**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 4](#_Toc104161070)

[INTRODUCTION 5](#_Toc104161071)

[1 REQUIREMENTS OF THE INFORMATIONAL SYSTEM 6](#_Toc104161072)

[2 INFORMATIONAL SYSTEM ARCHITECTURE 10](#_Toc104161073)

[2.1 High-level overview 10](#_Toc104161074)

[2.2 Business logic layer architecture 10](#_Toc104161075)

[2.3 Presentation layer architecture 11](#_Toc104161076)

[3 MODELING OF THE DOMAIN OF INFORMATIONAL SYSTEM 12](#_Toc104161077)

[4 USED TECHNOLOGIES AND SOFTWARE 17](#_Toc104161078)

[5 STRUCTURE OF THE APPLICATION 18](#_Toc104161079)

[5.1 Backend architecture 18](#_Toc104161080)

[5.2 Frontend architecture 20](#_Toc104161081)

[6 DATABASE CREATION 21](#_Toc104161082)

[6.1 Object definitions 21](#_Toc104161083)

[6.2 Accesses to database objects 24](#_Toc104161084)

[7 SQL QUERIES FOR USER TASKS 27](#_Toc104161085)

[8 APPLICATION IMPLEMENTATION 30](#_Toc104161086)

[8.1 Backend 30](#_Toc104161087)

[8.2 Frontend 32](#_Toc104161088)

[9 USER GUIDE WITH ILLUSTRATIONS 36](#_Toc104161089)

[9.1 Consumer 36](#_Toc104161090)

[9.2 Producer 40](#_Toc104161091)

[9.3 Manager 45](#_Toc104161092)

[CONCLUSIONS 46](#_Toc104161093)

[REFERENCES 47](#_Toc104161094)

[APPENDIX A Database tables creation 48](#_Toc104161095)

[APPENDIX B Database objects creation 52](#_Toc104161096)

[APPENDIX C Access Grants 71](#_Toc104161097)

[APPENDIX D Database Diagram 73](#_Toc104161098)

[APPENDIX E Shared frontend modules 74](#_Toc104161099)

[APPENDIX F Page Navigation 79](#_Toc104161100)

# 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 |  |
| С13 | 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.

1. 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.

1. 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.

1. 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.

1. Task Z2 is solved with the alter role [4] though the following query

ALTER ROLE session\_user WITH ENCRYPTED password 'NewPassword';

1. Task C13 is solved with the following query

INSERT INTO plant\_delivery(order\_id, delivery\_tracking\_number) VALUES(@orderId, @trackingNumber);

1. 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.
* С6 – 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.

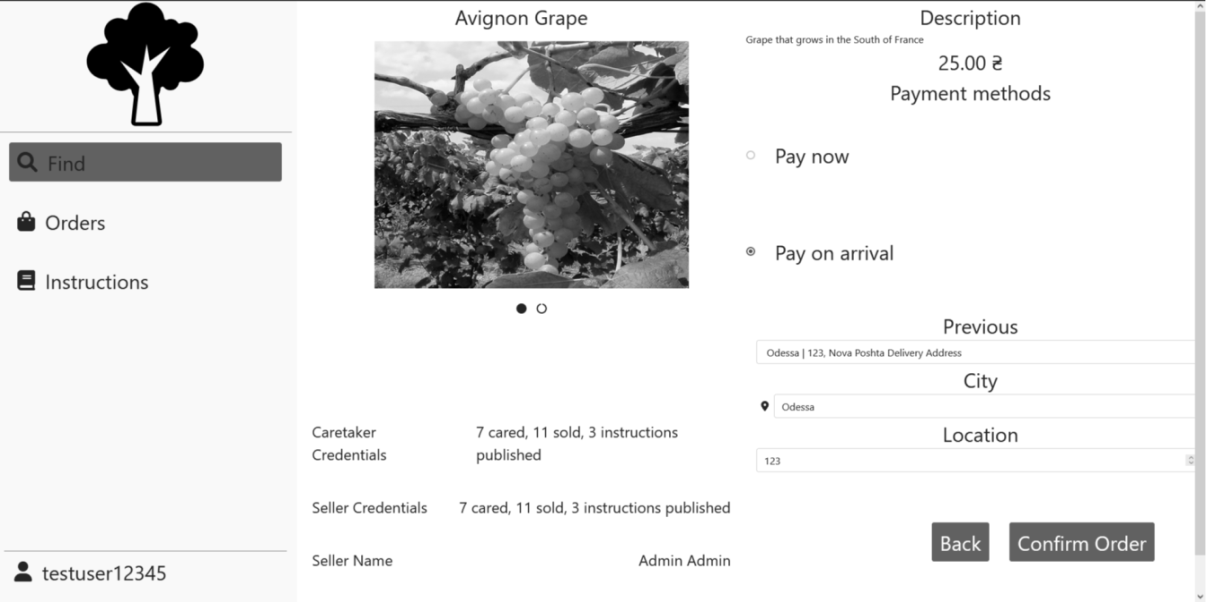


Figure 9.4- Order page

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.

