STATE UNIVERSITY OF INTELLIGENT TECHNOLOGIES AND TELECOMMUNICATIONS

Faculty of Information Technology and Cybersecurity

Software Engineering Department

Course Project

on the course «Organization of databases and knowledge»

on the topic **AUTOMATION OF PLANT SELLING BUSINESS PROCESS**

Completed by:

3rd-year student, group SE 3.2.01TE

Yaroslav Holota

Supervisor ScD. Prof. E.V. Malakhov

National scale _____

Number of points ___ ECTS grade ___

Odesa – 2022

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.

INTRODUCTION

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

In many cases the domain of business and the problems it is intending on providing solution for may be unnecessarily complicated by the non-existence of standardized business procedure or management structure.

The tasks of the business that is to be automated are as follows:

1. Care for the plants in preparation for their sale.
2. Put plants for sale and organize delivery through the postal service of convenience.
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 tasks of the business, the tasks of this project are as follows:

1. Analyze business domain and create a logical framework of this application, specified roles of actors and map business entities.
2. Select fitting software components.
3. Create SQL statements that would be used to automate business process.
4. Organize application architecture.
5. Create User Interface.
6. Organize testing strategy for the application.

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 their tasks as well as correspondence of input and output data related to them.

Table 1.1 – User Tasks

Number	Task	Explanation	Input	Output
	Consumer, Producer, Manager			
Z1	Access the system.		Login Password	Session
Z2	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

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 Group Soil 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 experience.		Addresses: City Location
B3	Confirm order to be delivered.		Order Id	
	Producer, Manager			
C1	Find plants that are being prepared for post.		Limit to Cared	Plants: Plant Name Plant Description Is cared flag

Continuation of Table 1.1

C2	Edit plant information.		Plant Id New Plant	New Plant
C3	Create plant.		Plant Name Plant Description Plant Regions Plant Soil Plant Group 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
C7	Find users.	This allows managers to manage producers and producers to manage consumers.	Name Phone Number	Users: Name Phone Number Roles

Continuation of Table 1.1

C8	Invite users.		Login Roles Email Name Phone Number	User created and email with temporary password send.
C9	Update user roles.	This is limited to adding or removing roles that have lesser priority that the current user.	Login Role	
C10	Remove post.	For producers this is limited to post that they have created.	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
	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

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.

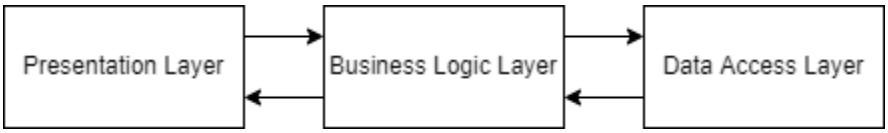

Figure 2.1 – Thee-tier architecture

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

Clean architecture 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. 2.2.

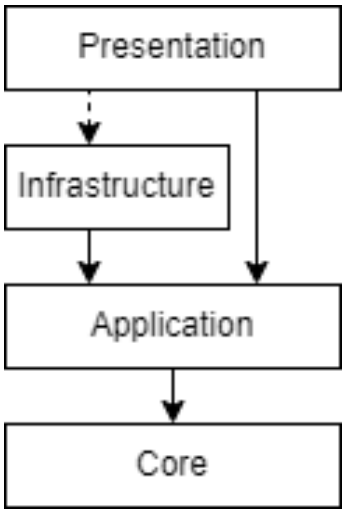

Figure 2.2 – Clean architecture

Here, core represents the base of application and contains system-wide concerns and business entities representation, application represents business logic, infrastructure represents external dependencies and presentation provides a medium for information transfer.

Clean architecture has been chosen to allow for separation of business concerns and the actual infrastructure.

2.3 Presentation layer architecture

As a pattern for development of presentation layer MVU 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.

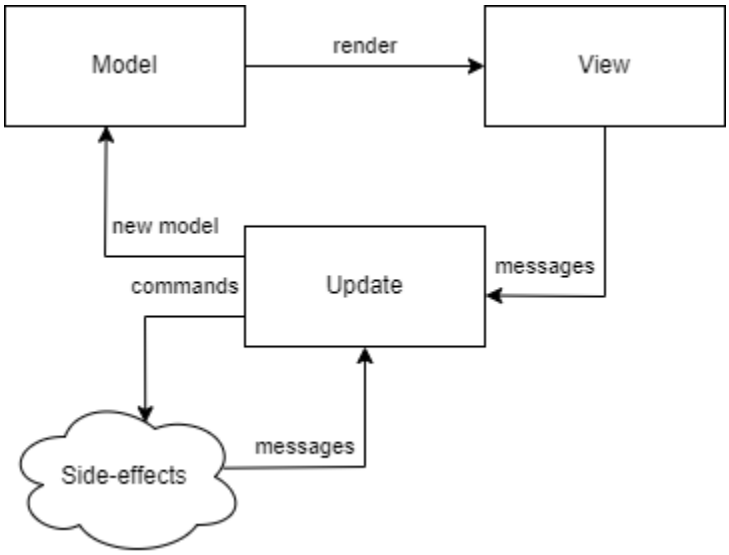

Figure 2.3 – MVU pattern

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

3 MODELING OF THE DOMAIN OF INFORMATIONAL SYSTEM

To model the domain of informational system we should defined business entities and their attributes. In relational databases concept of business entity maps to relation and concept of attribute maps to the notion of constraint. So, relations and their constraints are presented in table 3.1.

Table 3.1 – Relations and their constraints

Field	Description	Constraints
“plant_group” relation		
id	Identifier	PRIMARY KEY (PK)
group_name	Name of the group	NOT NULL
“plant_region” relation		
id	Identifier	PK
region_name	Name of the region	NOT NULL
“plant_soil” relation		
id	Identifier	PK
soil_name	Name of the soil	NOT NULL
“person” relation		
id	Identifier	PK
first_name	First name of person	NOT NULL
last_name	Last name of person	NOT NULL
phone_number	Contact address of person	NOT NULL
“person_to_login” relation		
person_id	Identifier of person	PK, FK
login	Database login of person	NOT NULL, lowercase, exists as database role name
“plant” relation		
id	Identifier	PK
group_id	Group Identifier	FOREIGN KEY (FK), NOT NULL
soil_id	Soil Identifier	FK, NOT NULL

Continuation of Table 3.1

Field	Description	Constraints
care_taker_id	Identifier of employee that is assigned with caring for plant	FK, NOT NULL
plant_name	Name of the plant	NOT NULL
description	Description of the plant	NOT NULL
created	Date of plant being physically created	NOT NULL
“plant_to_image” relation		
relation_id	Identifier	PK
plant_id	Plant identifier	FK, NOT NULL
image	Image of the plant	NOT NULL
“plant_to_region” relation		
id	Identifier	PK
plant_id	Plant Identifier	FK, NOT NULL, UNIQUE (plant_id, plant_region_id)
plant_region_id	Region Identifier	FK, NOT NULL, UNIQUE (plant_id, plant_region_id)
“delivery_address” relation		
id	Identifier	PK
city	Name of the city	NOT NULL, UNIQUE (city, nova_poshta_number)
nova_poshta_number	Number of postal service	NOT NULL, UNIQUE (city, nova_poshta_number)

Continuation of table 3.1

Field	Description	Constraints
“person_to_delivery” relation		
id	Identifier	PK
person_id	Person Identifier	FK, NOT NULL
delivery_address_id	Address Identifier	FK, NOT NULL
“plant_post” relation		
plant_id	Identifier of plant	PK, FK
seller_id	Identifier of seller	FK, NOT NULL
price	Price of plant	NOT NULL, >= 0
created	Date of post creation	NOT NULL
“plant_order” relation		
post_id	Identifier of post	PK, FK
customer_id	Identifier of customer	FK, NOT NULL
delivery_address_id	Identifier of address	FK, NOT NULL
created	Time of post being ordered	NOT NULL
“plant_delivery” relation		
order_id	Identifier of order	PK, FK
delivery_tracking_number	Tracking number for this delivery	NOT NULL
created	Time of delivery starting	NOT NULL
“plant_shipment” relation		
delivery_id	Identifier of delivery	PK, FK
shipped	Time of delivery being confirmed	NOT NULL

Continuation of table 3.1

Field	Description	Constraints
“plant_caring_instruction” relation		
id	Identifier	PK
instruction_text	Main text of the instruction, includes formatting	NOT NULL
posted_by_id	Identifier of producer that created this instruction	FK, NOT NULL
plant_group_id	Identifier of group	FK, NOT NULL
title	Title of instruction	NOT NULL
description	Description of instruction	NOT NULL
“instruction_to_cover” relation		
instruction_id	Identifier of instruction	PK, FK
image	Image of cover	NOT NULL

Additional constraints include:

Relation	Constraints
plant_post	Cannot be deleted by anyone except a manager or the producer that created it.
plant_order	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.
plant	Cannot be updated if post record exists that references it. Age of the plant cannot be edited under any condition.
plant_delivery	Can only be created by the customer that created underlying order.

There are three types of relationships between relations in this database:

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

Relationship “One-to-one” is formalized by adding primary key of main relation to dependent relation as primary and foreign key.

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

- plant and plant_post
- plant_post and plant_order
- plant_order and plant_delivery
- plant_delivery and plant_shipment
- person and person_to_login
- plant_caring_instruction and instruction_to_cover

Here, first relation is main and second is dependent.

Relationship “One-to-many” is formalized by adding primary key of main relation to dependent relation as foreign key.

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

- plant_group and plant
- plant_soil and plant
- person and plant
- plant and plant_to_image
- person and plant_order
- delivery_address and plant_order
- person and plant_caring_instruction
- plant_group and plant_caring_instruction

Here, first relation is main and second is dependent.

Relationship “Many-to-many” is formalized by creating connecting relation that has primary keys of both tables as foreign keys.

Relationship “Many-to-many” exists between following tables:

- plant_region and plant through plant_to_region
- person and delivery_address through person_to_delivery

Here, both sides of relationship go first and connecting relation after them.

The database is in the Boyce-Codd normal form. Entity relationship diagram of the database can be found in Appendix D.

4 USED TECHNOLOGIES AND SOFTWARE

The informational system is composed of three parts – Data Access layer build with PostgreSQL, Application Layer build with ASP.NET Core framework and Presentation Layer build with Elm, React and Bootstrap 5.

Out of many DBMS possibilities the PostgreSQL has been selected for following reasons:

- Complete implementation for relational database standard.
- Complex and throughout role and group access system.
- Advanced support for stored procedures using plpgSQL language.
- Support for byte array storage for storing large images.
- Support for local views.
- Throughout documentation.
- Actively supported and developed.

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 actual database authorization would apply.

5.1 Backend architecture

The communication between frontend and backend would be organized through the REST-full API, whose diagram is presented in fig. 5.1.

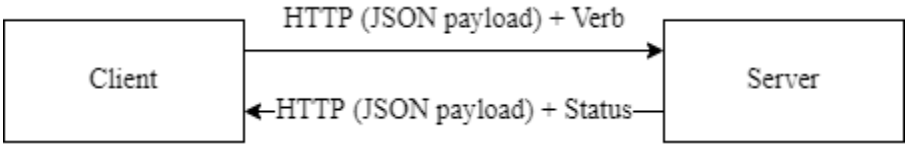

Figure 5.1 - REST API diagram

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.

The application layer of backend would have separate notion of request and request handler. Request contains all of the information necessary to process the request and defines expected result. Request handler contains all the logic needed to process the request. The diagram of such architecture can be seen on fig. 5.2 and a class diagram of financial stats request can be seen on the fig. 5.3.

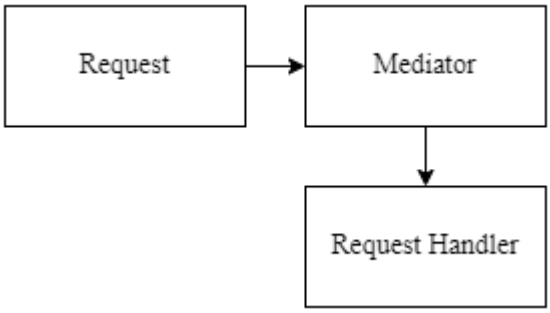

Figure 5.2 – Backend Application layer architecture

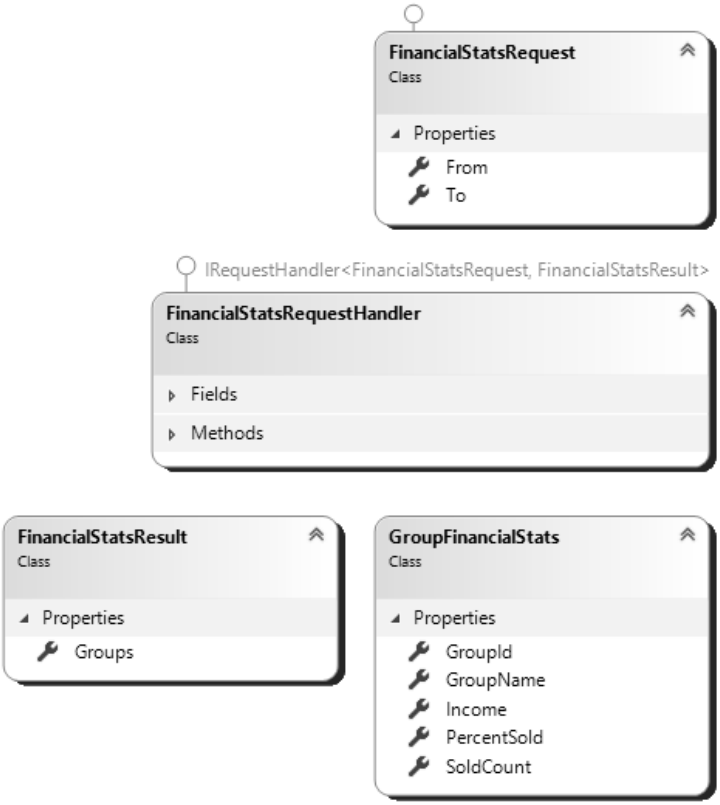

Figure 5.3 – Financial Request class diagram

The interaction with the database layer is performed through the infrastructure layer [3]. However, the actual interaction is performed though the usage of application layer contract that is being fulfilled by the infrastructure layer contract at the runtime through the usage of dependency injection [1]. The diagram of such interaction may be seen on the fig. 5.4 and the class diagram for order-related interactions of fig. 5.5.

