old enough to be posted for sale. Its illustration can be seen on fig 7.10. It has an option to hide all plants that are being cared for by other producers. It allows producer to add, edit and post a plant that opens corresponding pages.

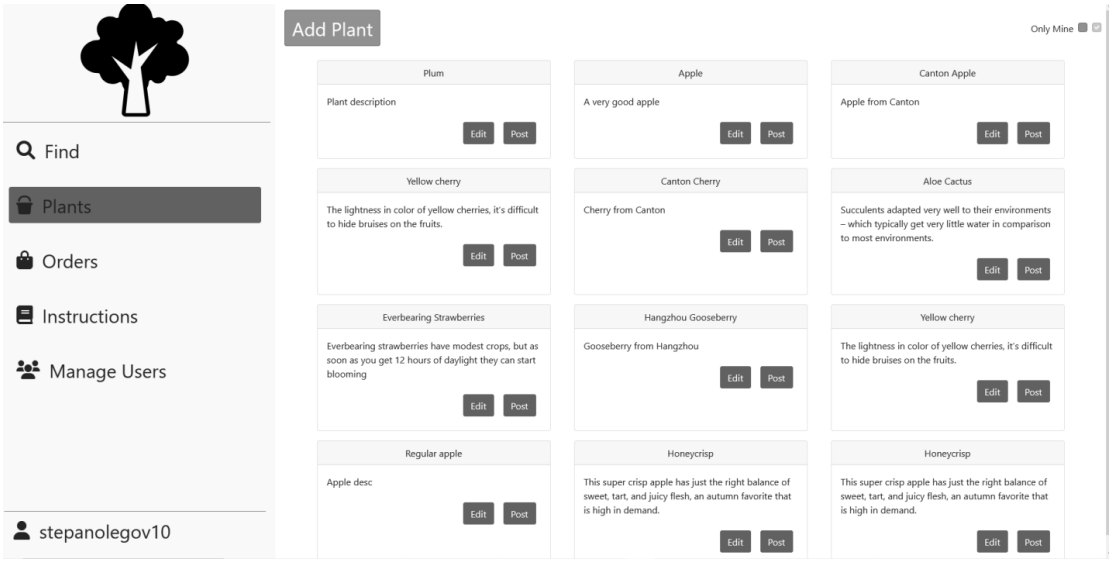

Figure 7.10 – Plants page

Add plant page can be accessed by selecting add plant in plants page. It allows the producer to input all of the information for the plant. Its illustration can be seen on fig 7.11.

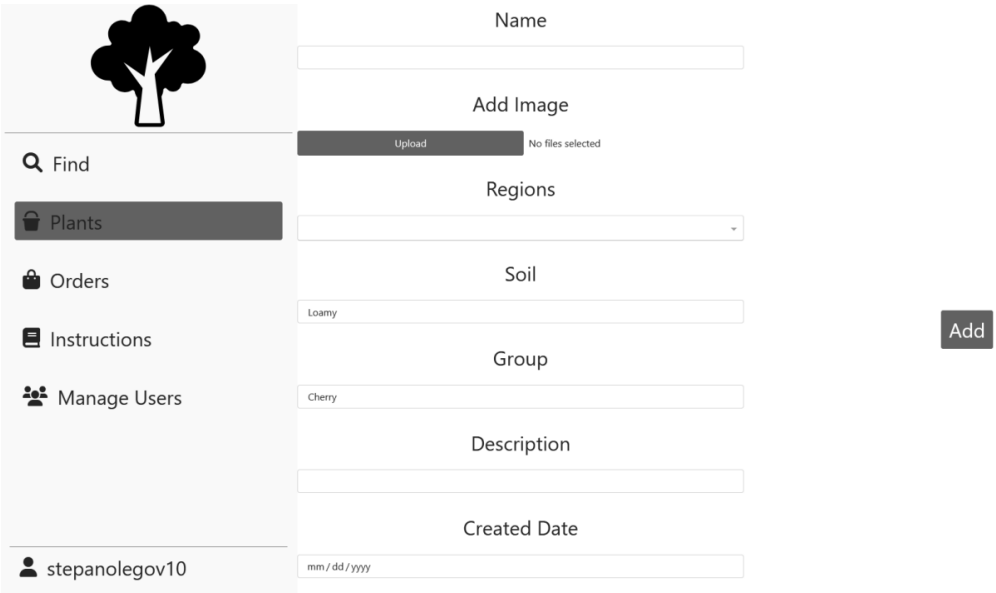

Figure 7.11 – Add plant

Edit plant page is accessible through selecting edit on plant from plants page. Its illustration can be seen on fig 7.12. It allows the producer to change the information about the plant with the limitation of Created Date not being editable. Upon clicking Save Changes the changes would apply.

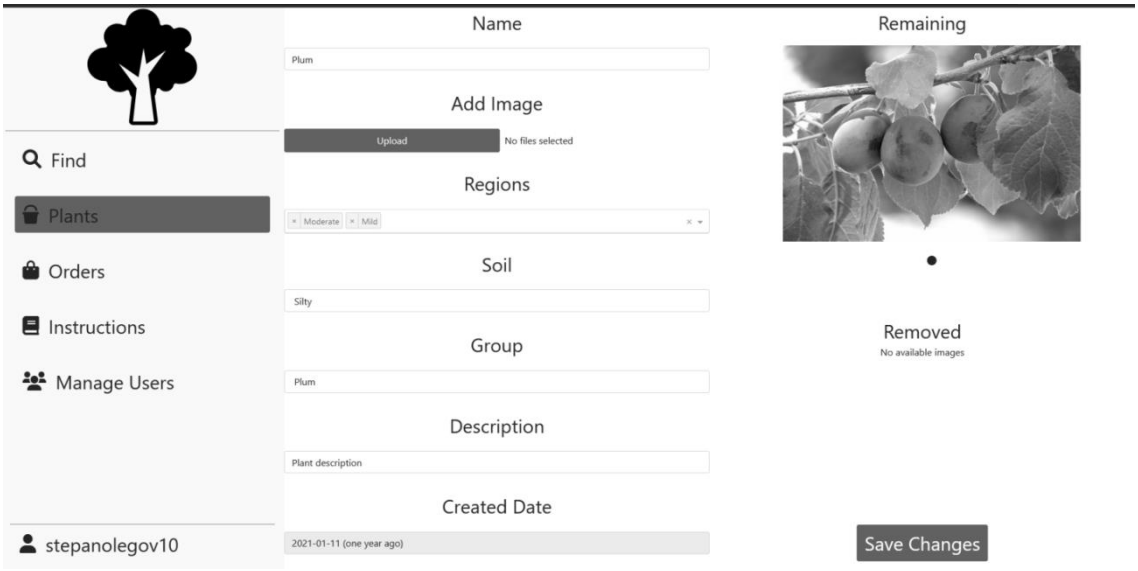

Figure 7.12 – Edit plant

Add instruction page can be accessed through Instruction page for producers, it allows the producer to create an instruction. Its illustration can be seen on fig 7.13. Upon clicking on edit text a full-screen text editor would be opened. After clicking on Create an instruction would be created.

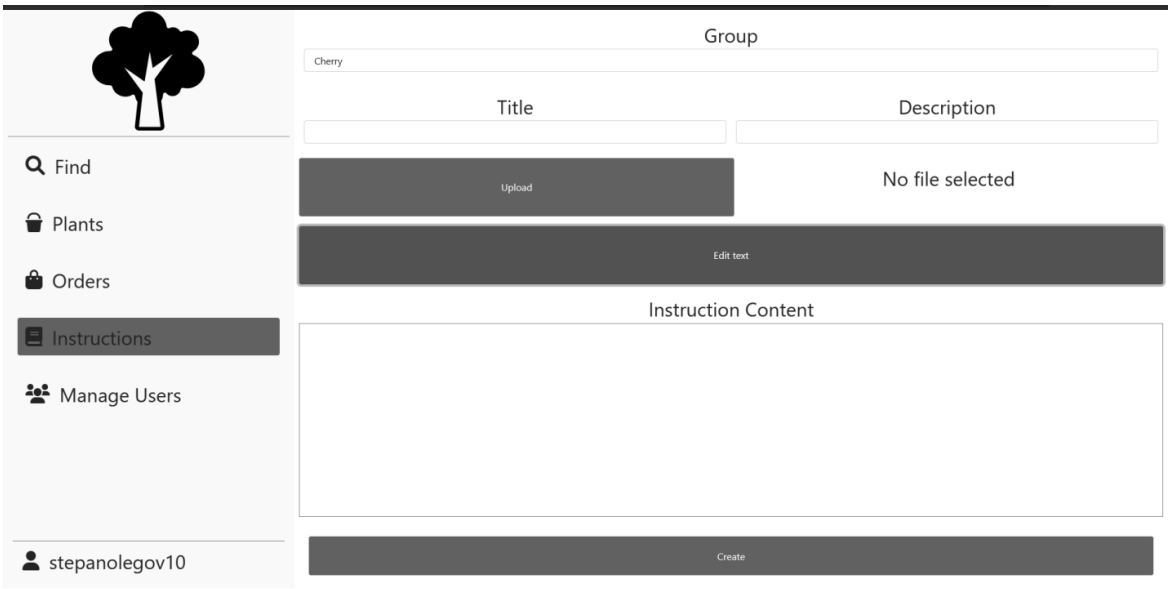

Figure 7.13 – Add instruction

Edit instruction page is accessible through instructions page by clicking on edit on an instruction. Its illustration can be seen on fig 7.14. It allows the producer to change any information about an instruction.