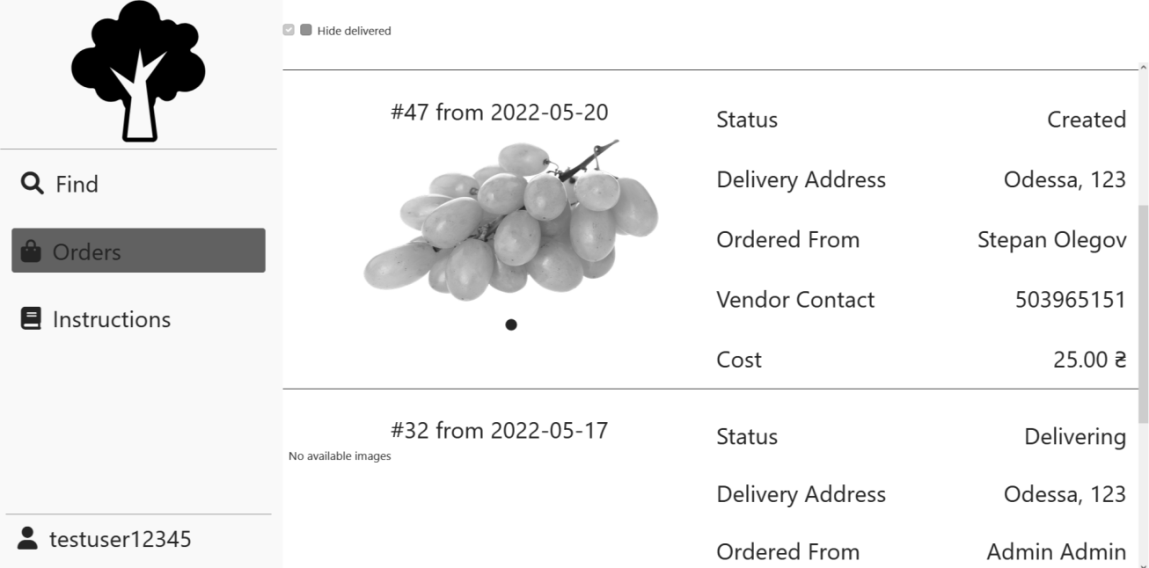


Figure 9.5 – Consumer Orders page

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.

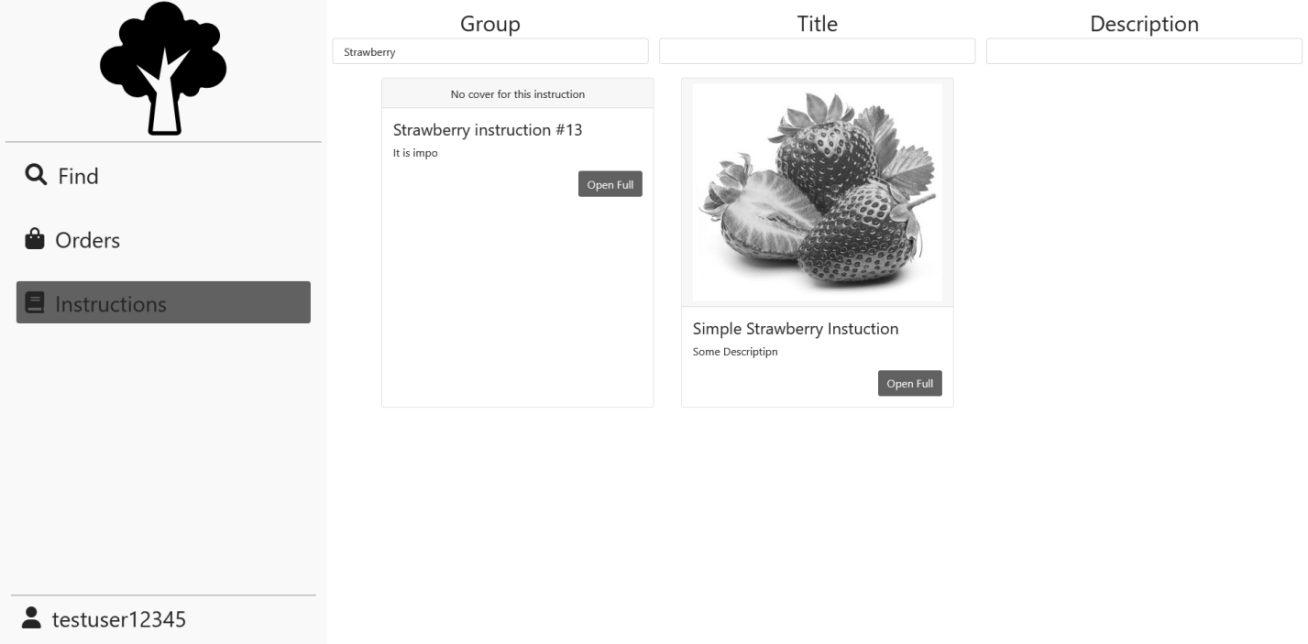


Figure 9.6 – Instructions page

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.