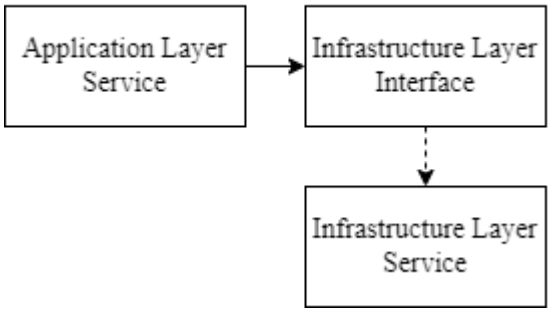

Figure 5.4 – Dependency Injection diagram

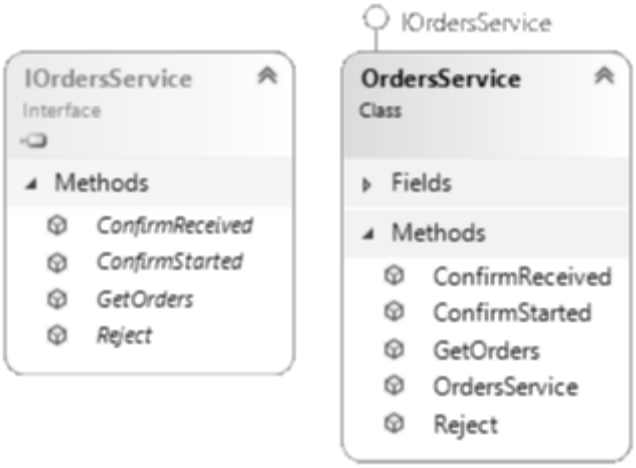

Figure 5.5 – Orders service class diagram

5.2 Frontend architecture

The frontend application is structured as many MVU [5] 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.

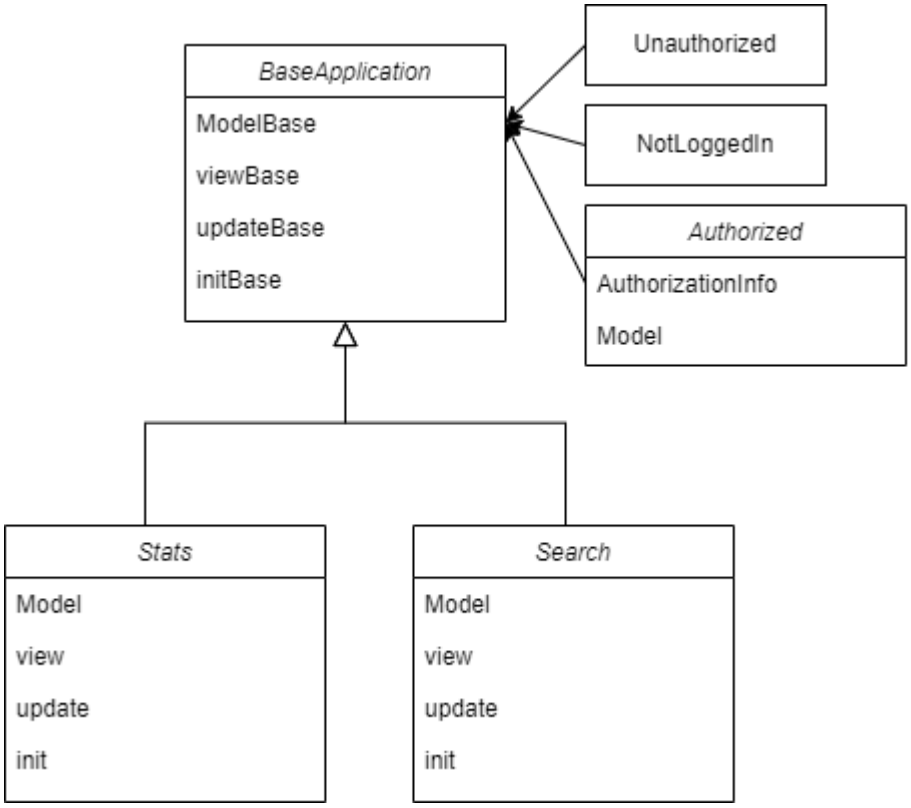

Figure 5.6 – Frontend architecture

6 DATABASE CREATION

In this section the database objects and permissions that are being granted for them are provided with explanation of their content. SQL statements for table creation can be found in Appendix A. SQL statements for other objects can be found in Appendix B. The grants of access to above-mentioned database objects can be found in Appendix C.

6.1 Object definitions

Types:

1) UserRoles – existing roles: consumer, producer, manager.

Tables:

1) person – personal information of any type of user.
2) person_to_login – logins of users.
3) delivery_address – city and the postal service number to which deliveries can be made.
4) person_to_delivery – connection table between person and delivery_address that contains addresses used for delivery by this person.
5) plant_group – groups of plants.
6) plant_region – regions of plants.
7) plant_soil – soils of plants.
8) plant_to_image – images of plants.
9) plant_to_region – connection table between plant and plant_region.
10) plant – main plant information.
11) plant_post – information about price of posted plant.
12) plant_order – information about address for delivery of order and the client.
13) plant_delivery – tracking number for delivery.
14) plant_shipment – date of completion of plant transfer.
15) plant_caring_instruction – instruction for caring for plant of specific group.
16) instruction_to_cover – cover images for instruction.

Views:

1) current_user_addresses – addresses that logged in user have used for ordering
2) current_user_orders – orders created by current user
3) current_user_roles – roles of connected user.
4) dicts_v – available values for plant groups, soils and regions
5) instruction_v – instructions with related entities
6) person_creds_v – employee to their experience in selling, caring and instruction producing.
7) plant_orders_v – combined information about order, delivery and shipment.
8) plant_post_v – expanded information for posts.
9) plant_search_v – search information for plants.
10) plant_stats_v – aggregation for various stats for plant group.
11) plants_v –plants that have not been posted.
12) prepared_for_post_v – plants that can be posted with posted specific information of current user.
13) user_to_roles – roles assigned to users.

Triggers:

1) plant_no_update_posted – verifies that plant that is to be updated have not been posted yet. Gets executed before update of plant.
2) person_check_login – verifies that login of person exists in the database. Gets executed before insert into plant_to_login.
3) delete_only_creator_or_manager – verifies that only manager or original poster can remove post or order. Gets executed before delete of plant_order and plant_post.
4) order_store_user_address – connects used address for order to the person making this order. Gets executed after insert of plant_order.
5) set_current_user_id_care_taker – sets id of employee on creation of plant. Gets executed before insert on plant.

6) set_current_user_id_instruction – sets id of employee on creation of instruction. Gets executed before insert on plant_caring_instruction.
7) set_current_user_id_seller – sets id of employee on creation of post. Gets executed before insert on plant_post.
8) set_current_user_id_order – sets id of customer on ordering of post. Gets executed before insert on plant_order.

Functions:

1) create_instruction – creates and instruction.
2) create_plant – creates a plant.
3) get_financial – financial statistics for specified time period partitioned by plant group.
4) place_order – creates an order and a delivery address if specified does not exist.
5) post_plant – posts a plant verifying creation parameters. It should not be used as the only source of truth, but as a backup for providing meaning-full response codes.
6) search_instructions – finds instructions matching specified parameters.
7) search_plant - finds instructions matching specified parameters.
8) search_users - finds users matching specified parameters.

Utility functions:

1) array_length_no_nulls – gets number of items in an array, excluding NULL values.
2) get_role_priority – gets numeric indicator of level of access of provided role.
3) parse_role – gets UserRoles type value of system role.
4) current_user_can_create – check for access to grant provided role.
5) current_user_can_create_all – check for access to grant all provided roles.
6) get_current_user_id – finds person id of connected user, if none are found - -1 is returned.

7) get_current_user_id_throw() – finds person id of connected user, if none are found – exception is thrown.

Procedures:

1) confirm_received – confirms successful delivery of an order, verifying that only user that made an order can confirm it.
2) edit_instruction – edits an instruction.
3) edit_plant – edits a plant.
4) add_user_to_group – adds user to group verifying that current user has rights to do so.
5) remove_user_from_group – removes user from group verifying access to do so.
6) create_user_login – created login for user.
7) create_user – creates login, person tuple and links them.

6.2 Accesses to database objects

User accesses can be found in table 6.1. The absence of record means that this record is being called through security definer option.

Table 6.1 - Access to tables

Table	User Role		
	Consumer	Producer	Manager
plant_post		SD	SD
plant_order		SD	SD
plant_delivery		I	I
instruction_to_cover	S	S	S
plant_to_image	S	S	S

Table 6.2 - Access to procedures

Procedure	User Role		
	Consumer	Producer	Manager
confirm_received	Execute		
edit_instruction		Execute	Execute

Continuation of Table 6.2

edit_plant		Execute	Execute
create_user		Execute	Execute
add_user_to_group		Execute	Execute
remove_user_from_group			Execute

Table 6.3 - Access to functions

Function	User Role		
	Consumer	Producer	Manager
create_instruction		Execute	Execute
create_plant		Execute	Execute
get_financial			Execute
place_order	Execute		
post_plant		Execute	Execute
search_instructions	Execute	Execute	Execute
search_plant	Execute	Execute	Execute
search_user		Execute	Execute
	Utility		
array_length_no_nulls	Execute	Execute	Execute
get_current_user_id	Execute	Execute	Execute
get_current_user_id_throw	Execute	Execute	Execute
parse_role	Execute	Execute	Execute

Table 6.4 - Access to views

View	User Role		
	Consumer	Producer	Manager
dicts_v	S	S	S
current_user_addresses	S	S	S
instruction_v	S	S	S
current_user_orders	S		
plant_orders_v		S	S

Continuation of Table 6.4

plants_v		S	S
prepared_for_post_v		S	S
plant_post_v	S	S	S
plant_stats_v			S
current_user_roles	S	S	S

Here, annotations are as follows:

- S: Select, projection of data from the table, view
- I: Insert, additional of tuple to table
- D: Delete, removal of tuple from table
- Execute: Execution of procedure or function

7 SQL QUERIES FOR USER TASKS

Source code for all mentioned tables can be found in Appendix A. All other database objects can be found in the Appendix B.

1) Tasks A3, B2, C4 and D1 are solved by projecting from view and can be invoked with the following query template

```
SELECT [columns] FROM [view] WHERE [predicate];
```

With following values:

- A3 – columns are all, view name is plant_post_v and predicate is id = @postId.
- B2 – columns are all, view name is current_user_addresses.
- C1 – columns are id, plant_name, description, is_mine and view name is plants_v.
- C4 – columns are all, view name is prepared_for_post_v, predicate is id = @plantId.
- D1 – columns are all, view name is plant_stats_v.

2) Tasks A1, A2, B1, C3, C5, C7 and D2 are solved with the database function that can be invoked with the following query template

```
SELECT * FROM [function_name]([parameters]);
```

With following values:

- A1 – function name is search_plant and parameters are @plantName, @lowerPrice, @topPrice, @lastDate, @groupIds, @soilIds, @regionIds.
- A2 – function name is search_instructions and parameters are @GroupId, @Title, @Description.
- B1 – function name is place_order and parameters are @postId, @city, @postNumber.
- C3 – function name is create_plant and parameters are @Name, @Description, @Regions, @SoilId, @GroupId, @Created, @Pictures.
- C5 – function name is post_plant and parameters are @plantId, @price.
- C6 – function name is create_instruction and parameters are @GroupId, @Text, @Title, @Description, @CoverImage.

- C7 – function name is search_users and parameters are @FullName, @Contact, @Roles.
- D2 – function name is get_financial and parameters are @from, @to.

3) Tasks B3, C2, C8, C9 and C11 are solved with the procedure that can be invoked with the following query template.

```
CALL [procedure_name]([parameters]);
```

- B3 – procedure name is confirm_received and parameters are @deliveryId.
- C2 – procedure name is edit_plant and parameters are @PlantId, @Name, @Description, @Regions, @SoilId, @GroupId, @RemovedImages, @NewImages.
- C8 – procedure name is create_user and parameters are @Login, @Password, @Roles, @FirstName, @LastName, @PhoneNumber.
- C9 – procedure names are add_user_to_group, remove_user_from_group and parameters for both are @login, @role.
- C11 – procedure name is edit_instruction and parameters are @InstructionId, @GroupId, @Text, @Title, @Description, @CoverImage.

4) Tasks C10 and C12 use trigger [2] that enforce removal by only poster or manager. For C10 the trigger is order_prevent_unlawfull_delete and for C12 the trigger is post_prevent_unlawfull_delete. The both use underlying delete_only_creator_or_manager trigger function and are solved with the following query template

```
DELETE FROM [table_name] WHERE [predicate]
```

- C10 – table name is plant_post and predicate is plant_id = @postId.
- C12 – table name is plant_order and predicate is post_id = @orderId.

5) Task Z2 is solved with the alter role [4] though the following query

```
ALTER ROLE session_user WITH ENCRYPTED password 'NewPassword';
```

6) Task C13 is solved with the following query

```
INSERT INTO plant_delivery(order_id,
delivery_tracking_number) VALUES(@orderId, @trackingNumber);
```

7) Tasks B1, B2, C2, C3, C5 and C6 additionally require the usage of trigger. A trigger has a name, its invocation conditions and its purpose.

- B1 – trigger name is order_set_customer, it gets invoked before insert on plant_order and its purpose is setting current user as the one that ordered plant.
- B2 – trigger name is order_store_used_address, it gets invoked after insert on plant_order and its purpose is saving addresses used by the customer.
- C2 – trigger name plant_prevent_update_of_posted, it gets invoked before update of plant and its purpose is to prevent update of plants that have already been published.
- C3 – trigger name is plant_set_poster, it gets invoked before insert on plant and its purpose is marking created plant's caretaker as current user.
- C5 – trigger name is post_set_poster, it gets invoked before insert on plant _post and its purpose is marking created post's seller as current user.
- C6 – trigger name is instruction_set_poster, it gets invoked before insert on plant_caring_instruction and its purpose is marking instruction's author as current user.