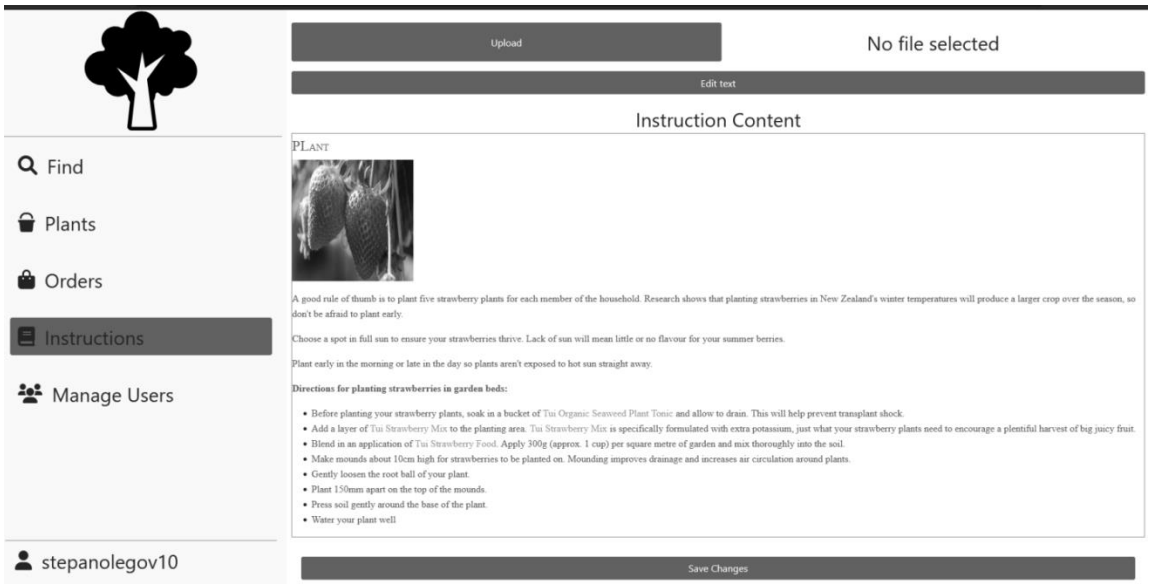

Figure 7.14 – Edit instruction

Orders page is accessible through left navigational bar. Its illustration can be seen on fig 7.15. It displays all of the orders that have been created so far with their statuses being the same as for consumer. However, for producer the interaction is with Created status orders – a producer can decided to reject it or confirm it as being sent by providing a delivery tracking number.

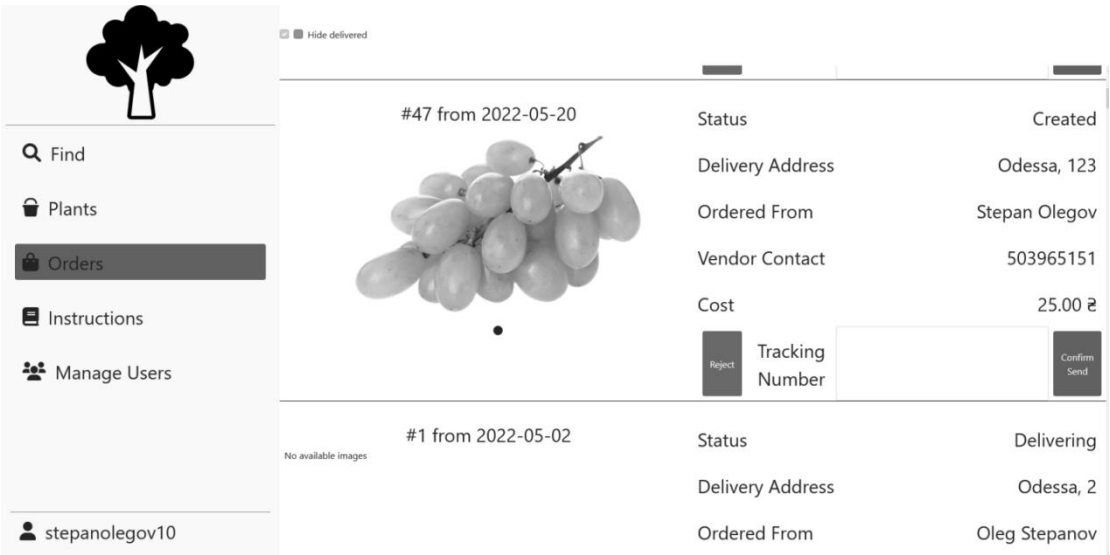

Figure 7.15 – Producer Orders page

Users page can be accessed through left navigational bar. Its illustration can be seen on fig 7.16. It displays a search by users and it allows a producer to grant producer role to some customer or to revoke customer access as well as an ability to create a user.

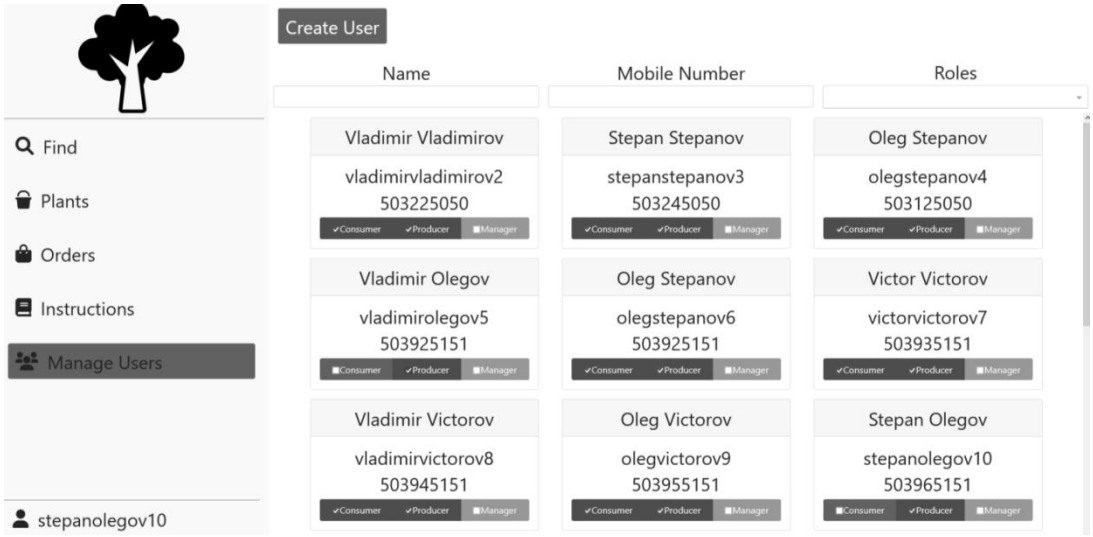

Figure 7.16 – Users page

Add user page displays information required to create a user. Its illustration can be seen on fig 7.17. Upon selecting all of the information and clicking on invite an invite would get send to the selected email.

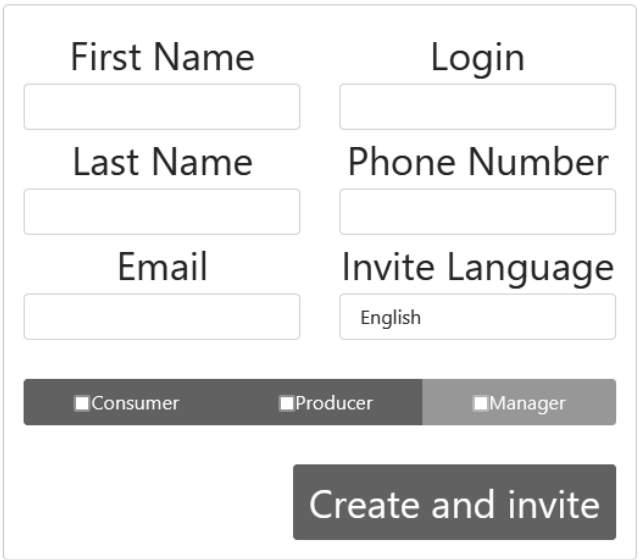

Figure 7.17 – Add user page

7.3 Manager

Managers have access to statistics pages that can be accessed through left sided navigational bar. There are two statistics pages: totals statistic page that can be found on fig. 7.18 and financial statistic page that can be found on fig 9.19. Those pages display pie charts for information plant information based on the plant group. Upon selecting a group on pie chart detailed information on it would get displayed in a table below it. Besides that, a manager has access to granting and removing more roles than producer and can remove any post or order.