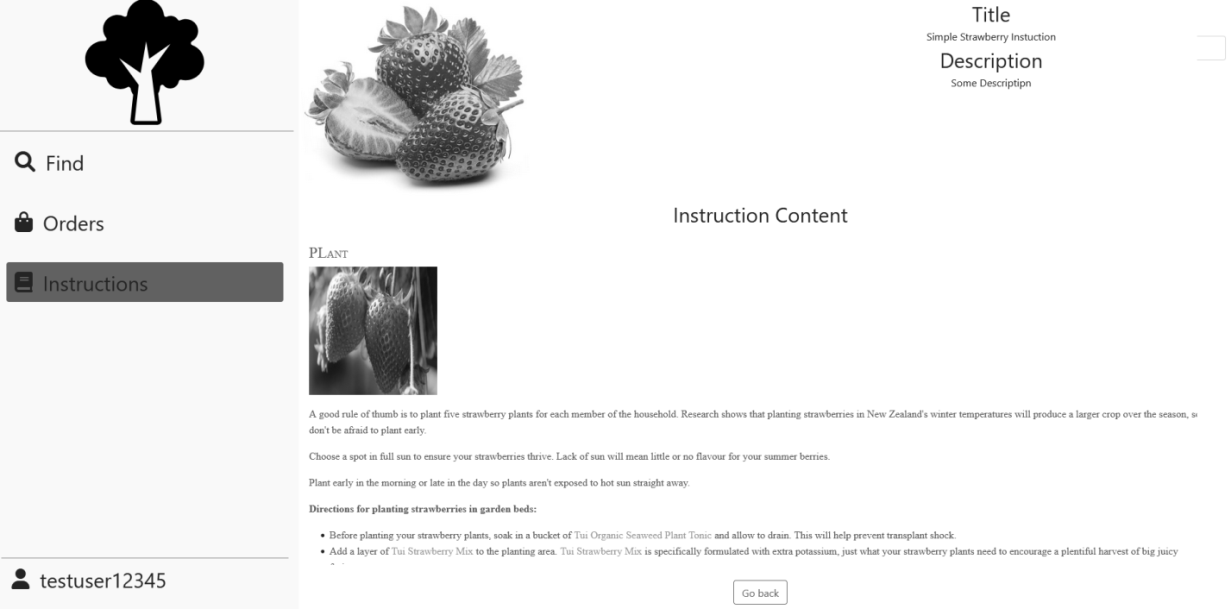


Figure 9.7 – Instruction page

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.