8 APPLICATION IMPLEMENTATION

8.1 Backend

Application layer interfaces for interface layer services can be seen on listing 8.1.

```
public interface IPlantsService
    {
        Task<IEnumerable<PlantResultItem>> GetNotPosted();
        Task<PreparedPostResultItem?> GetPrepared(int plantId);
        Task<CreatePostResult> Post(int plantId, decimal price);
        Task<AddPlantResult> Create(string Name, string
Description,
            int[] Regions, int SoilId,
            int GroupId, DateTime Created,
            byte[][] Pictures);

        Task Edit(int PlantId, string Name, string Description,
                int[] Regions, int SoilId,
                int GroupId, int[] RemovedImages, byte[][]
NewImages);
        Task<PlantResultDto?> GetBy(int id);
    }

public interface IOrdersService
    {
        Task<IEnumerable<OrdersResultItem>> GetOrders(bool
onlyMine);
        Task ConfirmStarted(int orderId, string trackingNumber);
        Task ConfirmReceived(int deliveryId);
        Task<RejectOrderResult> Reject(int orderId);
    }
```

Listing 8.1 – Infrastructure interfaces

Command and the command handler related to it for add plant feature can be seen on listing 8.2

```
  public record AddPlantCommand(string Name, string Description,
int[] Regions, int SoilId, int GroupId, DateTime Created,
byte[][] Pictures) : IRequest<AddPlantResult>;
    public record AddPlantResult(int Id);

public class AddPlantCommandHandler :
IRequestHandler<AddPlantCommand, AddPlantResult>
    {
        private readonly IPlantsService _plants;
```

```
        public AddPlantCommandHandler(IPlantsService plants)
        {
            _plants = plants;
        }

        public Task<AddPlantResult> Handle(AddPlantCommand
request, CancellationToken cancellationToken)
        {
            return _plants.Create(request.Name,
request.Description, request.Regions, request.SoilId,
request.GroupId, request.Created, request.Pictures);
        }
    }
```

Listing 8.2 – Command and Command Handler for Add Plant feature

An implementation for some infrastructure service can be seen on listing 8.3.

```
public class PlantsService : IPlantsService
    {
        public async Task<AddPlantResult> Create(string Name,
string Description, int[] Regions,
            int SoilId, int GroupId, DateTime Created, byte[][]
Pictures)
        {
            var ctx = _ctxFactory.CreateDbContext();
            await using (ctx)
            {
                await using (var connection =
ctx.Database.GetDbConnection())
                {
                    string sql = "SELECT create_plant(@Name,
@Description, @Regions, @SoilId, @GroupId, @Created,
@Pictures);";
                    var p = new
                    {
                        Name,
                        Description,
                        Regions,
                        SoilId,
                        GroupId,
                        Created,
                        Pictures
                    };
                    var res = await
connection.QueryAsync<int>(sql, p);
                    var first = res.FirstOrDefault();
                    return new AddPlantResult(first);
                }
            }
        }
    }
```

Listing 8.3 – Plants service implementation

8.2 Frontend

The base functions for creating an MVU application can be seen in the listing 8.4.

```
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 8.4 – base initialization, view functions and model structure

The definition of base application can be seen on listing 8.5.