Figure 7.18 – Total statistics page

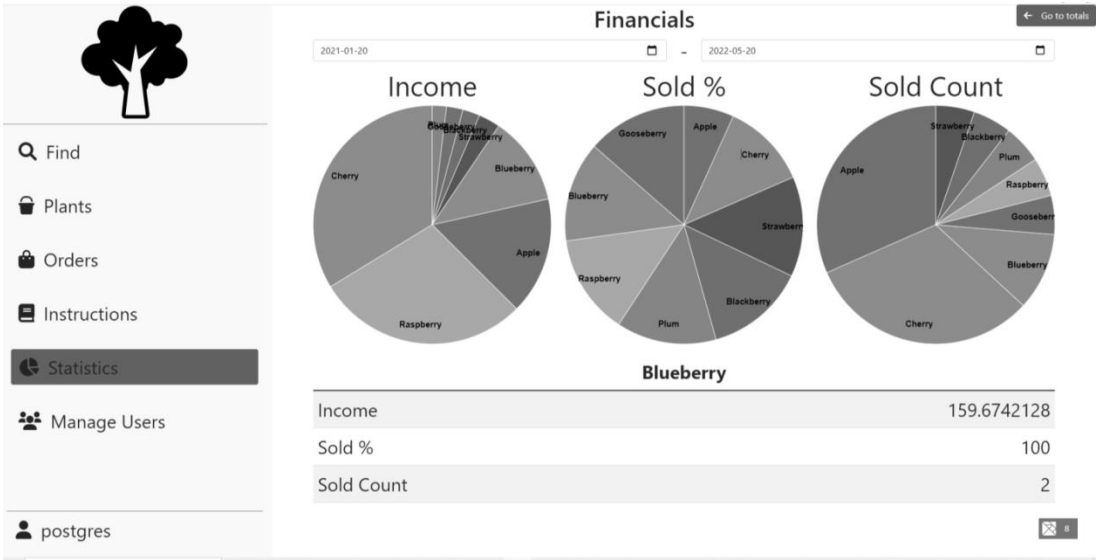

Figure 7.19 – Financial statistics page

In addition to Statistics page, users that are Managers would also see additional “View History” button in many places. Example of such buttons may be seen on fig. 7.20 and 7.21.

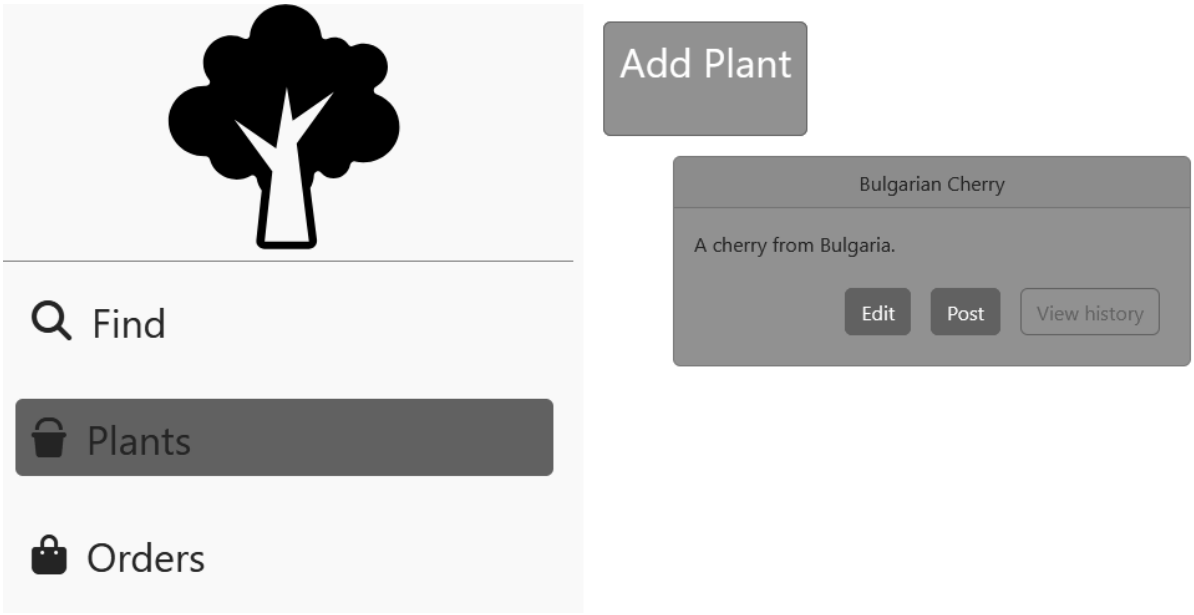

Figure 7.20 - Stock page with history button visible

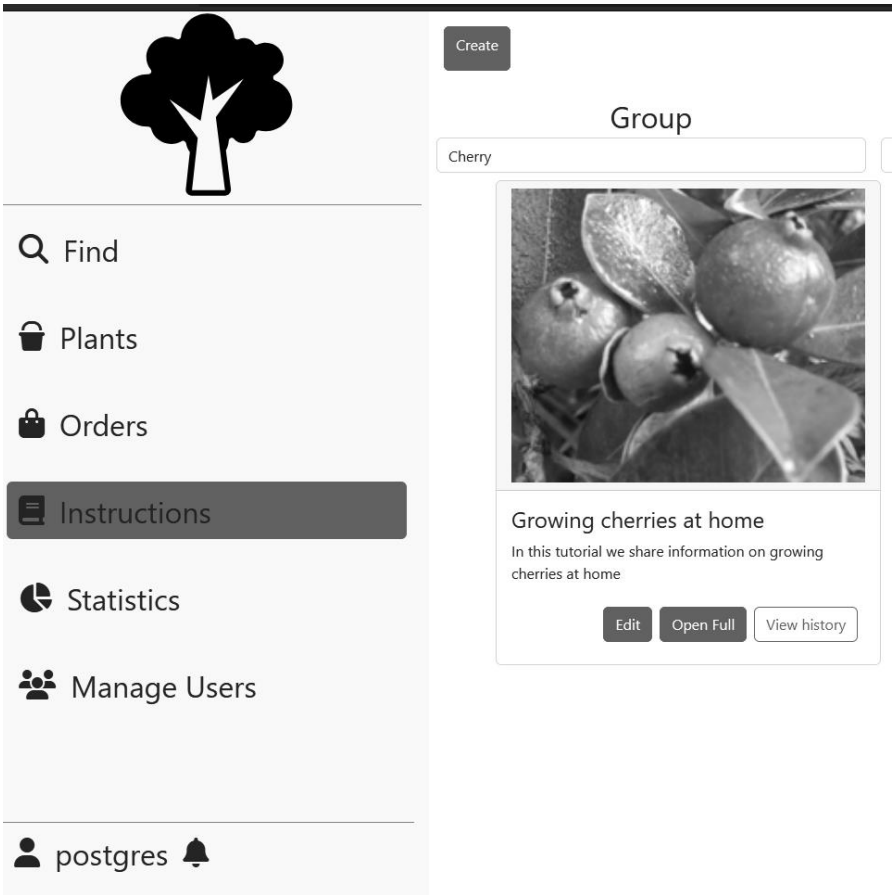

Figure 7.21 – Instructions page with history button visible

Upon clicking on “View History” button, the use would be transported to the history page that may be seen on fig. 7.22. It is showing all of the operations perform with this aggregation in the historical order. The user may reverse the order by checking “Reverse order” checkbox or limit the operations to ones that happened before the specified time. Upon clicking on any of visible commands in would be expanded to the view that may be seen on fig. 7.23.