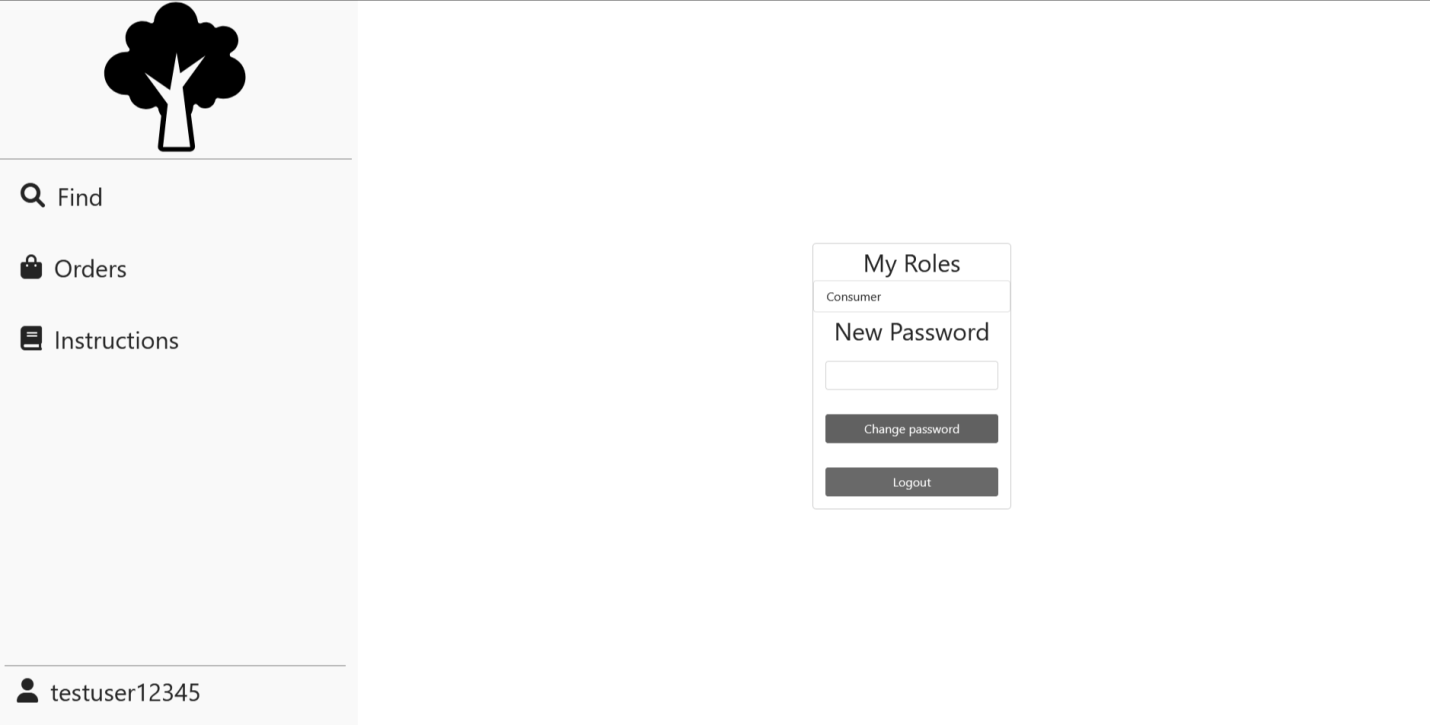


Figure 9.8 – Profile page

## 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.

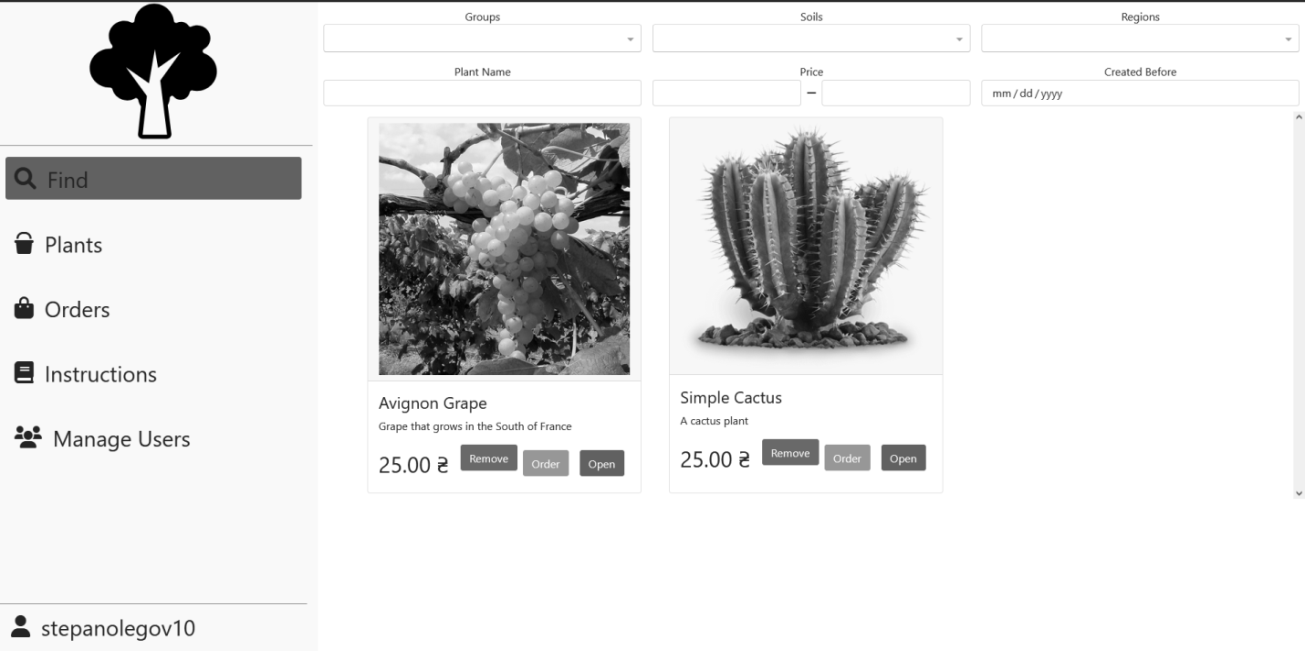


Figure 9.9 – Producer Search page

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.