```
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 8.5 – base application

The router of routing single page application can be seen on listing 8.6.

```
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 8.6 – application router

All of the pages are applications that use base application template, as shown in the listing 8.7 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 8.7 – login page components

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

9 USER GUIDE WITH ILLUSTRATIONS

9.1 Consumer

The initial page of the application is the login page. Its illustration can be seen on fig. 9.1. 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.

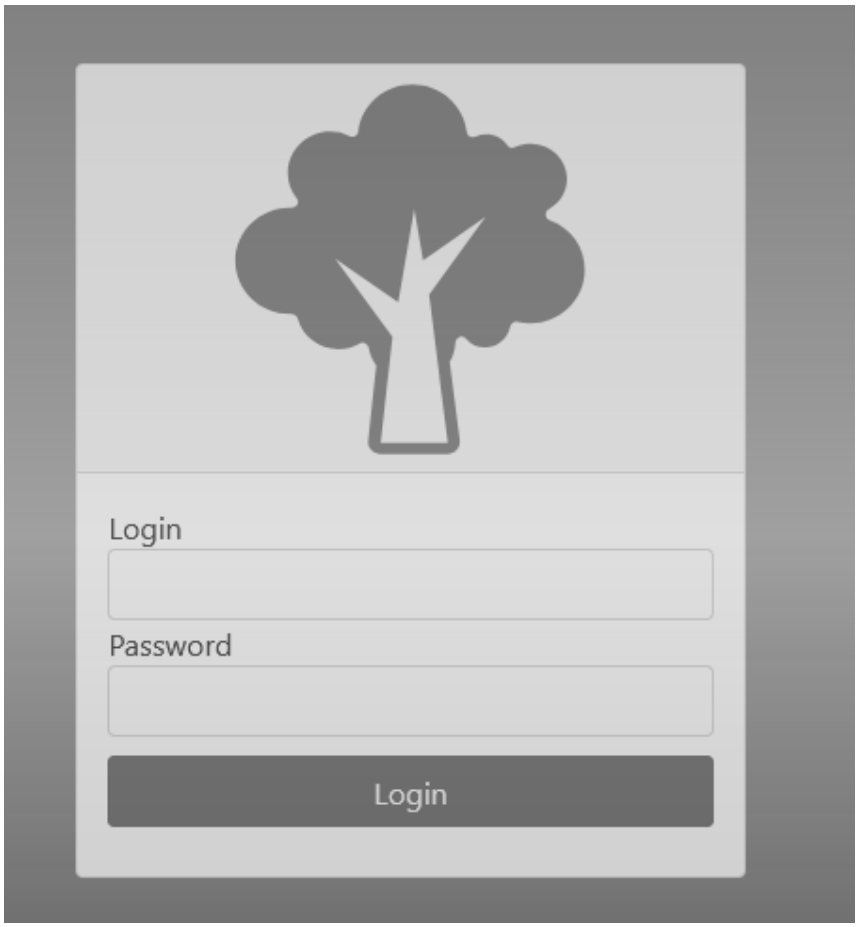

Figure 9.1 – Login Page

The page that you would be forwarded to is the search page. Its illustration can be seen on the fig 9.2. 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.

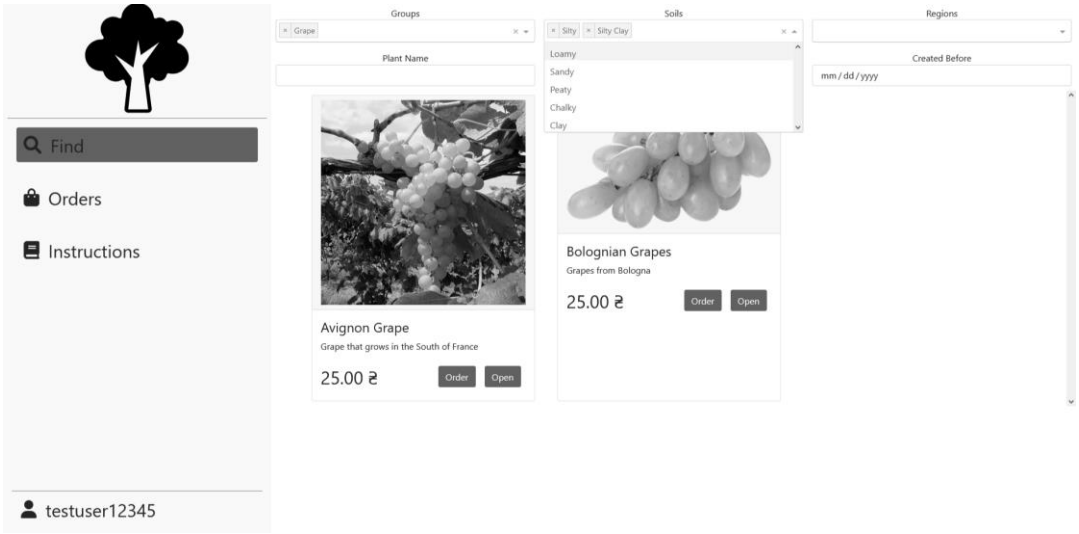

Figure 9.2 – Consumer Search page

Page with the detailed plant information can be accessed through search page. Its diagram can be found on fig 9.3. 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 9.3 – 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. 9.4. 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.

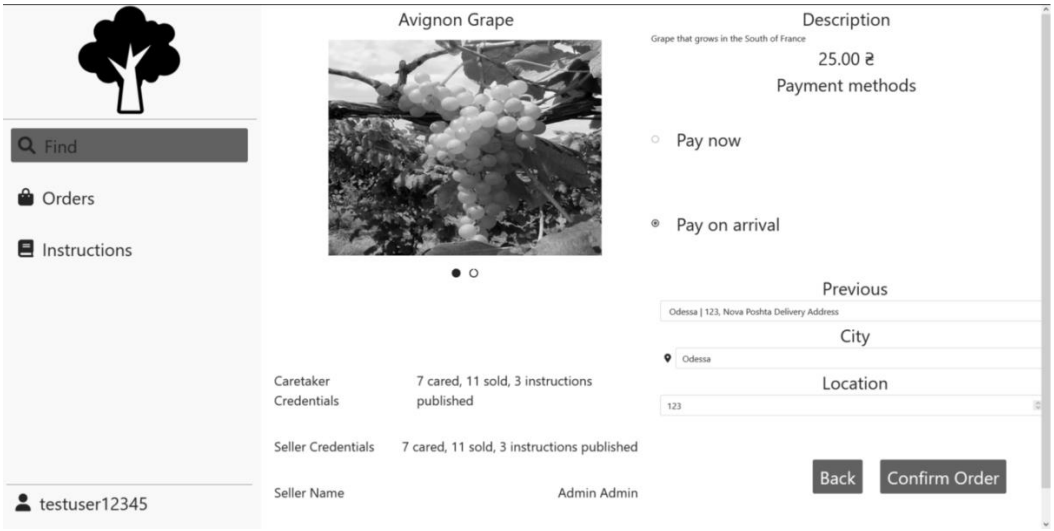

Figure 9.4- Order page

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. 9.5. 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.

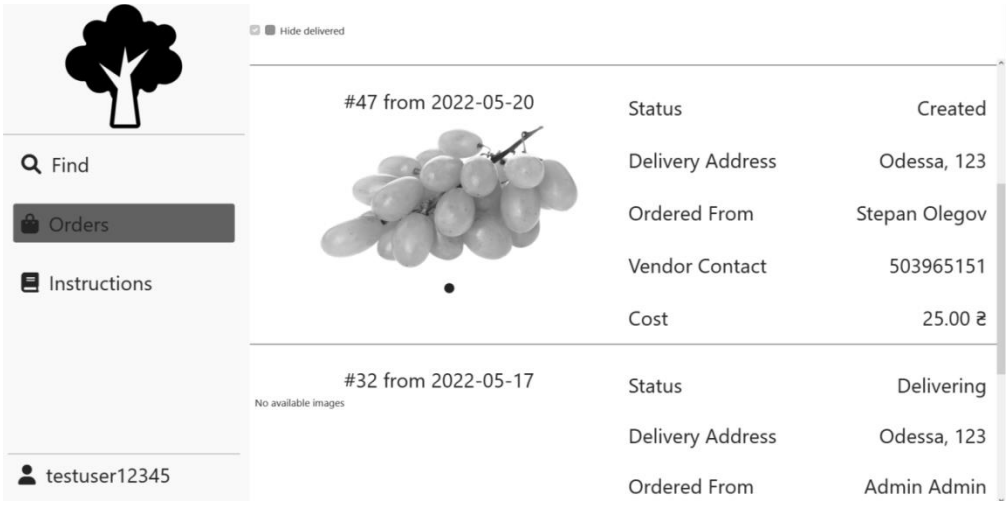

Figure 9.5 – Consumer Orders page

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. 9.6. This page allows you to change filtering options and then open one for the full view.

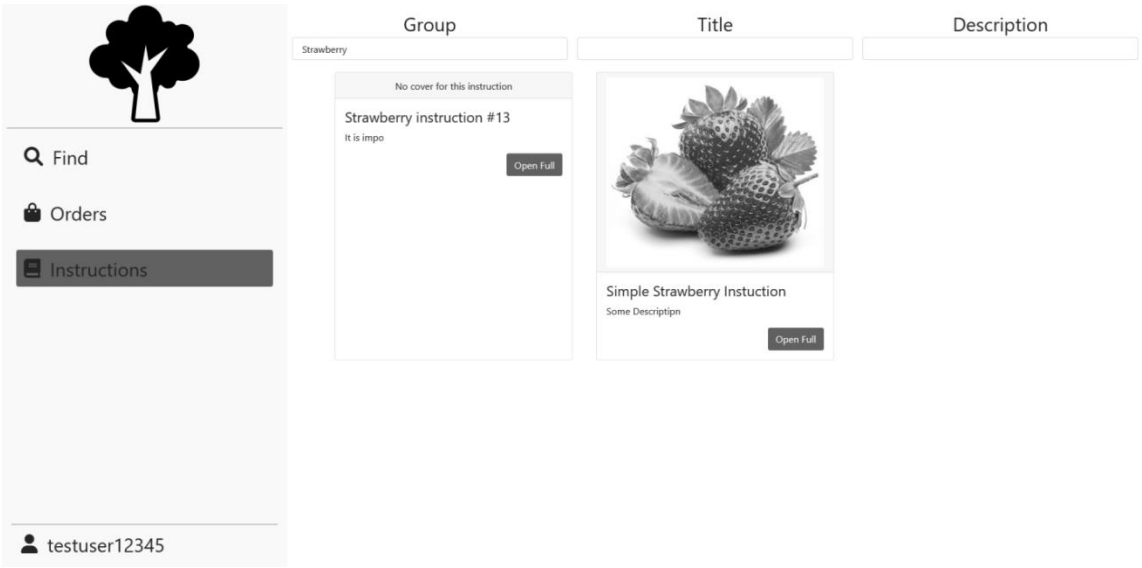

Figure 9.6 – Instructions page

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.

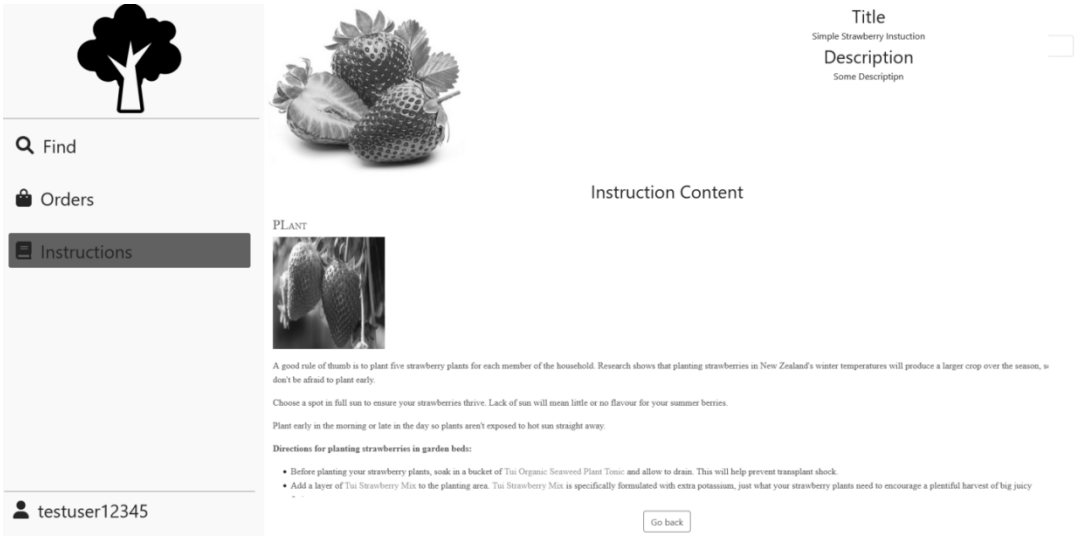

Figure 9.7 – Instruction page

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 9.8.

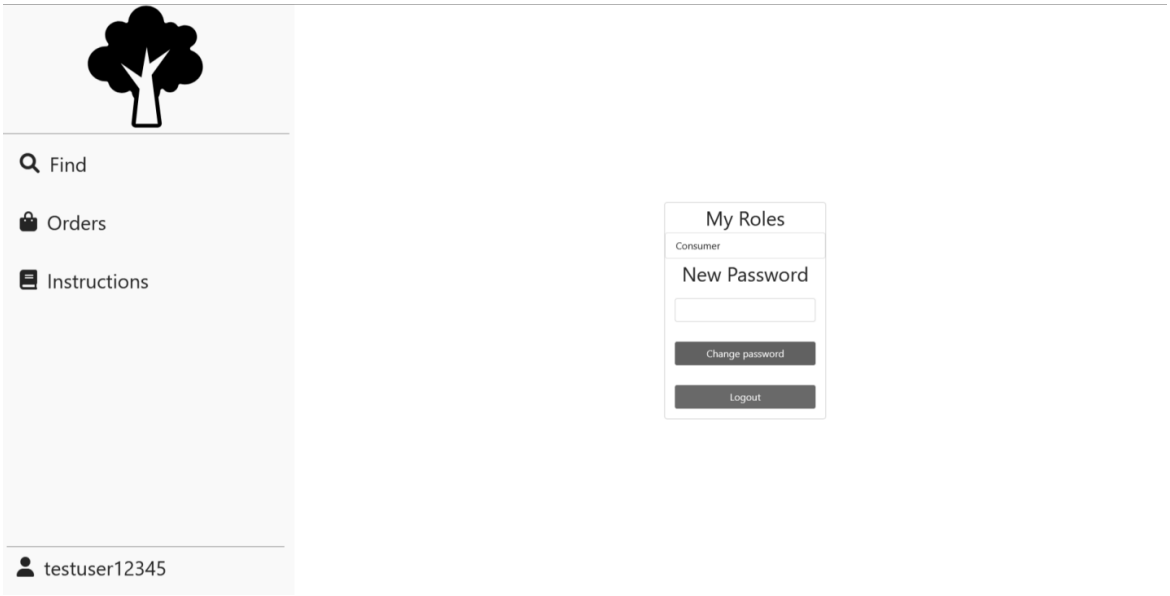

Figure 9.8 – Profile page

9.2 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 9.9.

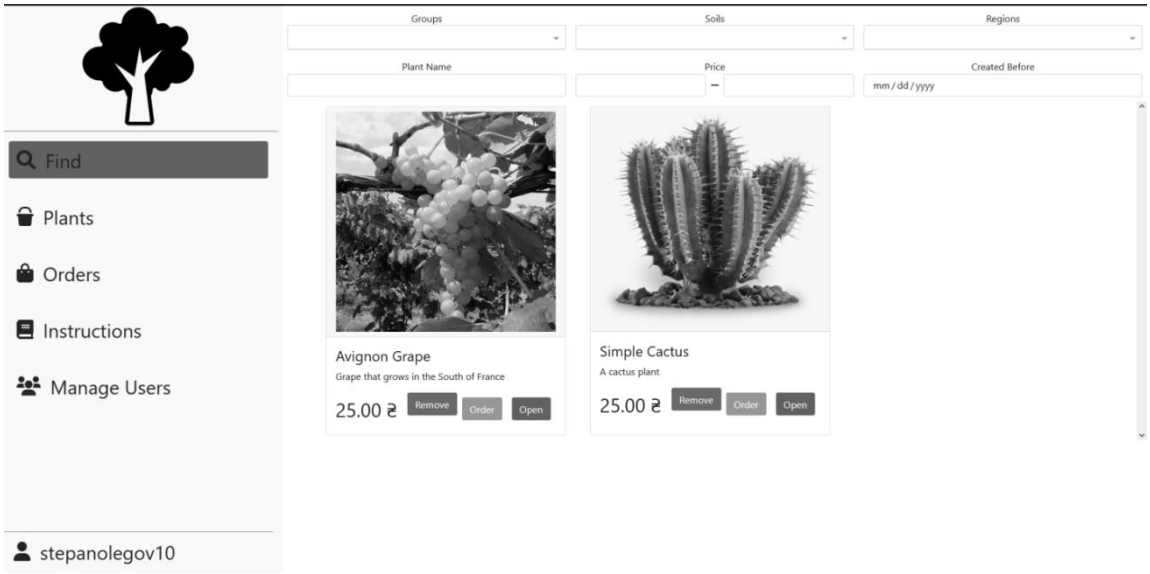

Figure 9.9 – Producer Search page

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 9.10. 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.

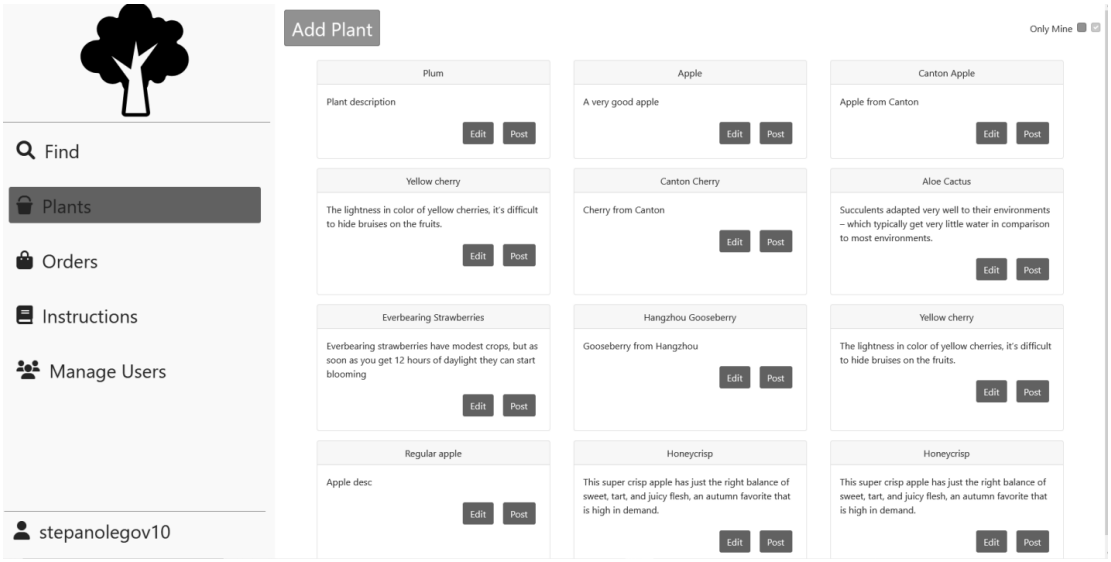

Figure 9.10 – Plants page

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 9.11.

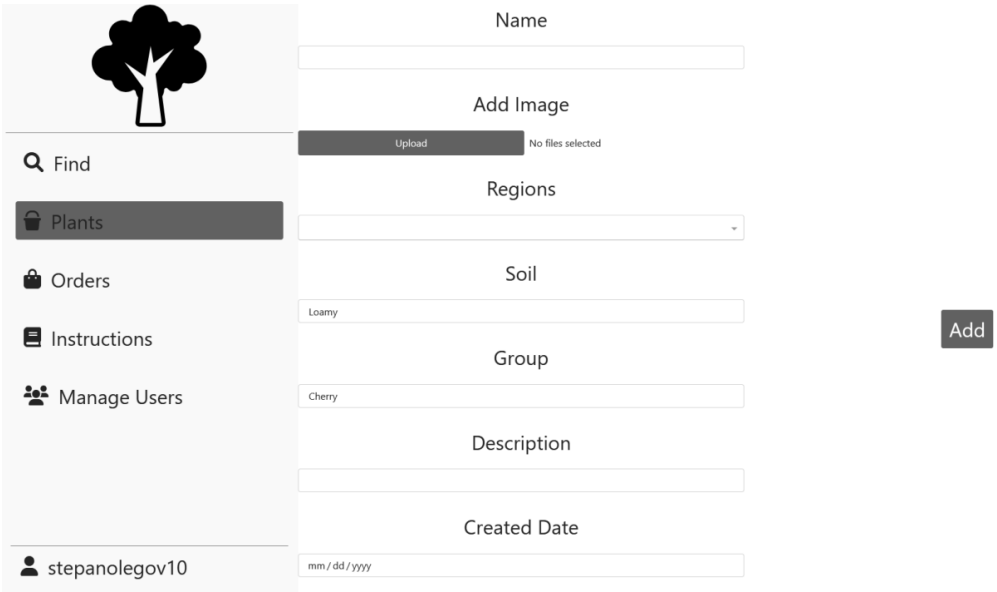

Figure 9.11 – Add plant

Edit plant page is accessible through selecting edit on plant from plants page. Its illustration can be seen on fig 9.12. 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.

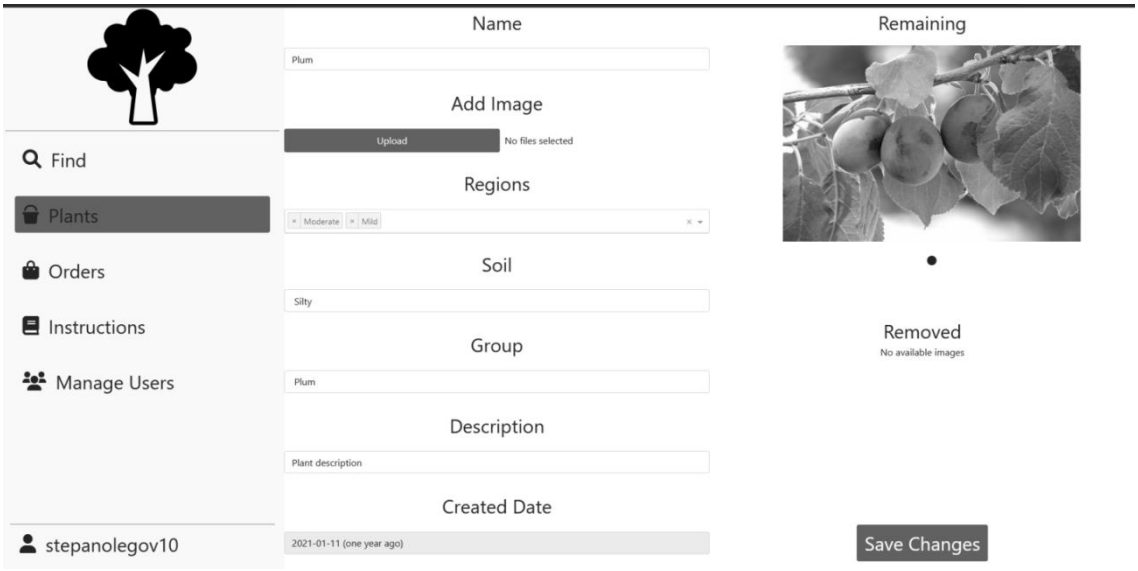

Figure 9.12 – Edit plant

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 9.13. Upon clicking on edit text a full-screen text editor would be opened. After clicking on Create an instruction would be created.

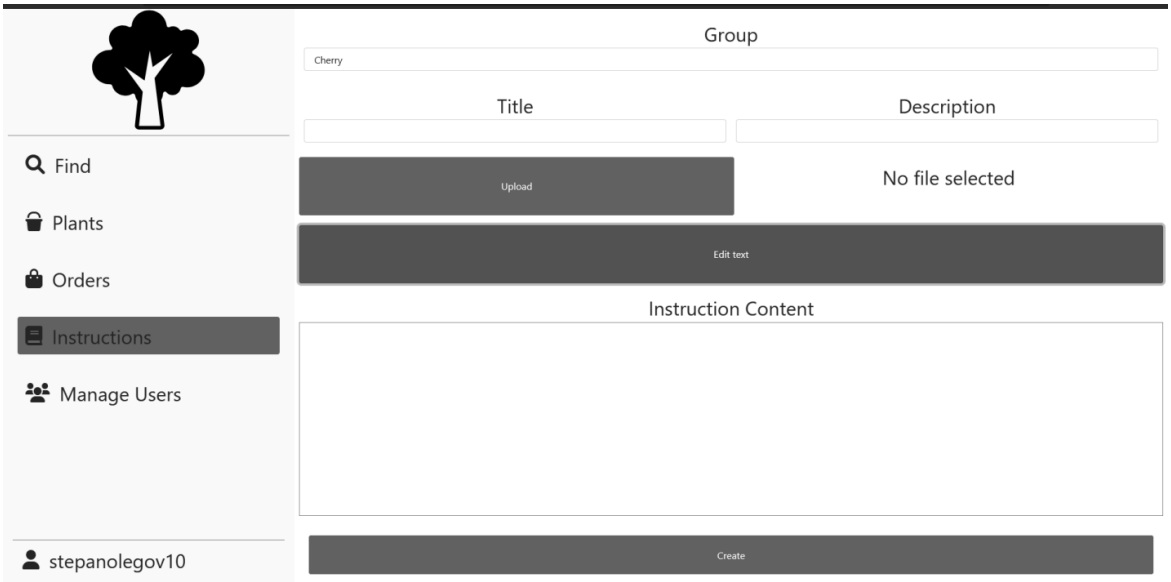

Figure 9.13 – Add instruction

Edit instruction page is accessible through instructions page by clicking on edit on an instruction. Its illustration can be seen on fig 9.14. It allows the producer to change any information about an instruction.

Figure 9.14 – Edit instruction

Orders page is accessible through left navigational bar. Its illustration can be seen on fig 9.15. 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.

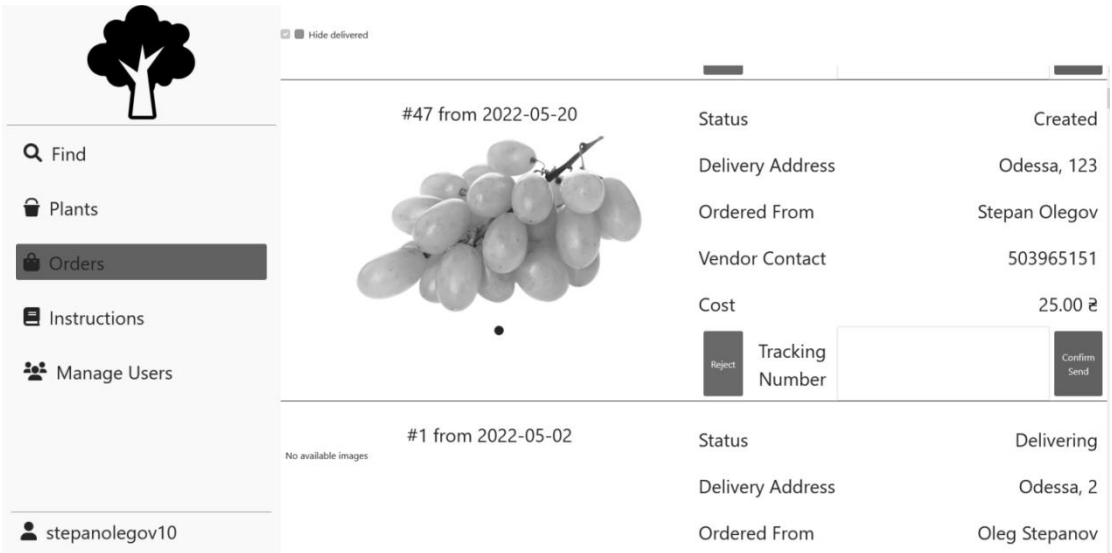

Figure 9.15 – Producer Orders page

Users page can be accessed through left navigational bar. Its illustration can be seen on fig 9.16. 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.

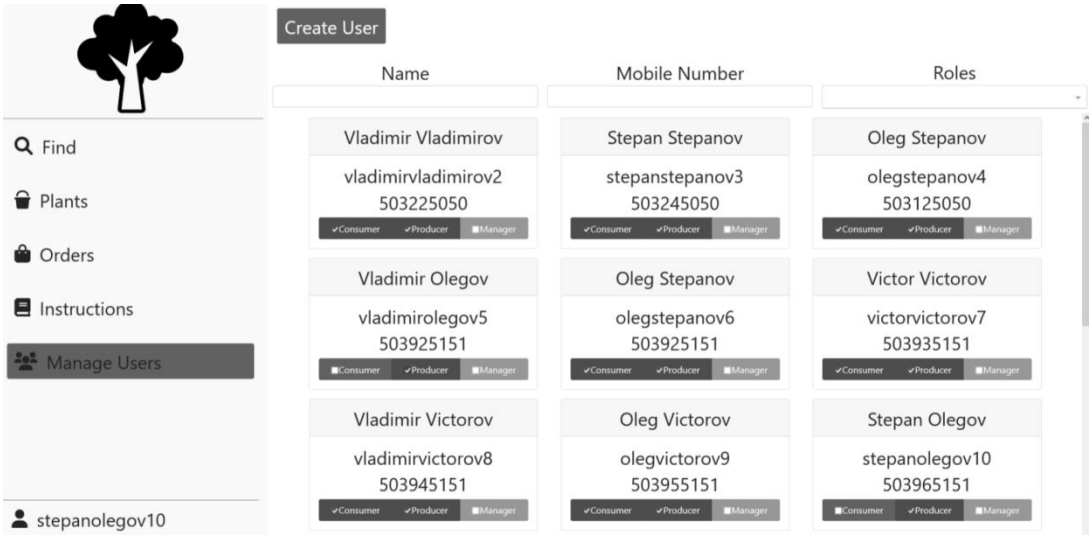

Figure 9.16 – Users page

Add user page displays information required to create a user. Its illustration can be seen on fig 9.17. Upon selecting all of the information and clicking on invite an invite would get send to the selected email.

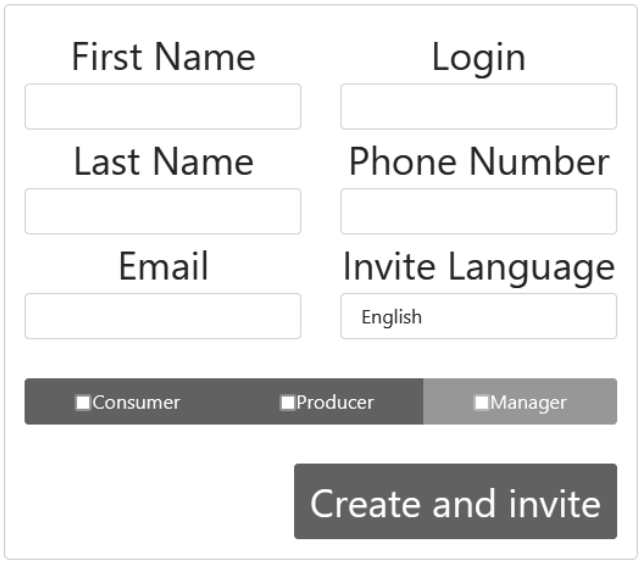

Figure 9.17 – Add user page

9.3 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. 9.18 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 9.18 – Total statistics page

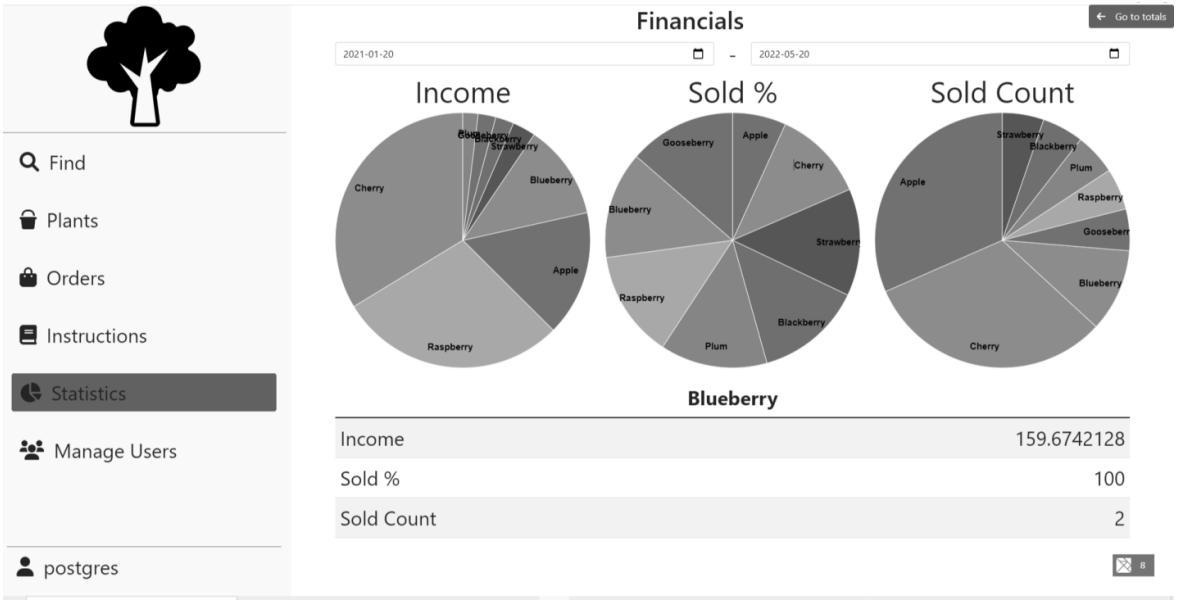

Figure 9.19 – Financial statistics page

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. The technological choices have been made based on the requirements and as such are a good fit for the system, which includes their relative popularity that greatly extends potential talent pool.

Potential venues of expansion may include:

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

REFERENCES

1. Lock A. ASP.NET Core in Action – Manning, 2018. – 278 p.
2. PostgreSQL: Documentation: 14: CREATE TRIGGER - https://www.postgresql.org/docs/current/sql-createtrigger.html.
3. Common web application architectures: Microsoft Docs - https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures.
4. PostgreSQL: Documentation: 9.4: Alter Role - https://www.postgresql.org/docs/9.4/sql-alterrole.html.
5. Elm Architecture Documentation - https://guide.elm-lang.org/architecture.

APPENDIX A

Database tables creation