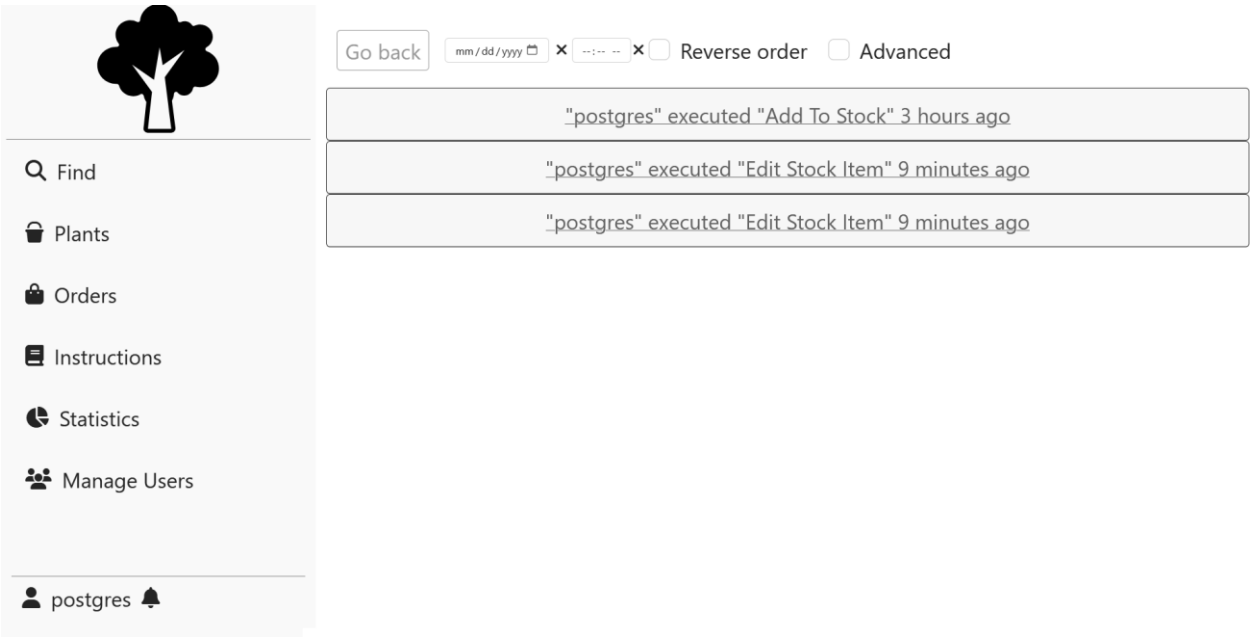

Figure 7.22 – History page

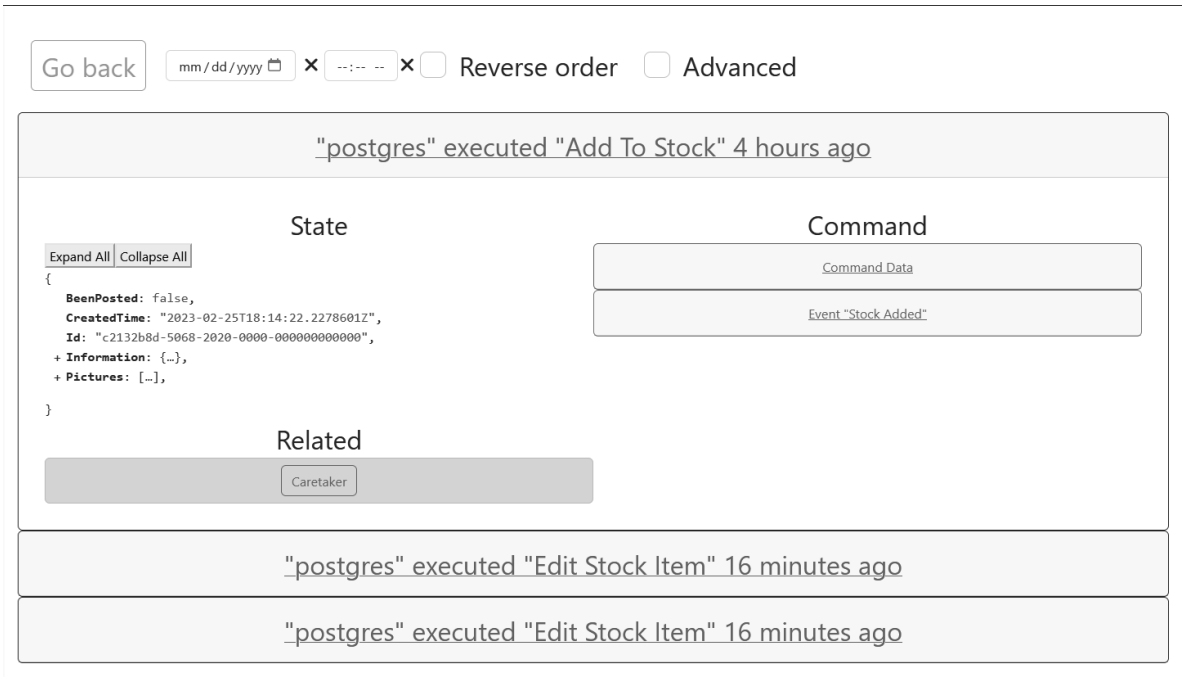

Figure 7.23 – History page with expanded operation

Once expanded, each operation would contain the state of the aggregate after the operation under the State column, related aggregates list that upon clicking on them would lead to the history of specified user and data related the request that was made by the user labeled as Command Data and results that were produced by the system labeled as Event with some name. Both command and event data may be expanded as is visible on fig. 7.24. That data for state, event and command may be expanded or collapsed by clicking on Expand/Collapse All or clicking on the field of interest.

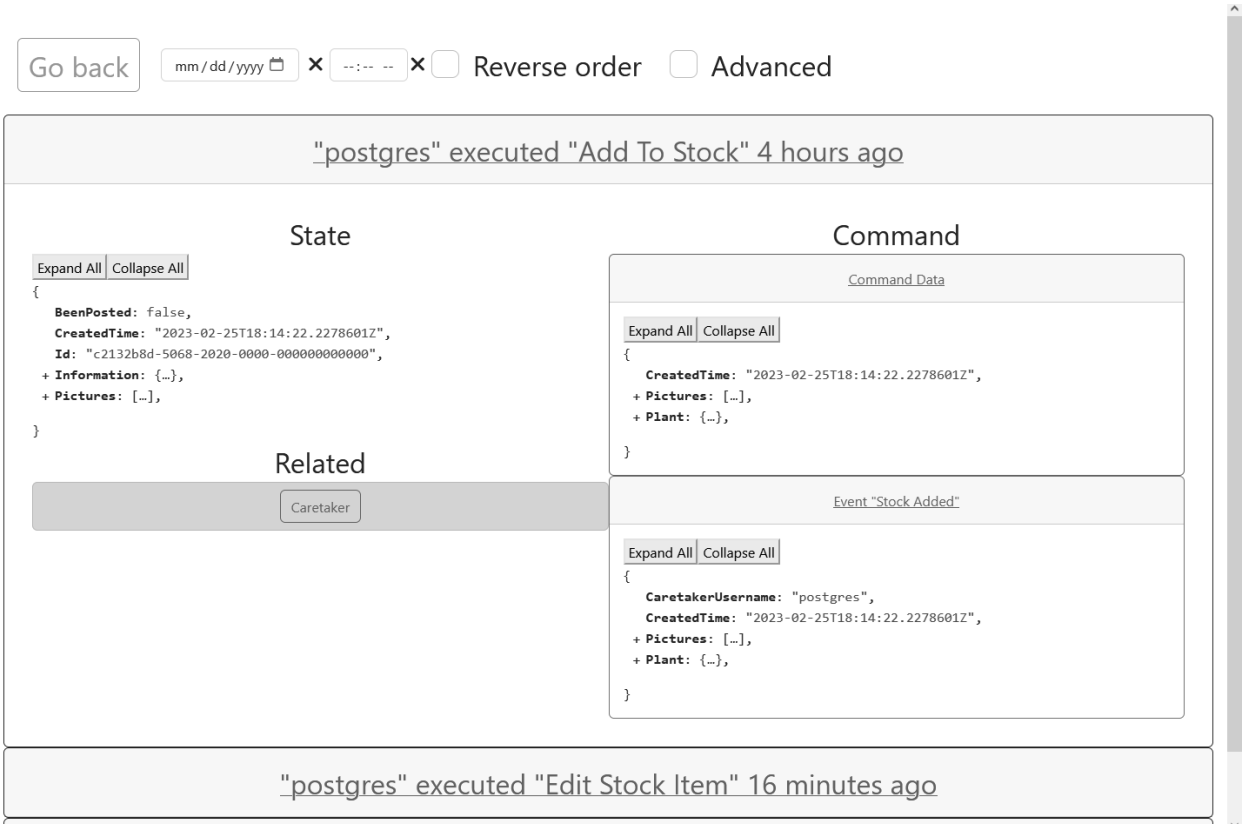

Figure 7.24 – Expanded Command and Event views

There is also checkbox that enables advanced mode, which is labeled with “Advanced” tag. Once checked, it would display additional Metadata button for State, Event and Command as is show on the fig. 7.25. Once clicked the button would display some additional information regarding each of those items in an overlaid windows, as may be seen on fig. 7.26.