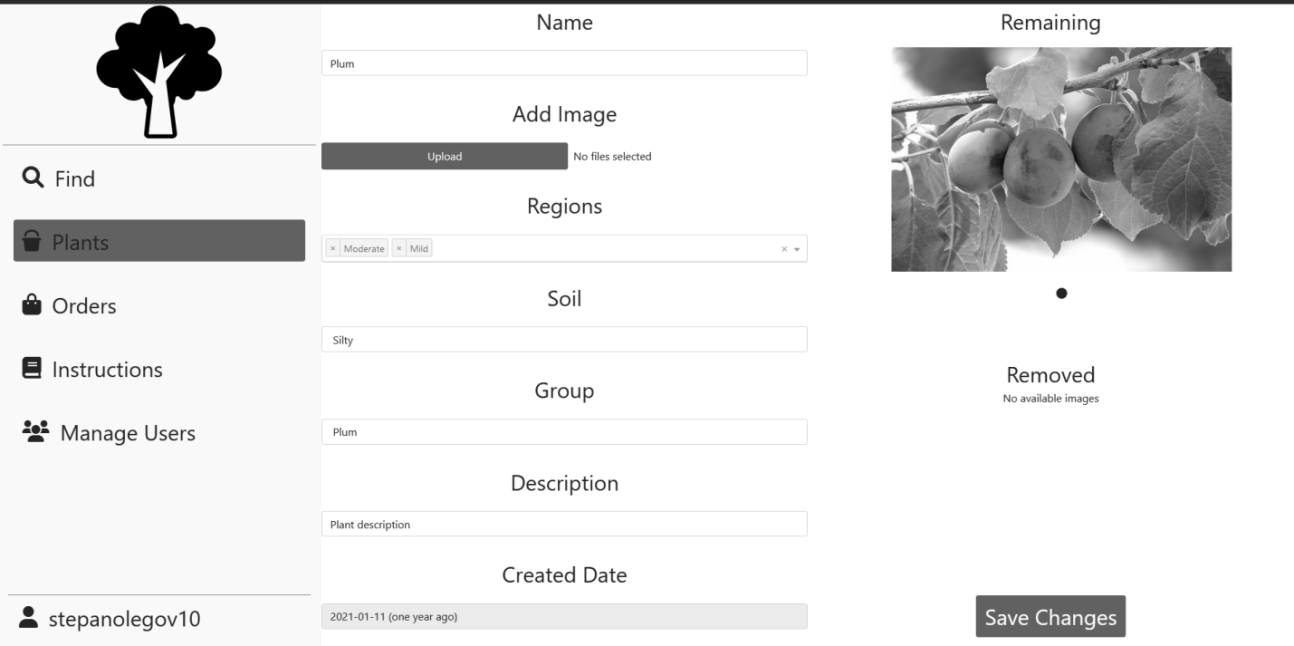


Figure 9.12 – Edit plant

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.

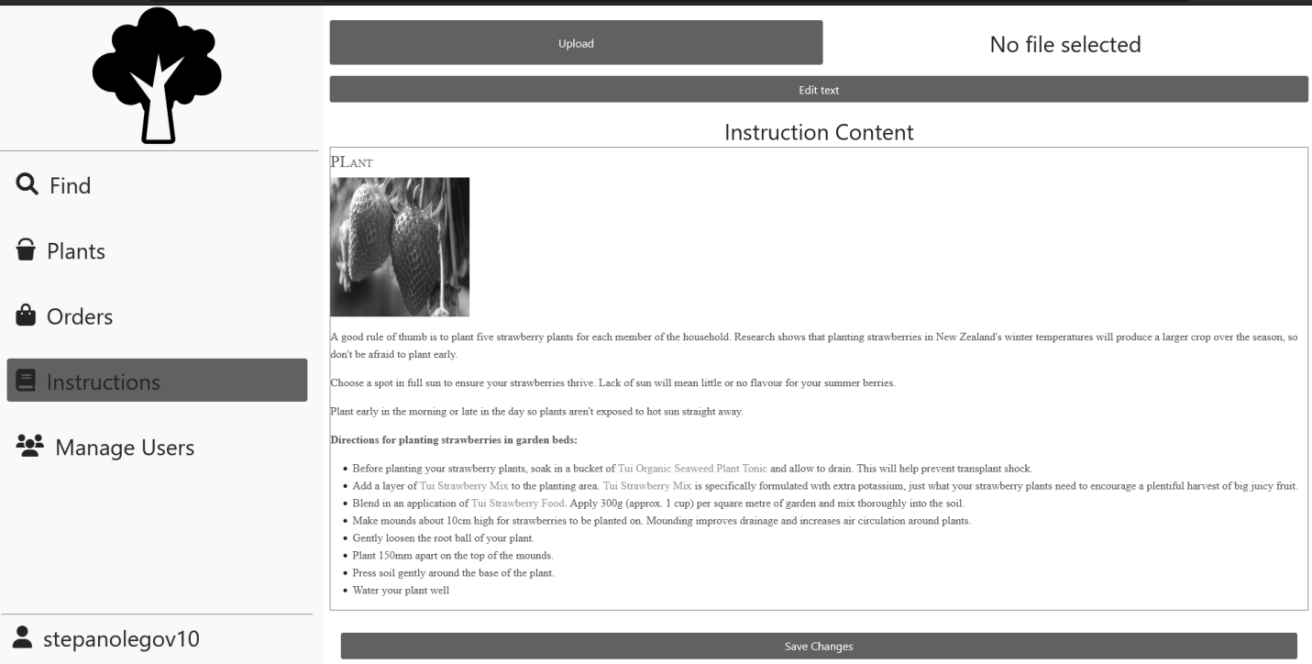


Figure 9.14 – Edit instruction

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.