```
--creating tables
CREATE TABLE plant_group (
    id serial PRIMARY KEY,
    group_name text
);

CREATE TABLE plant_region (
    id serial PRIMARY KEY,
    region_name text
);

CREATE TABLE plant_soil (
    id serial PRIMARY KEY,
    soil_name text
);

CREATE TABLE delivery_address (
    id serial PRIMARY KEY,
    city text,
    nova_poshta_number smallint,
    UNIQUE (city, nova_poshta_number)
);

CREATE TABLE person (
    id serial PRIMARY KEY,
    first_name text NOT NULL,
    last_name text NOT NULL,
    phone_number text NOT NULL,
    delivery_address_id int REFERENCES delivery_address (id)
);

CREATE TABLE plant (
    id serial PRIMARY KEY,
    group_id int NOT NULL REFERENCES plant_group (id),
    soil_id int NOT NULL REFERENCES plant_soil (id),
    region_id int NOT NULL REFERENCES plant_region (id),
    care_taker_id int NOT NULL REFERENCES person (id),
    plant_name text NOT NULL,
    description text NOT NULL,
    created date NOT NULL
);

CREATE TABLE plant_caring_instruction (
    id serial PRIMARY KEY,
    instruction_text text,
    posted_by_id int NOT NULL REFERENCES person (id),
    plant_group_id int NOT NULL REFERENCES plant_group (id),
    title text NOT NULL,
    description text NOT NULL
```

```
);

CREATE TABLE plant_post (
    plant_id serial PRIMARY KEY REFERENCES plant (id),
    seller_id int NOT NULL REFERENCES person (id),
    price decimal NOT NULL CHECK (price >= 0),
    created date NOT NULL DEFAULT CURRENT_DATE
);

CREATE TABLE plant_order (
    post_id int PRIMARY KEY REFERENCES plant_post (plant_id),
    customer_id int NOT NULL REFERENCES person (id),
    delivery_address_id int REFERENCES delivery_address (id),
    created timestamptz DEFAULT now() NOT NULL
);

CREATE TABLE plant_delivery (
    order_id int PRIMARY KEY REFERENCES plant_order (post_id),
    delivery_tracking_number text NOT NULL,
    created timestamptz DEFAULT now() NOT NULL
);

CREATE TABLE plant_shipment (
    delivery_id int PRIMARY KEY REFERENCES plant_delivery
(order_id),
    shipped timestamptz DEFAULT now() NOT NULL
);

CREATE TABLE plant_to_image (
    relation_id serial PRIMARY KEY,
    plant_id int REFERENCES plant (id) NOT NULL,
    image bytea NOT NULL
);

CREATE TABLE person_to_delivery (
    id serial PRIMARY KEY,
    person_id int NOT NULL REFERENCES person (id),
    delivery_address_id int NOT NULL REFERENCES delivery_address
(id) ON DELETE CASCADE
);

CREATE TABLE instruction_to_cover (
    instruction_id serial PRIMARY KEY REFERENCES
plant_caring_instruction (id) ON DELETE CASCADE,
    image bytea NOT NULL
);

--adding logins
CREATE TABLE person_to_login (
    person_id int PRIMARY KEY REFERENCES person (id) ON DELETE
CASCADE,
    login name UNIQUE CHECK (LOGIN = lower(LOGIN))
```

```
);

CREATE GROUP consumer;

CREATE GROUP producer;

CREATE GROUP manager;

CREATE TYPE UserRoles AS ENUM (
    'consumer',
    'producer',
    'manager',
    'other'
);

CREATE OR REPLACE PROCEDURE create_user_login (username name,
userPass text, userRoles UserRoles[])
SECURITY DEFINER
AS $$
BEGIN
    EXECUTE FORMAT('CREATE USER %s WITH PASSWORD %L in group
%s', username, userPass, array_to_string(userRoles, ', '));
END;
$$
LANGUAGE plpgsql;

--Creating roles for persons
DO $$
DECLARE
    person record;
    currentLogin name;
BEGIN
    FOR person IN (
        SELECT
            *
        FROM
            person)
        LOOP
            currentLogin := person.first_name ||
person.last_name || person.id;
            CALL create_user_login (currentLogin, 'tempPass',
ARRAY['producer'::UserRoles, 'consumer'::UserRoles]);
            INSERT INTO person_to_login
                VALUES (person.id, currentLogin);
        END LOOP;
END
$$;