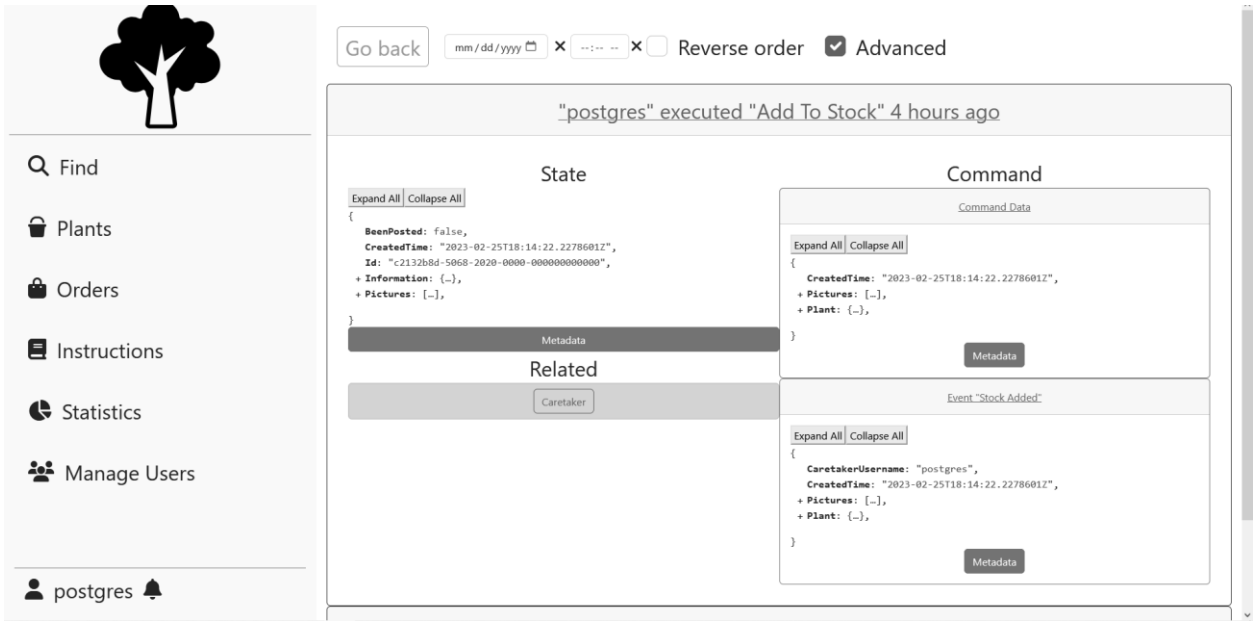

Figure 7.25 – Advanced mode of history page

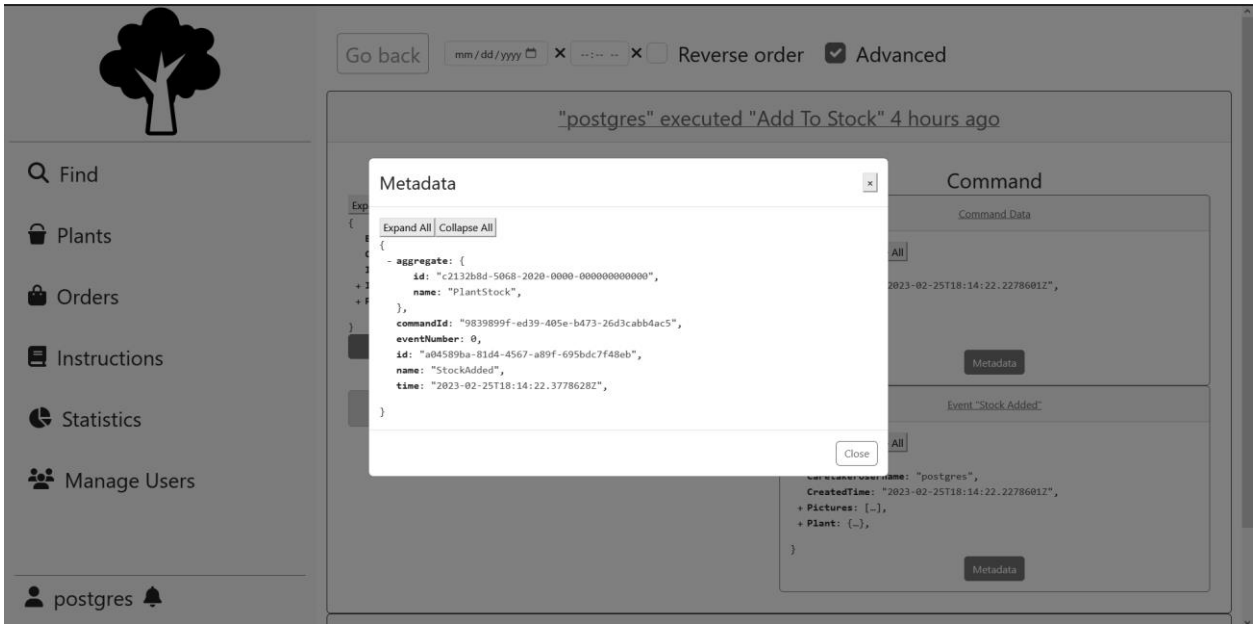

Figure 7.26 – Metadata overlay on the history page

CONCLUSIONS

In this course work, the business process has been automated that included creating software requirements, determining business entities and determining the architecture.

The backend, frontend and the database have been created, which constituted a fully-functional prototype of the software project that fulfilled all of the business requirements as well as being adherent to the laid out architectural requirements.

Potential venues of expansion may include:

- Implementation of bulk operations for ordering and posting
- Rating and comment system
- Addition of client-producer messaging service
- Additional of new payment methods

The technological choices were able to fulfill business requirements, but they were using less-standard approaches had its advantages and disadvantages. On the category of performance the result is inconclusive and depends on the context – in single user single deployable instance scenario the standard approach would have a better response time and would perform less operations overall. However, the event-driven approach has a better scaling ability, which increases multi-processing capabilities of the system. On the category of complexity of implementation the event-driven approach is much more complex as it requires much more infrastructure, and there is a larger disconnect between infrastructure to application layer. On the category of talent recruitment the conventional database-driven approach is a better choice, due to developers already having experience with such technologies. Additionally, I would add that event-driven approach has its benefits in form of having easier experience with tracing and producing historical data. Overall, the technological and architectural choices were a mixed success.

REFERENCES

1. Lock A. ASP.NET Core in Action – Manning, 2018. – 278 p.
2. Evans E. Domain-Driven Design: Tackling Complexity in the Heart of Software, 2003 – 560 p.
3. Garofolo E. Practical Microservices: Build Event-Driven Architectures with Event Sourcing and CQRS – 292 p.
4. Common web application architectures: Microsoft Docs - https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures.
5. Elm Architecture Documentation - https://guide.elm-lang.org/architecture.