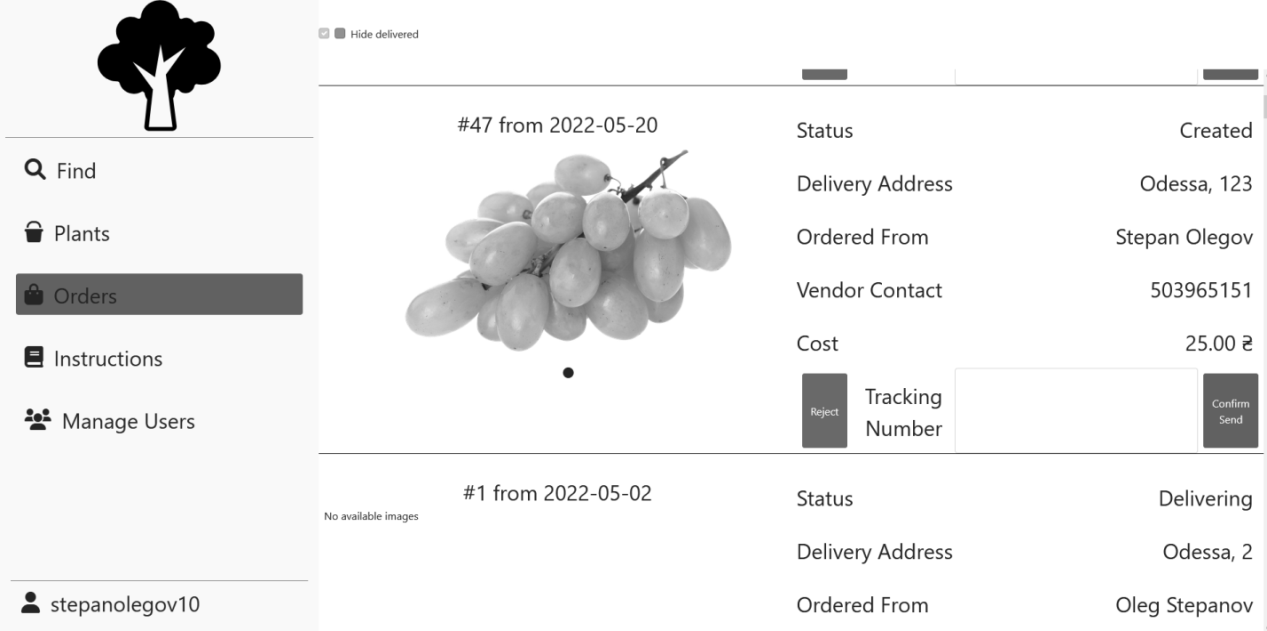


Figure 9.15 – Producer Orders page

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.

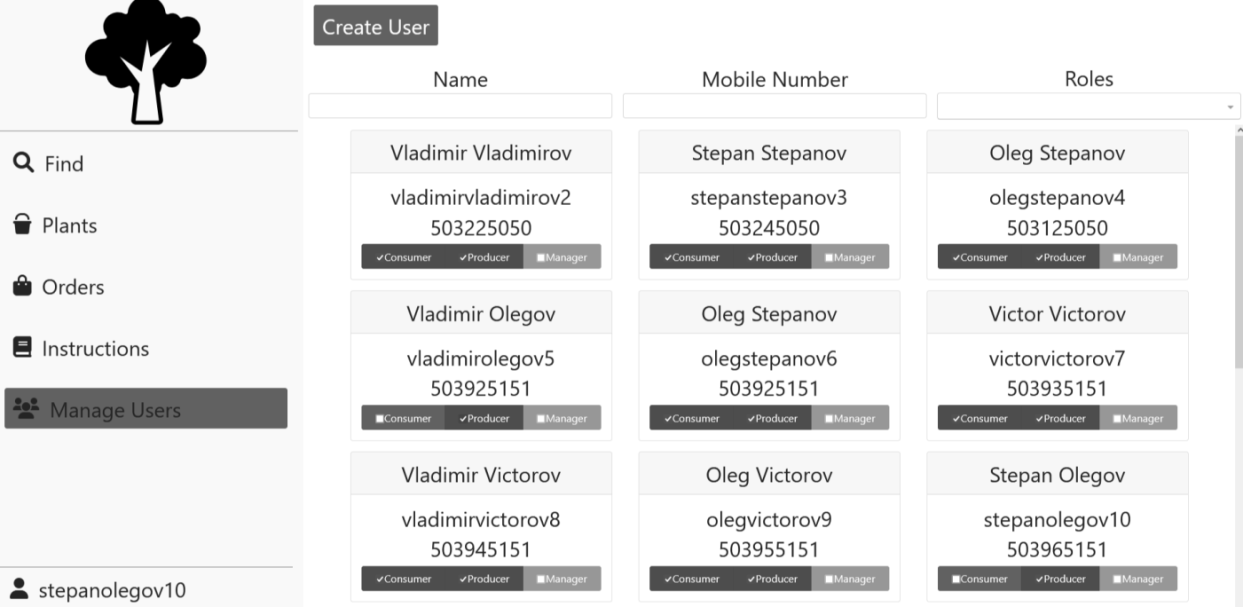


Figure 9.16 – Users page

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.