CREATE OR REPLACE FUNCTION person_check_login ()
    RETURNS TRIGGER
    SECURITY DEFINER
    AS $BODY$
```

```
BEGIN
    IF NOT EXISTS (
        SELECT
            1
        FROM
            pg_user
        WHERE
            usename = NEW.login) THEN
    RAISE EXCEPTION 'There is no login with id %', NEW.login
        USING HINT = 'Please, consider creating person through
specified sp';
    END IF;
    RETURN NEW;
END;
$BODY$
LANGUAGE 'plpgsql';

CREATE TRIGGER person_prevent_bad_login
    BEFORE INSERT OR UPDATE ON person_to_login
    FOR EACH ROW
    EXECUTE PROCEDURE person_check_login ();

-- Configuring root user
INSERT INTO person (id, first_name, last_name, phone_number)
    VALUES (0, 'Admin', 'Admin', '0503035050');

INSERT INTO person_to_login (person_id, login)
    VALUES (0, 'postgres');

ALTER
GROUP manager
    ADD USER postgres;
```

APPENDIX B

Database objects creation

```
--set poster
CREATE OR REPLACE FUNCTION get_current_user_id ()
  RETURNS integer
  SECURITY DEFINER
  AS $$
BEGIN
  RETURN COALESCE((
    SELECT
      p.person_id
    FROM person_to_login p
    WHERE
      p.login = SESSION_USER), -1);
END
$$
LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION set_current_user_id_care_taker ()
  RETURNS TRIGGER
  AS $BODY$
DECLARE
  userId int;
BEGIN
  userId := get_current_user_id ();
  IF userId = - 1 THEN
    RAISE EXCEPTION 'There is no person attached to %',
SESSION_USER
      USING HINT = 'Please, consider using credentials that have
a person attached to them';
    ELSE
      NEW.care_taker_id = userId;
    END IF;
    RETURN NEW;
END;
$BODY$
LANGUAGE 'plpgsql';

CREATE TRIGGER plant_set_poster
  BEFORE INSERT ON plant
  FOR EACH ROW
  EXECUTE PROCEDURE set_current_user_id_care_taker ();

CREATE OR REPLACE FUNCTION set_current_user_id_seller ()
  RETURNS TRIGGER
  AS $BODY$
DECLARE
  userId int;
BEGIN
  userId := get_current_user_id ();
  IF userId = - 1 THEN
```

```
    RAISE EXCEPTION 'There is no person attached to %',
SESSION_USER
      USING HINT = 'Please, consider using credentials that have
a person attached to them';
    ELSE
      NEW.seller_id = userId;
    END IF;
    RETURN NEW;
END;
$BODY$
LANGUAGE 'plpgsql';

CREATE TRIGGER post_set_poster
  BEFORE INSERT ON plant_post
  FOR EACH ROW
  EXECUTE PROCEDURE set_current_user_id_seller ();

CREATE OR REPLACE FUNCTION set_current_user_id_instruction ()
  RETURNS TRIGGER
  AS $BODY$
DECLARE
  userId int;
BEGIN
  userId := get_current_user_id ();
  IF userId = - 1 THEN
    RAISE EXCEPTION 'There is no person attached to %',
SESSION_USER
      USING HINT = 'Please, consider using credentials that have
a person attached to them';
    ELSE
      NEW.posted_by_id = userId;
    END IF;
    RETURN NEW;
END;
$BODY$
LANGUAGE 'plpgsql';

CREATE TRIGGER instruction_set_poster
  BEFORE INSERT ON plant_caring_instruction
  FOR EACH ROW
  EXECUTE PROCEDURE set_current_user_id_instruction ();
CREATE OR REPLACE VIEW plant_search_v AS
SELECT
  p.id,
  p.plant_name,
  po.price,
  p.created,
  gr.id AS group_id,
  s.id AS soil_id,
  ARRAY_REMOVE(array_agg(DISTINCT rg.id), NULL) AS regions
FROM
  plant_post po
```

```
  JOIN plant p ON p.id = po.plant_id
  JOIN plant_group gr ON gr.id = p.group_id
  JOIN plant_soil s ON s.id = p.soil_id
  LEFT JOIN plant_to_region prg ON prg.plant_id = p.id
  LEFT JOIN plant_region rg ON rg.id = prg.plant_region_id
GROUP BY
  p.id,
  gr.id,
  s.id,
  po.price;

--this would search plant table for provided values
--would skip search by specific field when null value is
provided
CREATE OR REPLACE FUNCTION search_plant (plantName text,
priceRangeBottom numeric, priceRangeTop numeric, lastDate
timestamp without time zone, groupIds integer[], soilIds
integer[], regionIds integer[])
  RETURNS TABLE (
    id integer,
    plant_name text,
    description text,
    price numeric,
    imageIds integer[])
  SECURITY DEFINER
  AS $$
BEGIN
  RETURN QUERY
  SELECT
    p.id,
    p.plant_name,
    p.description,
    se.price,
    array_remove(array_agg(i.relation_id), NULL)
  FROM
    plant_search_v se
    JOIN plant p ON p.id = se.id
    JOIN plant_group g ON g.id = p.group_id
    JOIN plant_soil s ON s.id = p.soil_id
    LEFT JOIN plant_to_image i ON i.plant_id = p.id
    LEFT JOIN plant_order o ON o.post_id = p.id
  WHERE
    o.customer_id IS NULL
    AND (plantName IS NULL
      OR to_tsvector(se.plant_name) @@ to_tsquery(plantName))
    AND (priceRangeBottom IS NULL
      OR se.price >= priceRangeBottom)
    AND (priceRangeTop IS NULL
      OR se.price <= priceRangeTop)
    AND (lastDate IS NULL
      OR se.created >= lastDate)
    AND (groupIds IS NULL
```

```
      OR se.group_id = ANY (groupIds))
    AND (soilIds IS NULL
      OR se.soil_id = ANY (soilIds))
    --&& means intersection
    AND (regionIds IS NULL
      OR regionIds && se.regions)
  GROUP BY
    p.id,
    se.price;
END;
$$
LANGUAGE plpgsql;
CREATE VIEW dicts_v AS (
  SELECT
    array_agg(g.id) AS ids,
    array_agg(g.group_name) AS
  VALUES
,
    'group' AS type
  FROM
    plant_group g
  UNION
  SELECT
    array_agg(s.id),
    array_agg(s.soil_name),
    'soil' AS type
  FROM
    plant_soil s
  UNION
  SELECT
    array_agg(r.id),
    array_agg(r.region_name),
    'region' AS type
  FROM
    plant_region r);
CREATE OR REPLACE FUNCTION array_length_no_nulls (arr integer[])
  RETURNS bigint
  SECURITY DEFINER
  AS $$
BEGIN
  RETURN coalesce(array_length(array_remove(arr, NULL), 1), 0);
END;
$$
LANGUAGE plpgsql;

CREATE OR REPLACE VIEW person_creds_v AS (
  SELECT
    p.id,
    array_length_no_nulls (ARRAY_AGG(DISTINCT pl.id)) AS
cared_count,
    array_length_no_nulls (ARRAY_AGG(DISTINCT po.plant_id)) AS
sold_count,
```

```
    array_length_no_nulls (ARRAY_AGG(DISTINCT i.id)) AS
instructions_count
  FROM
    person p
  LEFT JOIN plant pl ON pl.care_taker_id = p.id
  LEFT JOIN plant_post po ON po.seller_id = p.id
  LEFT JOIN plant_caring_instruction i ON i.posted_by_id = p.id
GROUP BY
  p.id);

CREATE OR REPLACE VIEW plant_post_v AS (
  WITH posts_extended AS (
    SELECT
      p.id,
      p.plant_name,
      po.price,
      gr.group_name,
      s.soil_name,
      p.description,
      po.seller_id,
      p.care_taker_id,
      array_remove(array_agg(DISTINCT rg.region_name), NULL) AS
regions,
      p.created,
      array_remove(array_agg(DISTINCT img.relation_id), NULL) AS
img_ids
    FROM
      plant_post po
      JOIN plant p ON p.id = po.plant_id
      JOIN plant_group gr ON gr.id = p.group_id
      JOIN plant_soil s ON s.id = p.soil_id
      LEFT JOIN plant_to_region prg ON prg.plant_id = p.id
      LEFT JOIN plant_region rg ON rg.id = prg.plant_region_id
      LEFT JOIN plant_to_image img ON img.plant_id = p.id
    GROUP BY
      p.id,
      gr.group_name,
      s.soil_name,
      po.price,
      po.seller_id,
      p.care_taker_id
)
    SELECT
      post.id,
      post.plant_name,
      post.description,
      post.price,
      post.soil_name,
      post.regions,
      post.group_name,
      post.created,
```

```
      FORMAT('%s %s', seller.first_name, seller.last_name) AS
seller_name,
      seller.phone_number AS seller_phone,
      seller_creds.cared_count AS seller_cared,
      seller_creds.sold_count AS seller_sold,
      seller_creds.instructions_count AS seller_instructions,
      care_taker_creds.cared_count AS care_taker_cared,
      care_taker_creds.sold_count AS care_taker_sold,
      care_taker_creds.instructions_count AS
care_taker_instructions,
      post.img_ids AS images
    FROM
      posts_extended post
      JOIN person seller ON seller.id = post.seller_id
      LEFT JOIN person_creds_v seller_creds ON seller_creds.id =
post.seller_id
      LEFT JOIN person_creds_v care_taker_creds ON
care_taker_creds.id = post.care_taker_id);

CREATE OR REPLACE VIEW current_user_addresses AS (
  SELECT
    array_agg(d.city) AS cities,
    array_agg(d.nova_poshta_number) AS posts
  FROM
    person_to_delivery pd
    JOIN delivery_address d ON d.id = pd.delivery_address_id
    JOIN person p ON p.id = pd.person_id
  WHERE
    p.id = get_current_user_id ()
  GROUP BY
    p.id);

CREATE OR REPLACE FUNCTION get_current_user_id_throw ()
  RETURNS integer
  AS $BODY$
DECLARE
  userId int;
BEGIN
  userId := get_current_user_id ();
  IF userId = - 1 THEN
    RAISE EXCEPTION 'There is no person attached to %',
SESSION_USER
      USING HINT = 'Please, consider using credentials that have
a person attached to them';
    ELSE
      RETURN userId;
    END IF;
END;
$BODY$
LANGUAGE 'plpgsql';

CREATE OR REPLACE FUNCTION set_current_user_id_order ()
```

```
  RETURNS TRIGGER
  AS $BODY$
DECLARE
  userId int;
BEGIN
  userId := get_current_user_id_throw ();
  NEW.customer_id = userId;
  RETURN NEW;
END;
$BODY$
LANGUAGE 'plpgsql';

CREATE TRIGGER order_set_customer
  BEFORE INSERT ON plant_order
  FOR EACH ROW
  EXECUTE PROCEDURE set_current_user_id_order ();
--This view displays plants that have not been posted yet
CREATE OR REPLACE VIEW plants_v AS (
  SELECT
    p.id,
    p.plant_name,
    p.description,
    p.care_taker_id = get_current_user_id_throw () AS is_mine,
    p.group_id,
    p.soil_id,
    ARRAY_REMOVE(ARRAY_AGG(DISTINCT img.relation_id), NULL) AS
images,
    ARRAY_REMOVE(ARRAY_AGG(DISTINCT prg.plant_region_id), NULL)
AS regions,
    p.created
  FROM
    plant p
  LEFT JOIN plant_to_region prg ON prg.plant_id = p.id
  LEFT JOIN plant_to_image img ON img.plant_id = p.id
  LEFT JOIN plant_post po ON po.plant_id = p.id
WHERE
  po.plant_id IS NULL
GROUP BY
  p.id);

--this view would display posts as they would be seen after
posting
CREATE VIEW prepared_for_post_v AS (
  WITH plant_extended AS (
    SELECT
      p.id,
      p.plant_name,
      gr.group_name,
      s.soil_name,
      p.description,
      p.care_taker_id,
```

```
      array_remove(array_agg(DISTINCT rg.region_name), NULL) AS
regions,
      p.created,
      array_remove(array_agg(DISTINCT img.relation_id), NULL) AS
images
    FROM
      plant p
      JOIN plant_group gr ON gr.id = p.group_id
      JOIN plant_soil s ON s.id = p.soil_id
      LEFT JOIN plant_to_region prg ON prg.plant_id = p.id
      LEFT JOIN plant_region rg ON rg.id = prg.plant_region_id
      LEFT JOIN plant_post po ON po.plant_id = p.id
      LEFT JOIN plant_to_image img ON img.plant_id = p.id
    WHERE
      po.plant_id IS NULL
    GROUP BY
      p.id,
      gr.group_name,
      s.soil_name
)
    SELECT
      p.id,
      p.plant_name,
      p.description,
      p.soil_name,
      p.regions,
      p.group_name,
      p.created,
      FORMAT('%s %s', seller.first_name, seller.last_name) AS
seller_name,
      seller.phone_number AS seller_phone,
      seller_creds.cared_count AS seller_cared,
      seller_creds.sold_count AS seller_sold,
      seller_creds.instructions_count AS seller_instructions,
      care_taker_creds.cared_count AS care_taker_cared,
      care_taker_creds.sold_count AS care_taker_sold,
      care_taker_creds.instructions_count AS
care_taker_instructions,
      p.images AS images
    FROM
      plant_extended p
      JOIN person seller ON seller.id =
get_current_user_id_throw ()
      LEFT JOIN person_creds_v seller_creds ON seller_creds.id =
seller.id
      LEFT JOIN person_creds_v care_taker_creds ON
care_taker_creds.id = p.care_taker_id);

--Reason Code:
-- 0 - all good
-- 1 - plant does not exist
-- 2 - already posted
```