APPENDIX A

Domain Diagram

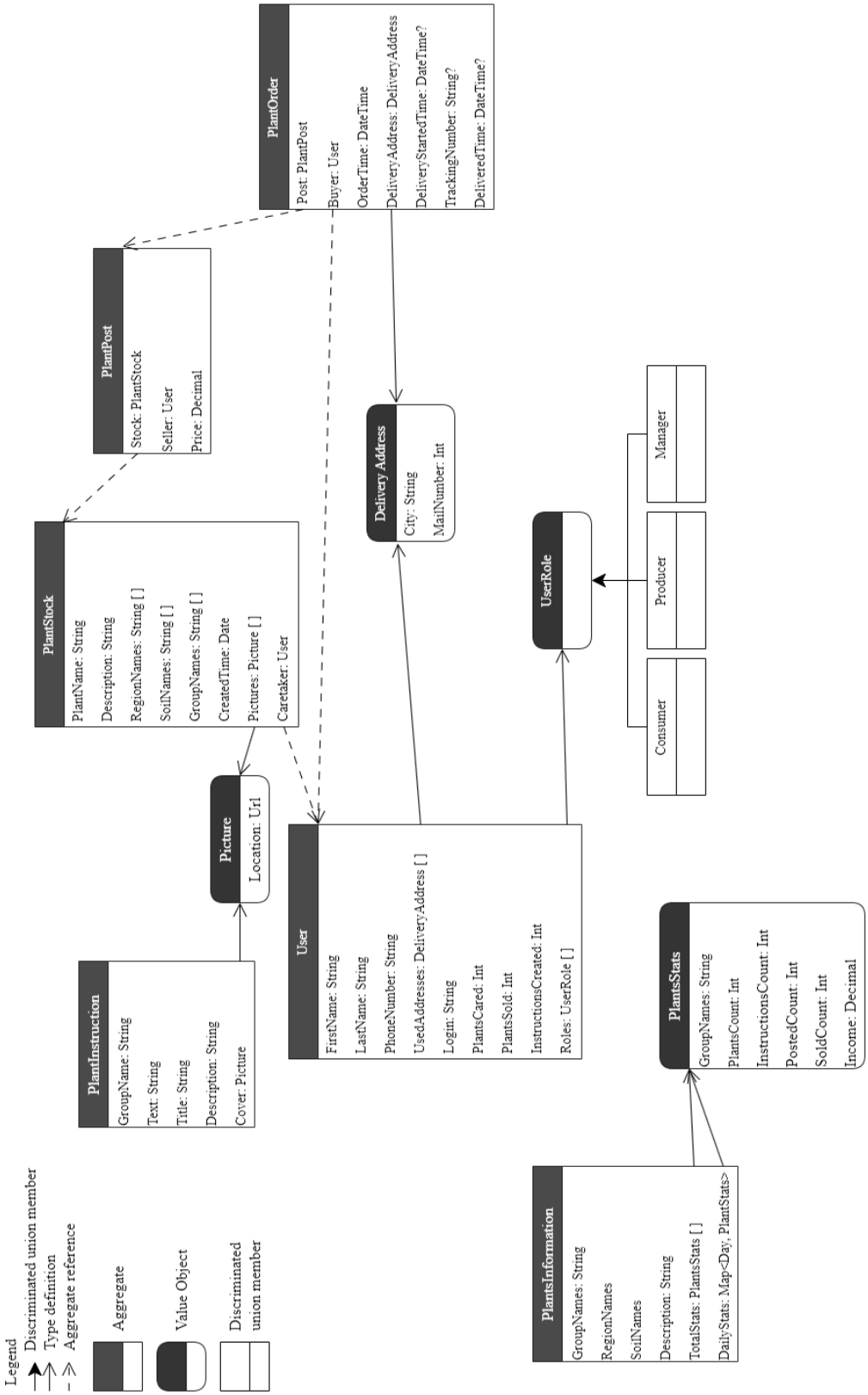

Figure A.1 – Domain Diagram

APPENDIX B

Aggregate Use Cases

PlantInstructions:

```
namespace Plants.Aggregates;

// Commands

public record CreateInstructionCommand(CommandMetadata Metadata,
InstructionModel Instruction, byte[] CoverImage) :
Command(Metadata);
public record InstructionCreatedEvent(EventMetadata Metadata,
InstructionModel Instruction, string CoverUrl, string
WriterUsername, Guid InstructionId) : Event(Metadata);

public record EditInstructionCommand(CommandMetadata Metadata,
InstructionModel Instruction, byte[] CoverImage) :
Command(Metadata);
public record InstructionEditedEvent(EventMetadata Metadata,
InstructionModel Instruction, string CoverUrl) :
Event(Metadata);

// Queries

public record SearchInstructions(PlantInstructionParams
Parameters, QueryOptions Options) :
IRequest<IEnumerable<FindInstructionsViewResultItem>>;
public record GetInstruction(Guid InstructionId) :
IRequest<GetInstructionViewResultItem?>;

public record FindInstructionsViewResultItem(Guid Id, string
Title, string Description, string CoverUrl);
public record PlantInstructionParams(string GroupName, string
Title, string Description) : ISearchParams;

// Types

public record GetInstructionViewResultItem(Guid Id, string
Title, string Description,
    string InstructionText, string CoverUrl, string
PlantGroupName);

public record InstructionModel(
    string GroupName, string Text, string Title,
    string Description);
```

PlantOrder:

```
namespace Plants.Aggregates;

// Commands
```

```
public record StartOrderDeliveryCommand(CommandMetadata
Metadata, string TrackingNumber) : Command(Metadata);
public record OrderDeliveryStartedEvent(EventMetadata Metadata,
string TrackingNumber) : Event(Metadata);

public record RejectOrderCommand(CommandMetadata Metadata) :
Command(Metadata);
public record RejectedOrderEvent(EventMetadata Metadata) :
Event(Metadata);

public record ConfirmDeliveryCommand(CommandMetadata Metadata) :
Command(Metadata);
public record DeliveryConfirmedEvent(EventMetadata Metadata,
string SellerUsername, string[] GroupNames, decimal Price) :
Event(Metadata);

// Queries

public record SearchOrders(PlantOrderParams Parameters,
QueryOptions Options) :
IRequest<IEnumerable<OrdersViewResultItem>>;

// Types
public record PlantOrderParams(bool OnlyMine) : ISearchParams;

public record OrdersViewResultItem(
    int Status, Guid PostId, string City,
    long MailNumber, string SellerName, string SellerContact,
    decimal Price, string? DeliveryTrackingNumber, Picture[]
Images,
    DateTime Ordered, DateTime? DeliveryStarted, DateTime?
Shipped)
{
    public string OrderedDate => Ordered.ToShortDateString();
    public string? DeliveryStartedDate =>
DeliveryStarted?.ToShortDateString();
    public string? ShippedDate => Shipped?.ToShortDateString();
}
```

PlantPost:

```
using Humanizer;

namespace Plants.Aggregates;

// Commands

public record RemovePostCommand(CommandMetadata Metadata) :
Command(Metadata);
public record PostRemovedEvent(EventMetadata Metadata) :
Event(Metadata);

public record OrderPostCommand(CommandMetadata Metadata,
DeliveryAddress Address) : Command(Metadata);
```

```
public record PostOrderedEvent(EventMetadata Metadata,
DeliveryAddress Address, string BuyerUsername) :
Event(Metadata);

// Queries

public record SearchPosts(PlantPostParams Parameters,
QueryOptions Options) :
IRequest<IEnumerable<PostSearchViewResultItem>>;
public record GetPost(Guid PostId) :
IRequest<PostViewResultItem?>;

// Types

public record DeliveryAddress(string City, long MailNumber);

public record PostViewResultItem(Guid Id, string PlantName,
string Description, decimal Price,
    string[] SoilNames, string[] RegionNames, string[]
GroupNames, DateTime Created,
    string SellerName, string SellerPhone, long SellerCared,
long SellerSold, long SellerInstructions,
    long CareTakerCared, long CareTakerSold, long
CareTakerInstructions, Picture[] Images
)
{
    public string CreatedHumanDate => Created.Humanize();
    public string CreatedDate => Created.ToShortDateString();
}

public record PlantPostParams(
    string? PlantName,
    decimal? LowerPrice,
    decimal? TopPrice,
    DateTime? LastDate,
    string[]? GroupNames,
    string[]? RegionNames,
    string[]? SoilNames) : ISearchParams;

public record PostSearchViewResultItem(Guid Id, string
PlantName, string Description, Picture[] Images, double Price);
```

PlantInformation:

```
namespace Plants.Aggregates;

// Queries

public record GetTotalStats :
IRequest<IEnumerable<TotalStatsViewResult>>;
public record GetFinancialStats(DateTime? From, DateTime? To) :
IRequest<IEnumerable<FinancialStatsViewResult>>;
public record GetUsedPlantSpecifications :
IRequest<PlantSpecifications>;
```

```
// Types

public record TotalStatsViewResult(string GroupName, decimal
Income, long Instructions, long Popularity);
public record FinancialStatsViewResult(decimal Income, string
GroupName, long SoldCount, long PercentSold);
public record PlantSpecifications(HashSet<string> Groups,
HashSet<string> Regions, HashSet<string> Soils);
```

PlantStock:

```
using Humanizer;

namespace Plants.Aggregates;

// Commands

public record AddToStockCommand(CommandMetadata Metadata,
PlantInformation Plant, DateTime CreatedTime, byte[][] Pictures)
: Command(Metadata);
public record StockAddedEvent(EventMetadata Metadata,
PlantInformation Plant, DateTime CreatedTime, Picture[]
Pictures, string CaretakerUsername) : Event(Metadata);

public record EditStockItemCommand(CommandMetadata Metadata,
PlantInformation Plant, byte[][] NewPictures, Guid[]
RemovedPictureIds) : Command(Metadata);
public record StockEdditedEvent(EventMetadata Metadata,
PlantInformation Plant, Picture[] NewPictures, Guid[]
RemovedPictureIds) : Event(Metadata);

public record PostStockItemCommand(CommandMetadata Metadata,
decimal Price) : Command(Metadata);
public record StockItemPostedEvent(EventMetadata Metadata,
string SellerUsername, decimal Price, string[] GroupNames) :
Event(Metadata);

// Queries

public record GetStockItems(PlantStockParams Params,
QueryOptions Options) :
IRequest<IEnumerable<StockViewResultItem>>;
public record GetStockItem(Guid StockId) :
IRequest<PlantViewResultItem?>;
public record GetPrepared(Guid StockId) :
IRequest<PreparedPostResultItem?>;

// Types

public record PlantStockParams(bool IsMine) : ISearchParams;
public record StockViewResultItem(Guid Id, string PlantName,
string Description, bool IsMine);
```

```
public record PlantInformation(
    string PlantName, string Description, string[] RegionNames,
    string[] SoilNames, string[] GroupNames
    );

public record PlantViewResultItem(string PlantName, string
Description, string[] GroupNames,
    string[] SoilNames, Picture[] Images, string[] RegionNames,
DateTime Created)
{
    public string CreatedHumanDate => Created.Humanize();
    public string CreatedDate => Created.ToShortDateString();
}


public record PreparedPostResultItem(
    Guid Id, string PlantName, string Description, string[]
SoilNames,
    string[] RegionNames, string[] GroupNames, DateTime Created,
    string SellerName, string SellerPhone, long SellerCared,
long SellerSold, long SellerInstructions,
    long CareTakerCared, long CareTakerSold, long
CareTakerInstructions, Picture[] Images)
{
    public string CreatedHumanDate => Created.Humanize();
    public string CreatedDate => Created.ToShortDateString();
}
```