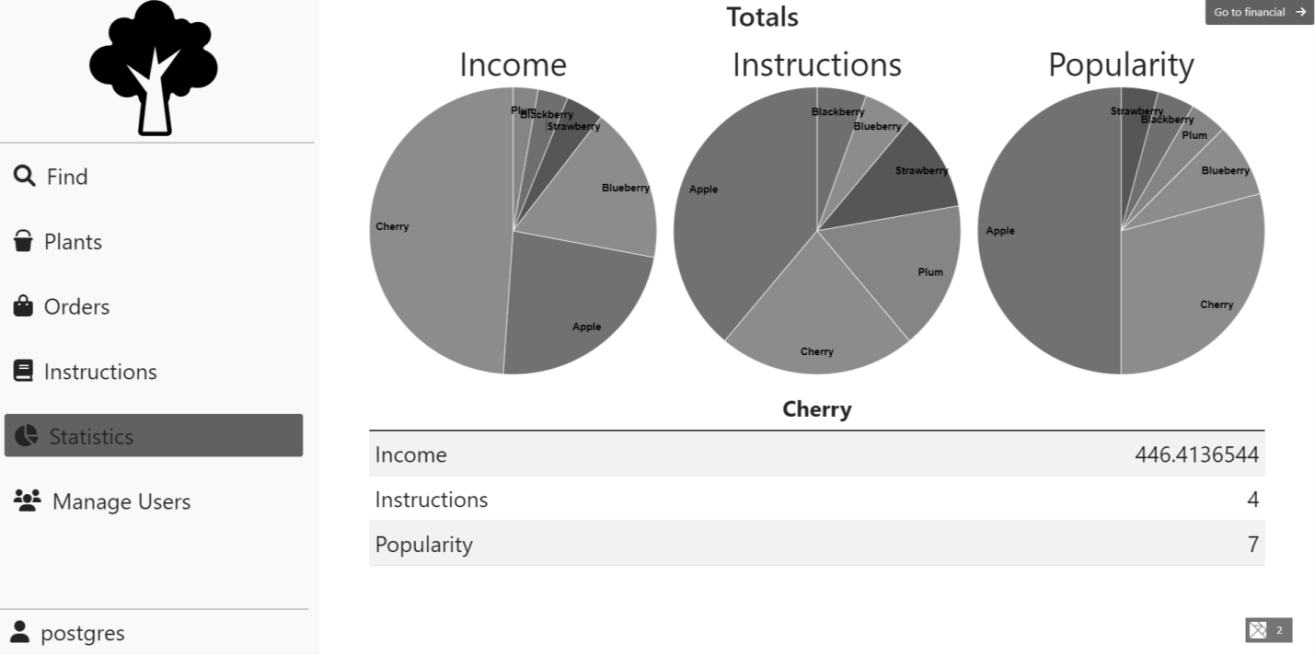


Figure 9.18 – Total statistics page

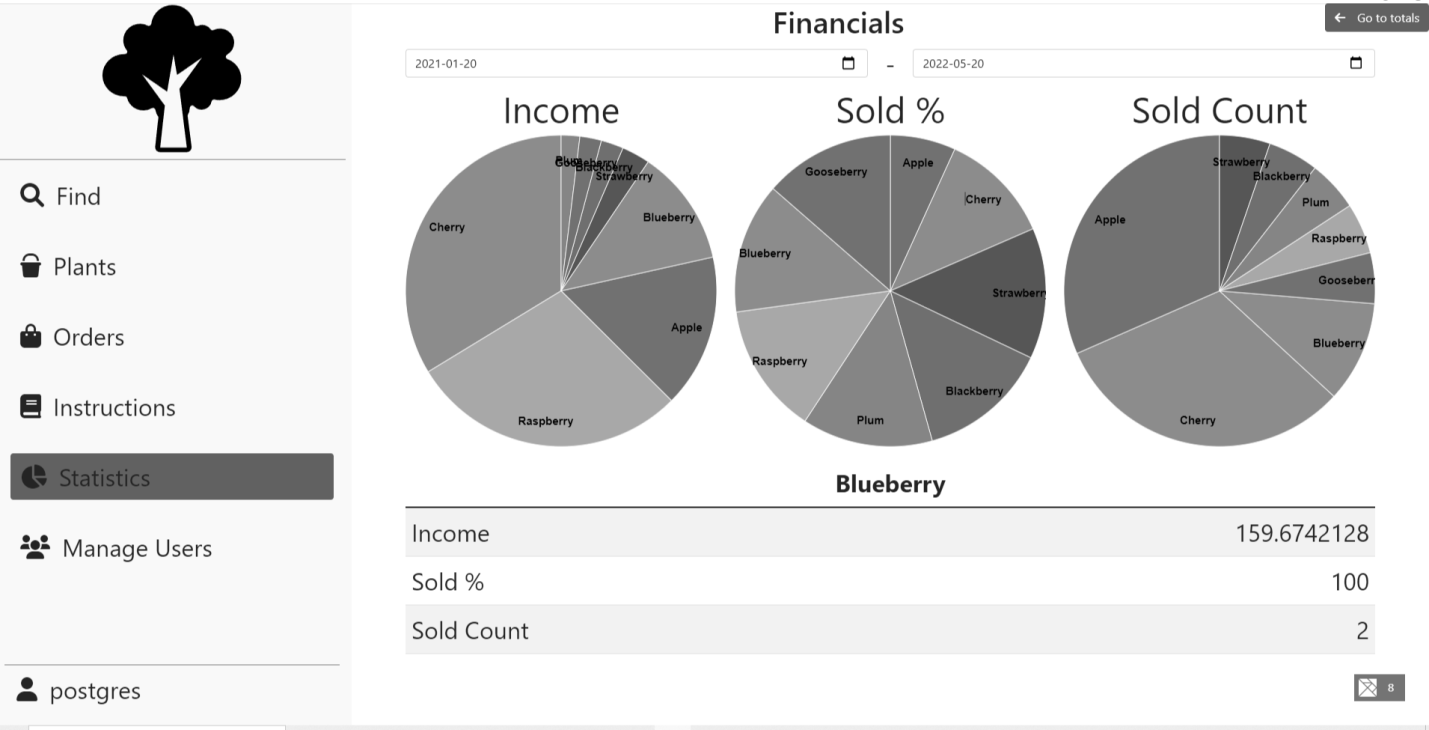


Figure 9.19 – Financial statistics page