```
-- 3 - bad price
-- 4 - is in planing
CREATE OR REPLACE FUNCTION post_plant (IN plantId int, IN price
numeric, OUT wasPlaced boolean, OUT reasonCode integer)
SECURITY DEFINER
AS $$
DECLARE
  plantExists boolean;
  postExists boolean;
  isInPlanning boolean;
BEGIN
  CREATE TEMP TABLE IF NOT EXISTS post_results AS
  SELECT
    p.id AS plant_id,
    po.plant_id AS post_id,
    p.created AS created
  FROM
    plant p
  LEFT JOIN plant_post po ON po.plant_id = p.id
WHERE
  p.id = plantId
LIMIT 1;
  plantExists := EXISTS (
    SELECT
      plant_id
    FROM
      post_results);
  postExists := (
    SELECT
      post_id
    FROM
      post_results) IS NOT NULL;
  isInPlanning := (
    SELECT
      created >= CURRENT_DATE
    FROM
      post_results);
  IF plantExists THEN
    IF postExists THEN
      wasPlaced := FALSE;
      reasonCode := 2;
    ELSE
      IF price <= 0 THEN
        wasPlaced := FALSE;
        reasonCode := 3;
      ELSE
        IF isInPlanning THEN
          wasPlaced := FALSE;
          reasonCode := 4;
        ELSE
          INSERT INTO plant_post (plant_id, price)
            VALUES (plantId, price);
```

```
          wasPlaced := TRUE;
          reasonCode := 0;
        END IF;
      END IF;
    END IF;
  ELSE
    wasPlaced := FALSE;
    reasonCode := 1;
  END IF;
  DROP TABLE post_results;
END;
$$
LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION create_plant (plantName text,
description text, regionIds int[], soilId int, groupId int,
created timestamp without time zone, pictures bytea[])
  RETURNS int
  SECURITY DEFINER
  AS $$
DECLARE
  plantId int;
  regionId int;
  picture bytea;
BEGIN
  INSERT INTO plant (created, description, group_id, plant_name,
soil_id)
    VALUES (created, description, groupId, plantName, soilId)
  RETURNING
    id INTO plantId;
  FOREACH regionId IN ARRAY regionIds LOOP
    INSERT INTO plant_to_region (plant_id, plant_region_id)
      VALUES (plantId, regionId);
  END LOOP;
  FOREACH picture IN ARRAY pictures LOOP
    INSERT INTO plant_to_image (plant_id, image)
      VALUES (plantId, picture);
  END LOOP;
  RETURN plantId;
END;
$$
LANGUAGE plpgsql;

CREATE OR REPLACE PROCEDURE edit_plant (plantId int, plantName
text, plantDescription text, regionIds int[], soilId int,
groupId int, removedImages int[], newImages bytea[])
SECURITY DEFINER
AS $$
DECLARE
  regionId int;
  picture bytea;
BEGIN
```

```
  UPDATE
    plant
  SET
    plant_name = plantName,
    description = plantDescription,
    soil_id = soilId,
    group_id = groupId
  WHERE
    id = plantId;
  DELETE FROM plant_to_region
  WHERE plant_id = plantId;
  FOREACH regionId IN ARRAY regionIds LOOP
    INSERT INTO plant_to_region (plant_id, plant_region_id)
      VALUES (plantId, regionId);
  END LOOP;
  DELETE FROM plant_to_image
  WHERE plant_id = plantId
    AND relation_id = ANY (removedImages);
  FOREACH picture IN ARRAY newImages LOOP
    INSERT INTO plant_to_image (plant_id, image)
      VALUES (plantId, picture);
  END LOOP;
END;
$$
LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION plant_no_update_posted ()
  RETURNS TRIGGER
  SECURITY DEFINER
  AS $BODY$
BEGIN
  IF EXISTS (
    SELECT
      plant_id
    FROM
      plant_post
    WHERE
      plant_id = NEW.id) THEN
  RAISE EXCEPTION 'You cannot edit posted plant';
ELSE
  RETURN NEW;
END IF;
END;
$BODY$
LANGUAGE 'plpgsql';

CREATE TRIGGER plant_prevent_update_of_posted
  BEFORE UPDATE ON plant
  FOR EACH ROW
  EXECUTE PROCEDURE plant_no_update_posted ();
--Reason Code:
-- 0 - all good
```

```
-- 1 - plant not posted
-- 2 - already ordered
CREATE OR REPLACE FUNCTION place_order (IN postId int,
delivery_city text, post_number integer, OUT wasPlaced boolean,
OUT reasonCode integer)
SECURITY DEFINER
AS $$
DECLARE
  userId int;
  postExists boolean;
  orderExists boolean;
  addressId int;
BEGIN
  CREATE TEMP TABLE IF NOT EXISTS order_results AS
  SELECT
    p.plant_id AS post_id,
    o.post_id AS order_id
  FROM
    plant_post p
  LEFT JOIN plant_order o ON p.plant_id = o.post_id
WHERE
  p.plant_id = postId
LIMIT 1;
  postExists := EXISTS (
    SELECT
      post_id
    FROM
      order_results);
  orderExists := (
    SELECT
      order_id
    FROM
      order_results) IS NOT NULL;
  IF postExists THEN
    IF orderExists THEN
      wasPlaced := FALSE;
      reasonCode := 2;
    ELSE
      userId := get_current_user_id_throw ();
      addressId := (
        SELECT
          id
        FROM
          delivery_address
        WHERE
          nova_poshta_number = post_number
          AND city = delivery_city);
      IF addressId IS NULL THEN
        INSERT INTO delivery_address (city, nova_poshta_number)
          VALUES (delivery_city, post_number)
        RETURNING
          id INTO addressId;
```

```
      END IF;
      INSERT INTO plant_order (delivery_address_id, post_id)
        VALUES (addressId, postId);
      wasPlaced := TRUE;
      reasonCode := 0;
    END IF;
  ELSE
    wasPlaced := FALSE;
    reasonCode := 1;
  END IF;
  DROP TABLE order_results;
END;
$$
LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION order_store_user_address ()
  RETURNS TRIGGER
  AS $BODY$
DECLARE
  userId int;
BEGIN
  userId := get_current_user_id_throw ();
  IF NOT EXISTS (
    SELECT
      delivery_address_id
    FROM
      person_to_delivery
    WHERE
      delivery_address_id = NEW.delivery_address_id
      AND person_id = userId) THEN
  INSERT INTO person_to_delivery (person_id,
delivery_address_id)
    VALUES (userId, NEW.delivery_address_id);
END IF;
  RETURN NEW;
END;
$BODY$
LANGUAGE 'plpgsql';

CREATE TRIGGER order_store_used_address
  AFTER INSERT ON plant_order
  FOR EACH ROW
  EXECUTE PROCEDURE order_store_user_address ();
CREATE OR REPLACE VIEW plant_stats_v AS (
  WITH gToInstruction AS (
    SELECT
      plant_group_id AS gid,
      Count(*) AS cnt
    FROM
      plant_caring_instruction
    GROUP BY
      plant_group_id),
```

```
    gToPlants AS (
      SELECT
        group_id AS gid,
        Count(*) AS cnt
      FROM
        plant
      GROUP BY
        group_id),
      gToIncome AS (
        SELECT
          p.group_id AS gid,
          SUM(price) AS total
        FROM
          plant_shipment s
          JOIN plant_order o ON o.post_id = s.delivery_id
          JOIN plant_post po ON po.plant_id = o.post_id
          JOIN plant p ON p.id = po.plant_id
        GROUP BY
          group_id),
        gToPopularity AS (
          SELECT
            p.group_id AS gid,
            COUNT(*) AS total
          FROM
            plant_order o
            JOIN plant_post po ON po.plant_id = o.post_id
            JOIN plant p ON p.id = po.plant_id
          GROUP BY
            group_id
)
          SELECT
            g.id,
            g.group_name,
            p.cnt AS plants_count,
            p2.total AS popularity,
            i.total AS income,
            i2.cnt AS instructions
          FROM
            gToPlants p
            JOIN gToPopularity p2 USING (gid)
            JOIN gToIncome i USING (gid)
            JOIN gToInstruction i2 USING (gid)
            JOIN plant_group g ON g.id = (gid));

--financial stats
CREATE OR REPLACE FUNCTION get_financial (start_date timestamp
without time zone, end_date timestamp without time zone)
  RETURNS TABLE (
    groupId int,
    group_name text,
    sold_count bigint,
    percent_sold numeric,
```

```
    income numeric)
  SECURITY DEFINER
  AS $$
BEGIN
  RETURN QUERY ( WITH group_to_post_count AS (
      SELECT
        p.group_id, count(*) AS total FROM plant p
      JOIN plant_post pl ON pl.plant_id = p.id
      LEFT JOIN plant_shipment s ON s.delivery_id = p.id
      WHERE
        s.delivery_id IS NULL
        OR s.shipped BETWEEN start_date AND end_date GROUP BY
p.group_id)
    SELECT
      g.id, g.group_name, count(*) AS sold_count,
round((count(*) * 1.0 / pc.total) * 100) AS percent_sold,
sum(p.price) AS income FROM plant pl
    JOIN plant_post p ON p.plant_id = pl.id
    JOIN plant_order o ON o.post_id = p.plant_id
    JOIN plant_shipment s ON s.delivery_id = o.post_id
    JOIN plant_group g ON g.id = pl.group_id
    JOIN group_to_post_count pc ON pc.group_id = g.id
    WHERE
      s.shipped BETWEEN start_date AND end_date GROUP BY g.id,
pc.total);
END
$$
LANGUAGE plpgsql;
--case : 0 - created, 1 - delivering, 2 - delivered
CREATE OR REPLACE VIEW plant_orders_v AS (
  SELECT
    (
      CASE WHEN s.delivery_id IS NOT NULL THEN
        2
      WHEN d.order_id IS NOT NULL THEN
        1
      ELSE
        0
      END) AS status,
    o.post_id,
    o.created AS ordered,
    da.city,
    da.nova_poshta_number AS mail_number,
    seller.first_name || ' ' || seller.last_name AS seller_name,
    seller.phone_number AS seller_contact,
    po.price,
    d.delivery_tracking_number,
    d.created AS delivery_started,
    s.shipped,
    ARRAY_REMOVE(ARRAY_AGG(DISTINCT img.relation_id), NULL) AS
images
  FROM
```

```
    plant_order o
    JOIN delivery_address da ON da.id = o.delivery_address_id
    JOIN plant_post po ON po.plant_id = o.post_id
    JOIN person seller ON seller.id = po.seller_id
    LEFT JOIN plant_delivery d ON d.order_id = o.post_id
    LEFT JOIN plant_shipment s ON s.delivery_id = d.order_id
    LEFT JOIN plant_to_image img ON img.plant_id = o.post_id
  GROUP BY
    o.post_id,
    s.delivery_id,
    da.id,
    d.order_id,
    seller.id,
    po.price
  ORDER BY
    status,
    ordered,
    post_id);

CREATE OR REPLACE VIEW current_user_orders AS (
  SELECT
    v.*
  FROM
    plant_orders_v v
    JOIN plant_order o ON v.post_id = o.post_id
  WHERE
    o.customer_id = get_current_user_id_throw ());

CREATE OR REPLACE PROCEDURE confirm_received (deliveryId int)
SECURITY DEFINER
AS $$
DECLARE
  buyerId int;
BEGIN
  SELECT
    customer_id INTO buyerId
  FROM
    plant_order
  WHERE
    post_id = deliveryId;
  IF buyerId = get_current_user_id_throw () THEN
    INSERT INTO plant_shipment (delivery_id)
      VALUES (deliveryId);
  ELSE
    RAISE EXCEPTION 'You cannot confirm delivery on order you
have not made';
  END IF;
END
$$
LANGUAGE plpgsql;
CREATE OR REPLACE VIEW instruction_v AS (
  SELECT
```

```
    i.id,
    i.plant_group_id,
    i.title,
    i.description,
    i.instruction_text,
    c.image IS NOT NULL AS has_cover
  FROM
    plant_caring_instruction i
  LEFT JOIN instruction_to_cover c ON c.instruction_id = i.id);

CREATE OR REPLACE FUNCTION search_instructions (groupId int,
instructionTitle text, instructionDescription text)
  RETURNS TABLE (
    id int,
    title text,
    description text,
    has_cover boolean)
  SECURITY DEFINER
  AS $$
BEGIN
  RETURN QUERY (
    SELECT
      i.id, i.title, i.description, i.has_cover
    FROM instruction_v i
    WHERE
      plant_group_id = groupId
      AND (instructionTitle IS NULL
        OR to_tsvector(i.title) @@ to_tsquery(instructionTitle))
    AND (instructionDescription IS NULL
      OR to_tsvector(i.description) @@
to_tsquery(instructionDescription)));
END
$$
LANGUAGE plpgsql;

--producer
CREATE OR REPLACE FUNCTION create_instruction (groupId int,
instructionText text, instructionTitle text,
instructionDescription text, coverImage bytea)
  RETURNS int
  SECURITY DEFINER
  AS $$
DECLARE
  instructionId int;
BEGIN
  INSERT INTO plant_caring_instruction (instruction_text,
plant_group_id, title, description)
    VALUES (instructionText, groupId, instructionTitle,
instructionDescription)
  RETURNING
    id INTO instructionId;
  IF coverImage IS NOT NULL THEN
```

```
    INSERT INTO instruction_to_cover (instruction_id, image)
      VALUES (instructionId, coverImage);
  END IF;
  RETURN instructionId;
END
$$
LANGUAGE plpgsql;

CREATE OR REPLACE PROCEDURE edit_instruction (instructionId int,
groupId int, instructionText text, instructionTitle text,
instructionDescription text, coverImage bytea)
SECURITY DEFINER
AS $$
BEGIN
  UPDATE
    plant_caring_instruction
  SET
    plant_group_id = groupId,
    instruction_text = instructionText,
    title = instructionTitle,
    description = instructionDescription
  WHERE
    id = instructionId;
  IF coverImage IS NOT NULL THEN
    INSERT INTO instruction_to_cover (instruction_id, image)
      VALUES (instructionId, coverImage)
    ON CONFLICT (instruction_id)
      DO UPDATE SET
        image = coverImage;
  END IF;
END
$$
LANGUAGE plpgsql;
CREATE OR REPLACE FUNCTION delete_only_creator_or_manager
(isOrder boolean)
  RETURNS TRIGGER
  SECURITY DEFINER
  AS $BODY$
DECLARE
  userId int;
  posterId int;
BEGIN
  IF isOrder THEN
    posterId := (
      SELECT
        seller_id
      FROM
        plant_post
      WHERE
        plant_id = OLD.post_id);
  ELSE
    posterId := OLD.seller_id;
```