User:

```
namespace Plants.Aggregates;

// Commands

public record CreateUserCommand(CommandMetadata Metadata,
UserCreationDto Data) : Command(Metadata);
public record UserCreatedEvent(EventMetadata Metadata,
UserCreationDto Data) : Event(Metadata);

public record ChangeRoleCommand(CommandMetadata Metadata,
UserRole Role) : Command(Metadata);
public record RoleChangedEvent(EventMetadata Metadata, UserRole
Role) : Event(Metadata);

public record ChangeOwnPasswordCommand(CommandMetadata Metadata,
string OldPassword, string NewPassword) : Command(Metadata);
public record ChangePasswordCommand(CommandMetadata Metadata,
string Login, string OldPassword, string NewPassword) :
Command(Metadata);
public record PasswordChangedEvent(EventMetadata Metadata) :
Event(Metadata);

// Queries
```

```
public record SearchUsers(UserSearchParams Parameters,
QueryOptions Options) :
IRequest<IEnumerable<FindUsersResultItem>>;
public record GetOwnUsedAddresses : IRequest<AddressViewResult>;

// Types

public record UserCreationDto(string FirstName, string LastName,
string PhoneNumber, string Login, string Email, string Language,
UserRole[] Roles);

public record AddressViewResult(List<DeliveryAddress>
Addresses);
public record UserSearchParams(string Name, string Phone,
UserRole[] Roles) : ISearchParams;
public record FindUsersResultItem(Guid Id, string FullName,
string Mobile, string Login, UserRole[] RoleCodes);
```

APPENDIX C

Shared frontend modules

Endpoints:

```
module Endpoints exposing (Endpoint(..), IdType(..),
endpointToUrl, getAuthed, getAuthedQuery, getImageUrl,
historyUrl, imagesDecoder, postAuthed, postAuthedQuery)

import Dict
import Http exposing (header, request)
import ImageList
import Json.Decode as D
import Main exposing (UserRole, roleToNumber)


baseUrl : String
baseUrl =
    "https://localhost:5001/"


type Endpoint
    = Login
    | StatsTotal
    | StatsFinancial
    | Search
    | Dicts
    | Post String
    | OrderPost String String Int --plantId, city, mailNumber
    | Addresses
    | NotPostedPlants
    | NotPostedPlant String
    | PreparedPlant String
    | PostPlant String Float
    | AddPlant
    | EditPlant String
    | AllOrders Bool
    | SendOrder String String
    | ReceivedOrder String
    | SearchUsers
    | AddRole String UserRole
    | RemoveRole String UserRole
    | CreateUser
    | FindInstructions
    | CreateInstruction
    | EditInstruction String
    | GetInstruction String
    | DeletePost String
    | RejectOrder String
    | ChangePassword
    | History
```

```
type IdType
    = LongId Int
    | StringId String
    | GuidId String


endpointToUrl : Endpoint -> String
endpointToUrl endpoint =
    case endpoint of
        Login ->
            baseUrl ++ "auth/login"

        StatsTotal ->
            baseUrl ++ "stats/total"

        StatsFinancial ->
            baseUrl ++ "stats/financial"

        Search ->
            baseUrl ++ "search"

        Dicts ->
            baseUrl ++ "info/dicts"

        Post plantId ->
            baseUrl ++ "post/" ++ plantId

        OrderPost plantId city mailNumber ->
            baseUrl ++ "post/" ++ plantId ++ "/order" ++
"?city=" ++ city ++ "&mailNumber=" ++ String.fromInt mailNumber

        Addresses ->
            baseUrl ++ "info/addresses"

        NotPostedPlants ->
            baseUrl ++ "plants/notposted"

        PreparedPlant plantId ->
            baseUrl ++ "plants/prepared/" ++ plantId

        PostPlant plantId price ->
            baseUrl ++ "plants/" ++ plantId ++ "/post?price=" ++
String.fromFloat price

        NotPostedPlant id ->
            baseUrl ++ "plants/notposted/" ++ id

        AddPlant ->
            baseUrl ++ "plants/add"

        EditPlant plantId ->
```

```
            baseUrl ++ "plants/" ++ plantId ++ "/edit"

        AllOrders onlyMine ->
            let
                mineStr =
                    if onlyMine then
                        "true"

                    else
                        "false"
            in
            baseUrl ++ "orders?onlyMine=" ++ mineStr

        SendOrder orderId ttn ->
            baseUrl ++ "orders/" ++ orderId ++
"/deliver?trackingNumber=" ++ ttn

        ReceivedOrder orderId ->
            baseUrl ++ "orders/" ++ orderId ++ "/delivered"

        SearchUsers ->
            baseUrl ++ "users"

        AddRole login role ->
            baseUrl ++ "users/" ++ login ++ "/add/" ++
(String.fromInt <| roleToNumber role)

        RemoveRole login role ->
            baseUrl ++ "users/" ++ login ++ "/remove/" ++
(String.fromInt <| roleToNumber role)

        CreateUser ->
            baseUrl ++ "users/create"

        FindInstructions ->
            baseUrl ++ "instructions/find"

        CreateInstruction ->
            baseUrl ++ "instructions/create"

        GetInstruction id ->
            baseUrl ++ "instructions/" ++ id

        EditInstruction id ->
            baseUrl ++ "instructions/" ++ id ++ "/edit"

        DeletePost id ->
            baseUrl ++ "post/" ++ id ++ "/delete"

        RejectOrder orderId ->
            baseUrl ++ "orders/" ++ orderId ++ "/reject"
```

```
        ChangePassword ->
            baseUrl ++ "users/changePass"

        History ->
            baseUrl ++ "eventsourcing/history"


imagesDecoder : String -> List String -> D.Decoder
ImageList.Model
imagesDecoder token at =
    let
        baseDecoder tuples =
            ImageList.fromDict <| Dict.fromList tuples
    in
    D.map baseDecoder (D.at at (D.list <| imageDecoder token))


imageDecoder : String -> D.Decoder ( String, String )
imageDecoder token =
    D.map2 Tuple.pair (D.field "id" D.string) (D.map
(getImageUrl token) <| D.field "location" D.string)


getImageUrl : String -> String -> String
getImageUrl token location =
    baseUrl ++ location ++ "?token=" ++ token


postAuthed : String -> Endpoint -> Http.Body -> Http.Expect msg
-> Maybe Float -> Cmd msg
postAuthed token endpoint body expect timeout =
    baseRequest "POST" token (endpointToUrl endpoint) body
expect timeout Nothing


getAuthed : String -> Endpoint -> Http.Expect msg -> Maybe Float
-> Cmd msg
getAuthed token endpoint expect timeout =
    baseRequest "GET" token (endpointToUrl endpoint)
Http.emptyBody expect timeout Nothing


getAuthedQuery : String -> String -> Endpoint -> Http.Expect msg
-> Maybe Float -> Cmd msg
getAuthedQuery query token endpoint expect timeout =
    baseRequest "GET" token (endpointToUrl endpoint ++ query)
Http.emptyBody expect timeout Nothing


postAuthedQuery : String -> String -> Endpoint -> Http.Expect
msg -> Maybe Float -> Cmd msg
postAuthedQuery query token endpoint expect timeout =
```

```
    baseRequest "POST" token (endpointToUrl endpoint ++ query)
Http.emptyBody expect timeout Nothing


baseRequest : String -> String -> String -> Http.Body ->
Http.Expect msg -> Maybe Float -> Maybe String -> Cmd msg
baseRequest method token url body expect timeout tracker =
    request
        { method = method
        , headers = [ header "Authorization" <| "Bearer " ++
token ]
        , url = url
        , body = body
        , expect = expect
        , timeout = timeout
        , tracker = tracker
        }


historyUrl name id =
    "/history/" ++ name ++ "/" ++ id
```