# 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

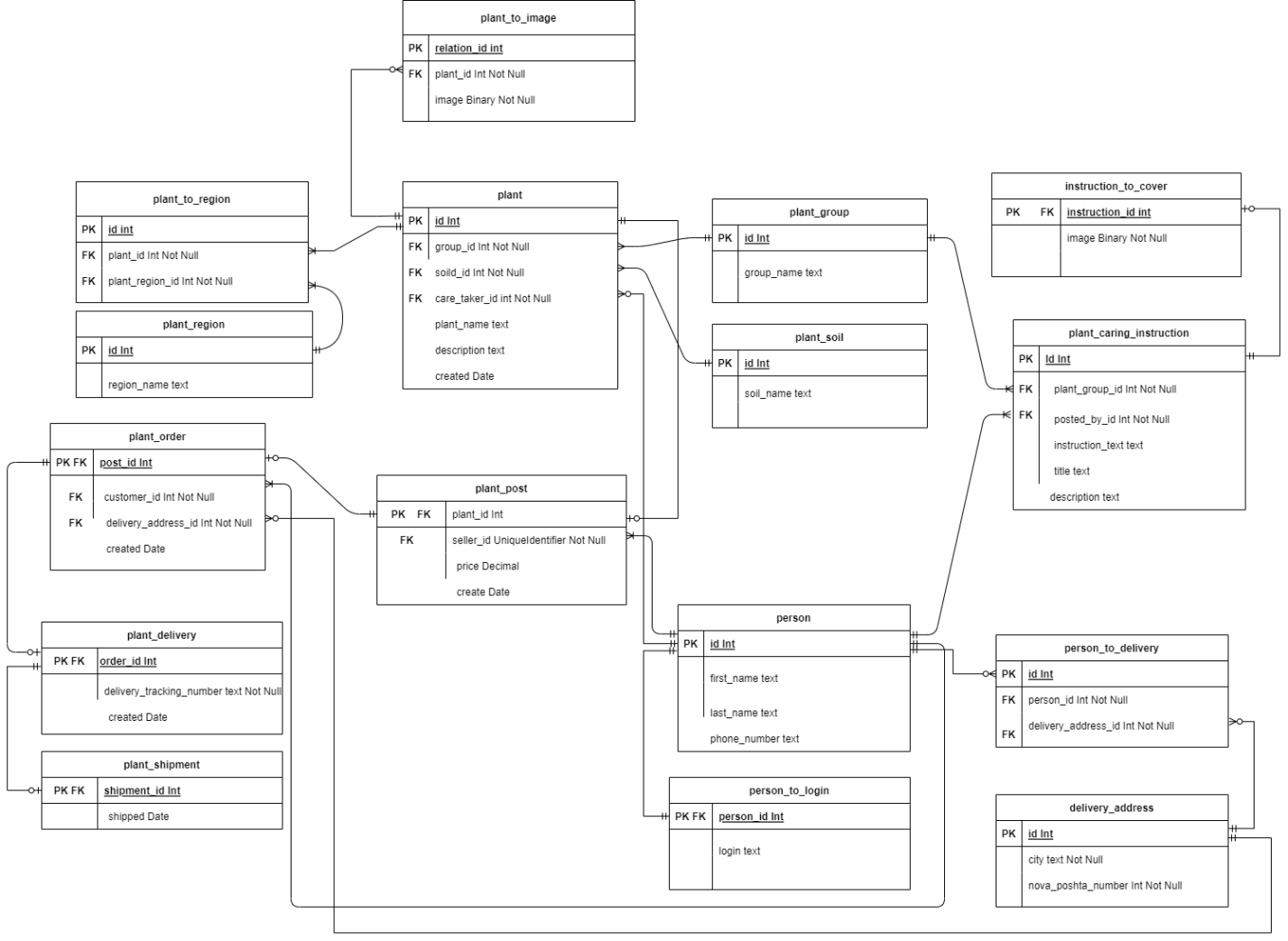


Figure D.1 – Entity relationship diagram of the plants database.

# 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