```
  END IF;
  userId := get_current_user_id_throw ();
  IF NOT (
    SELECT
      'manager'::UserRoles = ANY (ur.roles) OR posterId =
ur.person_id
  FROM
    user_to_roles ur
  WHERE
    ur.person_id = userId) THEN
    RAISE EXCEPTION 'You cannot delete post you have not
created';
  END IF;
  RETURN OLD;
END;
$BODY$
LANGUAGE 'plpgsql';

CREATE TRIGGER post_prevent_unlawfull_delete
  BEFORE DELETE ON plant_post
  FOR EACH ROW
  EXECUTE PROCEDURE delete_only_creator_or_manager (FALSE);

CREATE TRIGGER order_prevent_unlawfull_delete
  BEFORE DELETE ON plant_order
  FOR EACH ROW
  EXECUTE PROCEDURE delete_only_creator_or_manager (TRUE);
```

APPENDIX C

Access Grants

```
REVOKE ALL ON SCHEMA public FROM public;

GRANT USAGE ON SCHEMA public TO public;

REVOKE ALL ON ALL TABLES IN SCHEMA public FROM PUBLIC;

REVOKE ALL ON ALL FUNCTIONS IN SCHEMA public FROM PUBLIC;

REVOKE ALL ON ALL PROCEDURES IN SCHEMA public FROM PUBLIC;

--tables
GRANT SELECT ON plant_to_image TO consumer, producer, manager;

GRANT SELECT ON instruction_to_cover TO consumer, producer,
manager;

GRANT SELECT, DELETE ON plant_post TO producer, manager;

GRANT SELECT, DELETE ON plant_order TO producer, manager;

GRANT INSERT ON plant_delivery TO producer, manager;

--views
GRANT SELECT ON current_user_roles TO consumer, producer,
manager;

GRANT SELECT ON dicts_v TO consumer, producer, manager;

GRANT SELECT ON plant_post_v TO consumer, producer, manager;

GRANT SELECT ON current_user_addresses TO consumer;

GRANT SELECT ON current_user_orders TO consumer;

GRANT SELECT ON instruction_v TO consumer, producer, manager;

GRANT SELECT ON plants_v TO producer, manager;

GRANT SELECT ON prepared_for_post_v TO producer, manager;

GRANT SELECT ON plant_orders_v TO producer, manager;

GRANT SELECT ON plant_stats_v TO manager;

--functions
--business
GRANT EXECUTE ON FUNCTION search_plant TO consumer, producer,
manager;
```

```
GRANT EXECUTE ON FUNCTION place_order TO consumer;

GRANT EXECUTE ON FUNCTION search_instructions TO consumer,
producer, manager;

GRANT EXECUTE ON FUNCTION post_plant TO producer, manager;

GRANT EXECUTE ON FUNCTION create_plant TO producer, manager;

GRANT EXECUTE ON FUNCTION create_instruction TO producer,
manager;

GRANT EXECUTE ON FUNCTION search_users TO producer, manager;

GRANT EXECUTE ON FUNCTION get_financial TO manager;

--utility
GRANT EXECUTE ON FUNCTION array_length_no_nulls TO consumer,
producer, manager;

GRANT EXECUTE ON FUNCTION get_current_user_id TO consumer,
producer, manager;

GRANT EXECUTE ON FUNCTION get_current_user_id_throw TO consumer,
producer, manager;

GRANT EXECUTE ON FUNCTION parse_role TO consumer, producer,
manager;

--procedures
GRANT EXECUTE ON PROCEDURE edit_plant TO producer, manager;

GRANT EXECUTE ON PROCEDURE confirm_received TO consumer;

GRANT EXECUTE ON PROCEDURE edit_instruction TO producer,
manager;

GRANT EXECUTE ON PROCEDURE add_user_to_group TO producer,
manager;

GRANT EXECUTE ON PROCEDURE remove_user_from_group TO manager;

GRANT EXECUTE ON PROCEDURE create_user TO producer, manager;

//paste 014_Grants here
```

APPENDIX D

Database Diagram

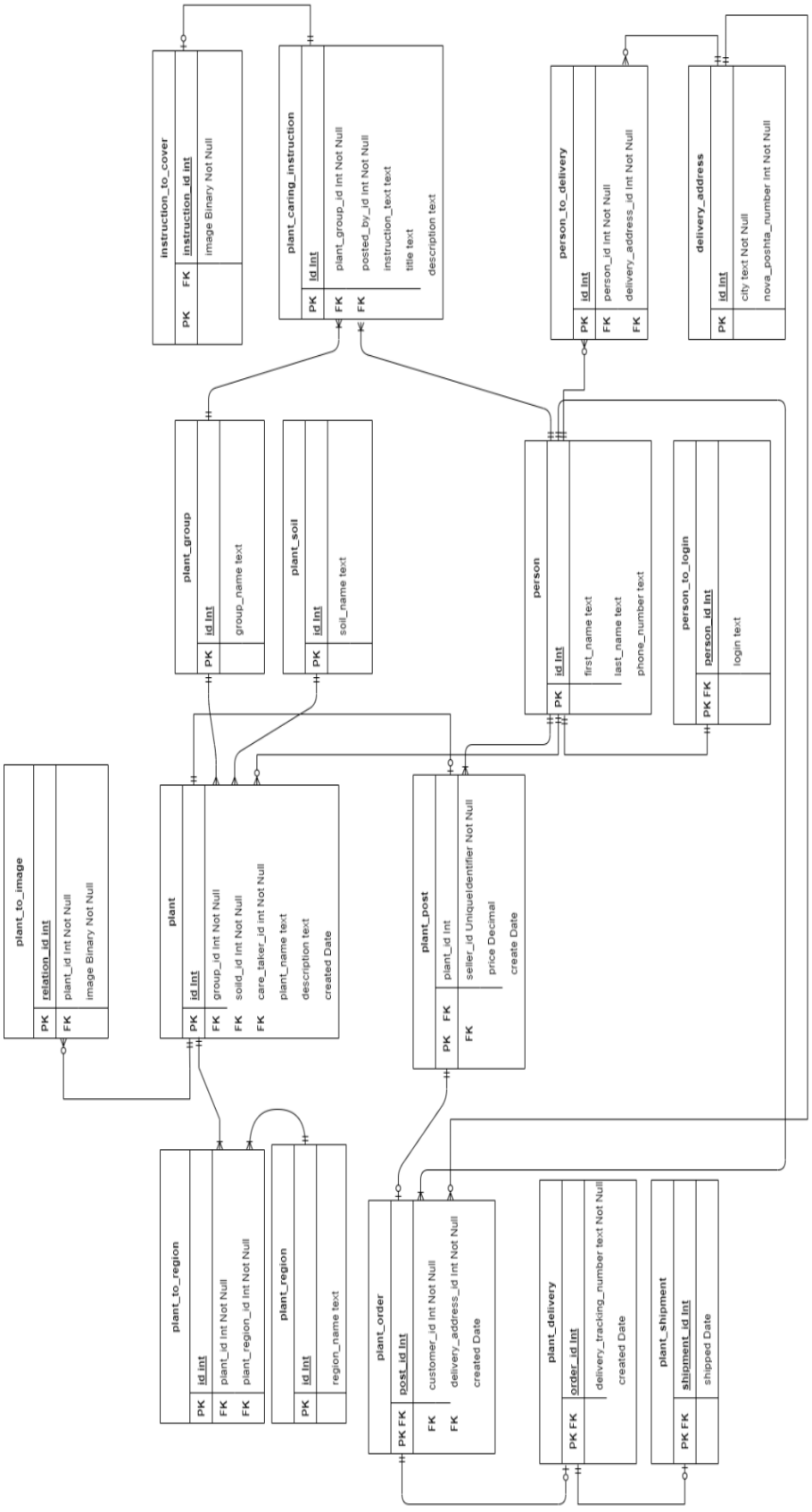

Figure D.1 – Entity relationship diagram of the plants database.

APPENDIX E

Shared frontend modules

Endpoints

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


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"

        Image id token ->
            baseUrl ++ "file/plant/" ++ String.fromInt id ++
"?token=" ++ token

        Post plantId ->
            baseUrl ++ "post/" ++ String.fromInt plantId

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

        Addresses ->
            baseUrl ++ "info/addresses"

        NotPostedPlants ->
            baseUrl ++ "plants/notposted"

        PreparedPlant plantId ->
            baseUrl ++ "plants/prepared/" ++ String.fromInt
plantId

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

        NotPostedPlant id ->
            baseUrl ++ "plants/notposted/" ++ String.fromInt id

        AddPlant ->
            baseUrl ++ "plants/add"

        EditPlant plantId ->
            baseUrl ++ "plants/" ++ String.fromInt plantId ++
"/edit"

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

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

        SendOrder orderId ttn ->
```

```
            baseUrl ++ "orders/" ++ String.fromInt orderId ++
"/deliver?trackingNumber=" ++ ttn

        ReceivedOrder orderId ->
            baseUrl ++ "orders/" ++ String.fromInt 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"

        CoverImage id token ->
            baseUrl ++ "file/instruction/" ++ String.fromInt id
++ "?token=" ++ token

        CreateInstruction ->
            baseUrl ++ "instructions/create"

        GetInstruction id ->
            baseUrl ++ "instructions/" ++ String.fromInt id

        EditInstruction id ->
            baseUrl ++ "instructions/" ++ String.fromInt id ++
"/edit"

        DeletePost id ->
            baseUrl ++ "post/" ++ String.fromInt id ++ "/delete"

        RejectOrder orderId ->
            baseUrl ++ "orders/" ++ String.fromInt orderId ++
"/reject"

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

NavBar

```
viewNav : ModelBase model -> Maybe Link -> (AuthResponse ->
model -> Html msg) -> Html msg
viewNav model link pageView =
```

```
    let
        viewP =
            viewMain link pageView
    in
    viewBase viewP model


viewMain : Maybe Link -> (AuthResponse -> model -> Html msg) ->
AuthResponse -> model -> Html msg
viewMain link pageView resp model =
    viewNavBase resp.username resp.roles link (pageView resp
model)


viewNavBase : String -> List UserRole -> Maybe Link -> Html msg
-> Html msg
viewNavBase username roles currentLink baseView =
    div fillScreen
        [ div ([ flex, Flex.row ] ++ fillParent) [ navBar
username roles currentLink, div [ style "flex" "3", style
"margin-left" "25vw" ] [ baseView ] ]
        ]


navBar : String -> List UserRole -> Maybe Link -> Html msg
navBar username roles currentLink =
    div
        [ flex1
        , style "height" "100%"
        , style "width" "25vw"
        , style "margin-right" "0.5em"
        , class "bg-light"
        , style "position" "fixed"
        ]
        [ div ([ flex, Flex.col, style "justify-content" "space-
between" ] ++ fillParent)
            [ div []
                [ div [ flex, Flex.row, Flex.justifyCenter ]
                    [ treeIcon (px 200) Color.black
                    ]
                , linksView currentLink <| getLinksFor roles
                ]
            , div [] [ userView username ]
            ]
        ]


linksView : Maybe Link -> List Link -> Html msg
linksView selected links =
    let
        isSelected link =
            case selected of
```

```
                Just selectedLink ->
                    link.url == selectedLink.url

                Nothing ->
                    False
    in
    div [ flex, Flex.col, style "border-top" "solid gray 1px",
smallMargin ] (List.map (\link -> linkView (isSelected link)
link) links)


linkView : Bool -> Link -> Html msg
linkView isSelected link =
    let
        backColor =
            if isSelected then
                "bg-primary"

            else
                ""
    in
    a [ href link.url ]
        [ div [ class "nav-bar-item", flex, Flex.row,
smallMargin, Flex.alignItemsCenter, largeFont, class ("btn " ++
backColor) ]
            [ i [ class link.icon, style "margin-right" "0.5em"
] []
            , text link.text
            ]
        ]


userView : String -> Html msg
userView username =
    div [ flex, Flex.row, smallMargin, style "border-top" "solid
gray 1px", Flex.alignItemsCenter, largeFont ]
        [ i [ class "fa-solid fa-user", style "margin-right"
"2em", smallMargin ] []
        , a [ href "/profile" ] [ text username ]
        ]
```

APPENDIX F

Page Navigation

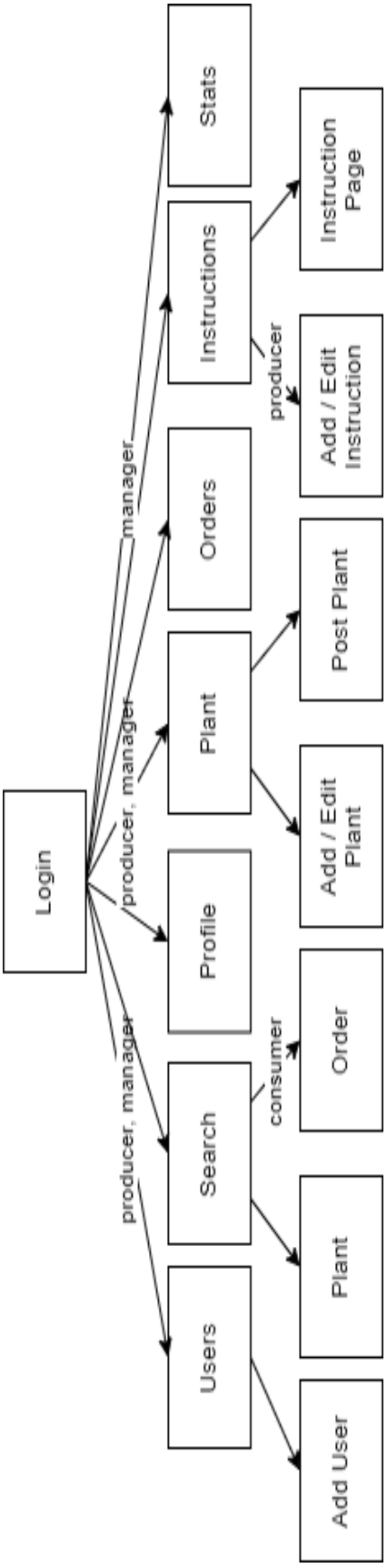

Figure E.1 – Page navigation diagram