Main:

```
port module Main exposing (..)

import Bootstrap.Accordion as Accordion
import Bootstrap.Modal as Modal
import Browser
import Html exposing (Html)
import Json.Decode as D
import Json.Decode.Pipeline exposing (hardcoded, required)
import Task
import Utils exposing (Notification, SubmittedResult(..),
decodeNotificationPair, intersect)


notifyCmd : SubmittedResult -> Cmd (MsgBase msg)
notifyCmd result =
    case result of
        SubmittedSuccess _ command ->
            Task.perform (\_ -> NotificationStarted <|
Notification command True) (Task.succeed True)

        SubmittedFail a ->
            Cmd.none


type UserRole
    = Consumer
    | Producer
```

```
    | Manager


allRoles =
    [ Producer, Consumer, Manager ]


isAdmin auth =
    List.any (\r -> r == Manager) auth.roles


rolesDecoder : D.Decoder (List Int) -> D.Decoder (List UserRole)
rolesDecoder idsDecoder =
    D.map convertRoles idsDecoder


convertRoles : List Int -> List UserRole
convertRoles roleIds =
    List.map convertRole roleIds


convertRole : Int -> UserRole
convertRole roleId =
    case roleId of
        1 ->
            Consumer

        2 ->
            Producer

        3 ->
            Manager

        _ ->
            Consumer


roleToNumber : UserRole -> Int
roleToNumber role =
    case role of
        Consumer ->
            1

        Producer ->
            2

        Manager ->
            3


type alias AuthResponse =
    { token : String
```

```
    , roles : List UserRole
    , username : String
    , userId : String
    , notifications : List ( Notification, Bool )
    , notificationsModal : Modal.Visibility
    , notificationsAccordion : Accordion.State
    }


type alias ApplicationConfig model msg =
    { init : Maybe AuthResponse -> D.Value -> ( model, Cmd msg )
    , view : model -> Html msg
    , update : msg -> model -> ( model, Cmd msg )
    , subscriptions : model -> Sub msg
    }


baseApplication : ApplicationConfig model msg -> Program D.Value
model msg
baseApplication config =
    Browser.element
        { init = mainInit config.init
        , view = config.view
        , update = config.update
        , subscriptions = config.subscriptions
        }


mainInit : (Maybe AuthResponse -> D.Value -> ( model, Cmd msg ))
-> D.Value -> ( model, Cmd msg )
mainInit initFunc flags =
    let
        authResp =
            case D.decodeValue decodeFlags flags of
                Ok res ->
                    Just res

                Err _ ->
                    Nothing
    in
    initFunc authResp flags


convertRolesStr roleIds =
    List.map convertRoleStr roleIds


roleToStr : UserRole -> String
roleToStr role =
    case role of
        Consumer ->
            "Consumer"
```

```
        Producer ->
            "Producer"

        Manager ->
            "Manager"


convertRoleStr : String -> UserRole
convertRoleStr roleId =
    case roleId of
        "Consumer" ->
            Consumer

        "Producer" ->
            Producer

        "Manager" ->
            Manager

        _ ->
            Consumer


decodeFlags : D.Decoder AuthResponse
decodeFlags =
    D.succeed AuthResponse
        |> required "token" D.string
        |> required "roles" (D.list D.string |> D.map
convertRolesStr)
        |> required "username" D.string
        |> required "userId" D.string
        |> required "notifications" (D.list
decodeNotificationPair)
        |> hardcoded Modal.hidden
        |> hardcoded Accordion.initialState


type ModelBase model
    = Unauthorized
    | NotLoggedIn
    | Authorized AuthResponse model


port navigate : String -> Cmd msg


port goBack : () -> Cmd msg


port notificationReceived : (Notification -> msg) -> Sub msg
```

```
port dismissNotification : Notification -> Cmd msg


port resizeAccordions : () -> Cmd msg


type MsgBase msg
    = Navigate String
    | GoBack
    | Main msg
    | NotificationStarted Notification
    | NotificationReceived Notification
    | NotificationDismissed Notification
    | AllNotificationsDismissed
    | CloseNotificationsModal
    | ShowNotificationsModal
    | AnimateNotificationsModal Modal.Visibility
    | NotificationsAccordion Accordion.State


subscriptionBase : ModelBase model -> Sub (MsgBase msg) -> Sub
(MsgBase msg)
subscriptionBase mod baseSub =
    let
        notifications =
            case mod of
                Authorized auth _ ->
                    [ Modal.subscriptions
auth.notificationsModal AnimateNotificationsModal
                    , Accordion.subscriptions
auth.notificationsAccordion NotificationsAccordion
                    ]

                _ ->
                    []
    in
    ([ notificationReceived NotificationReceived, baseSub ] ++
notifications) |> Sub.batch


initBase : List UserRole -> model -> (AuthResponse -> Cmd msg) -
> Maybe AuthResponse -> ( ModelBase model, Cmd msg )
initBase requiredRoles initialModel initialCmd response =
    case response of
        Just resp ->
            if intersect requiredRoles resp.roles then
                ( Authorized resp initialModel, initialCmd resp
)

            else
                ( Unauthorized, Cmd.none )
```

```
        Nothing ->
            ( NotLoggedIn, Cmd.none )


updateBase : (msg -> ModelBase model -> ( ModelBase model, Cmd
(MsgBase msg) )) -> MsgBase msg -> ModelBase model -> (
ModelBase model, Cmd (MsgBase msg) )
updateBase updateFunc message model =
    case message of
        Navigate location ->
            ( model, navigate location )

        GoBack ->
            ( model, goBack () )

        Main main ->
            updateFunc main model

        NotificationStarted notification ->
            case model of
                Authorized auth page ->
                    let
                        updateNotifications =
                            if List.any (\( n, _ ) ->
n.command.id == notification.command.id) auth.notifications then
                                auth.notifications

                            else
                                [ ( notification, False ) ] ++
auth.notifications
                    in
                    ( Authorized { auth | notifications =
updateNotifications } page, resizeAccordions () )

                _ ->
                    ( model, Cmd.none )

        NotificationDismissed notification ->
            case model of
                Authorized auth page ->
                    ( Authorized { auth | notifications =
List.filter (\( n, _ ) -> n.command.id /=
notification.command.id) auth.notifications } page, [
resizeAccordions (), dismissNotification notification ] |>
Cmd.batch )

                _ ->
                    ( model, Cmd.none )

        AllNotificationsDismissed ->
            case model of
```

```
                Authorized auth page ->
                    ( Authorized
                        { auth
                            | notifications = []
                        }
                        page
                    , List.map (\( n, _ ) -> dismissNotification
n) auth.notifications |> Cmd.batch
                    )

                _ ->
                    ( model, Cmd.none )

        NotificationReceived notification ->
            case model of
                Authorized auth page ->
                    let
                        mapNotification not succ =
                            if not.command.id ==
notification.command.id then
                                ( not, True )

                            else
                                ( not, succ )
                    in
                    ( Authorized { auth | notifications =
List.map (\( n, s ) -> mapNotification n s) auth.notifications }
page, resizeAccordions () )

                _ ->
                    ( model, Cmd.none )

        CloseNotificationsModal ->
            case model of
                Authorized auth page ->
                    ( Authorized { auth | notificationsModal =
Modal.hidden } page, Cmd.none )

                _ ->
                    ( model, Cmd.none )

        ShowNotificationsModal ->
            case model of
                Authorized auth page ->
                    ( Authorized { auth | notificationsModal =
Modal.shown } page, Cmd.none )

                _ ->
                    ( model, Cmd.none )

        AnimateNotificationsModal visibility ->
            case model of
```

```
                Authorized auth page ->
                    ( Authorized { auth | notificationsModal =
visibility } page, Cmd.none )

                _ ->
                    ( model, Cmd.none )

        NotificationsAccordion state ->
            case model of
                Authorized auth page ->
                    ( Authorized { auth | notificationsAccordion
= state } page, Cmd.none )

                _ ->
                    ( model, Cmd.none )


mapCmd : Cmd a -> Cmd (MsgBase a)
mapCmd msg =
    Cmd.map Main msg


mapSub : Sub a -> Sub (MsgBase a)
mapSub msg =
    Sub.map Main msg
```

APPENDIX D

Page Navigation

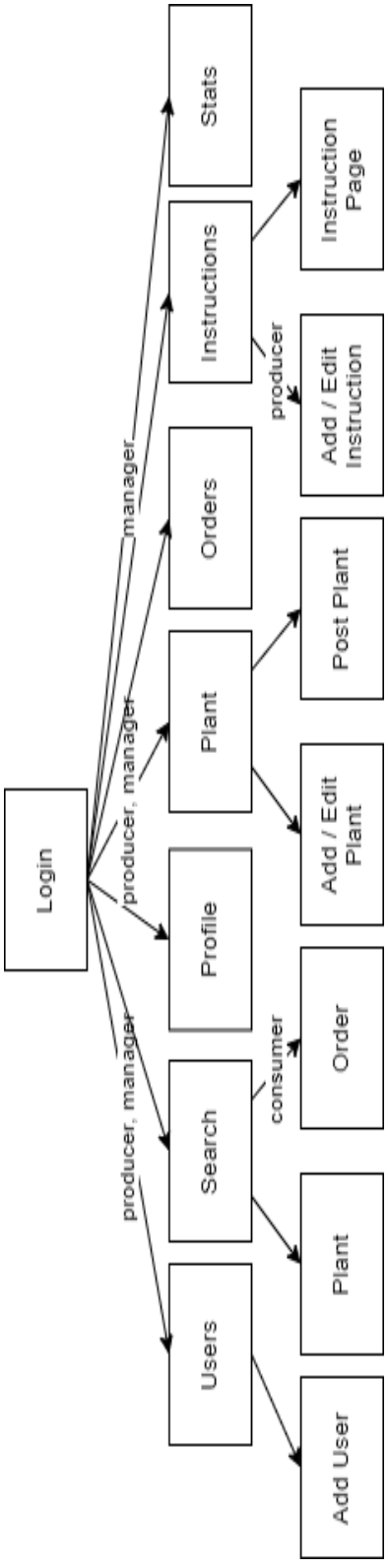

Figure D.1 – Page navigation diagram