**CONTENTS**

[DEFINITIONS AND TERMS 4](#_Toc103862921)

[INTRODUCTION 5](#_Toc103862922)

[1 REQUIREMENTS OF THE INFORMATIONAL SYSTEM 6](#_Toc103862923)

[2 INFORMATIONAL SYSTEM ARCHITECTURE 11](#_Toc103862924)

[2.1 High-level overview 11](#_Toc103862925)

[2.2 Business logic layer architecture 11](#_Toc103862926)

[2.3 Presentation layer architecture 12](#_Toc103862927)

[3 MODELING OF THE DOMAIN OF INFORMATIONAL SYSTEM 13](#_Toc103862928)

[4 USED TECHNOLOGIES AND SOFTWARE 18](#_Toc103862929)

[5 STRUCTURE OF THE APPLICATION 19](#_Toc103862930)

[5.1 Backend architecture 19](#_Toc103862931)

[5.2 Frontend architecture 21](#_Toc103862932)

[6 DATABASE CREATION 22](#_Toc103862933)

[7 SQL QUERIES FOR USER GOALS 28](#_Toc103862934)

[8 APPLICATION IMPLEMENTATION 30](#_Toc103862935)

[8.1 Backend 30](#_Toc103862936)

[8.2 Frontend 32](#_Toc103862937)

[9 USER GUIDE WITH ILLUSTRATIONS 35](#_Toc103862938)

[9.1 Consumer 35](#_Toc103862939)

[9.2 Producer 35](#_Toc103862940)

[9.3 Manager 35](#_Toc103862941)

[CONCLUSIONS 36](#_Toc103862942)

[REFERENCES 37](#_Toc103862943)

[APPENDIX A Database tables creation 38](#_Toc103862944)

[APPENDIX B Database objects creation 39](#_Toc103862945)

[APPENDIX C Access Grants 40](#_Toc103862946)

[APPENDIX D Database Diagram 41](#_Toc103862947)

# 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 goals 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 goals of the business, the goals 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 goals as well as correspondence of input and output data related to them.

Table 1.1 – User Goals

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Number | Goal | Explanation | Input | Output |
|  | Consumer, Producer, Manager | | | |
| Z1 | Access the system. |  | Login  Password | Session |
| Z2 | Update your Password |  | New Password | New Session |
|  | Consumer, Producer | | | |
| A1 | Search for plants that can be ordered. | Consumers have this goal to be able to order plants. Producers have this goal 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. |  | 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 |

Continuation of Table 1.1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| C7 | Find users. | This allows managers to manage producers and producers to manage consumers. | Name  Phone Number | Users:  Name  Phone Number  Roles |
| C8 | Invite users. |  | Login  Roles  Email  Name  Phone Number | User created and email with temporary password send. |
| C9 | Update user roles. | 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 |

Continuation of Table 1.1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Manager | | | |
| D1 | See popularity for plants based on their group. |  |  | Plant Groups:  Income  Stock Number  Instructions |
| D2 | See financial info for plant based on their group. |  | Time Range | Plant Groups:  Income  Sales Number  Sold Percent |

# 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.



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 |  |
| “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 |

There are three types of relationships between relations:

* 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 then their connecting relation is presented.

The database is in the Boyce-Codd normal form. Its diagram may 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 local 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. 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. 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 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 MVC 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 databases objects and permissions that are being granted for them are laid out. SQL statements used to create objects can be found in Appendix A for tables, Appendix B for other objects and Appendix C for grants.

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 plant
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.

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 |
| 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 |
| 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 GOALS

* Z2, Update your Password.

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

* A1, Search for plants that can be ordered.

SELECT \* FROM search\_plant(@plantName, @lowerPrice, @topPrice, @lastDate, @groupIds, @soilIds, @regionIds;

* A2, Search for instructions for plants.

SELECT \* FROM search\_instructions(@GroupId, @Title, @ Description);

* A3, See detailed information for posted plant.

SELECT \* FROM plant\_post\_v WHERE id = @postId;

* B1, Order plant.

SELECT \* FROM place\_order(@postId, @city, @postNumber);

To accomplish this goal a trigger has been created order\_set\_customer that would set current user as the one that ordered plant.

* B2, See previously used addresses on order.

SELECT \* FROM current\_user\_addresses;

To accomplish this goal a trigger order\_store\_used\_address has been created that would store used addresses. Its source code can be found in Appendix B.

* B3, Confirm order to be delivered.

CALL confirm\_received(@deliveryId);

* C1, Find plants that are being prepared for post.

SELECT id, plant\_name, description, is\_mine FROM plants\_v;

* C2, Edit plant information.

call edit\_plant(@PlantId, @Name, @Description, @Regions, @SoilId, @GroupId, @RemovedImages, @NewImages);

To accomplish this goal plant\_prevent\_update\_of\_posted trigger were created that would prevent anyone from updating plant information after it had been posted.

* C3, Create plant.

SELECT create\_plant(@Name, @Description, @Regions, @SoilId, @GroupId, @Created, @Pictures);

To accomplish this goal a trigger plant\_set\_poster has been created that would mark created plants caretaker with current user.

* C4, See plant prepared for sale.

SELECT \* FROM prepared\_for\_post\_v WHERE id = @plantId;

* C5, Post plant for sale.

SELECT \* FROM post\_plant(@plantId, @price);

To accomplish this goal a trigger post\_set\_poster has been created that would mark created posts seller as current user.

* C6, Create Instruction.

SELECT \* FROM create\_instruction(@GroupId, @Text, @Title, @Description, @CoverImage);

To accomplish this goal a trigger instruction\_set\_poster has been created that would mark instruction author as current user.

* C7, Find users.

SELECT \* FROM search\_users(@FullName, @Contact, @Roles::UserRoles[]);

* C8, Invite users.

CALL create\_user(@Login, @Password, @Roles::UserRoles[], @FirstName, @LastName, @PhoneNumber);

* C9, Update user roles.

CALL add\_user\_to\_group(@login, @role::UserRoles);

CALL remove\_user\_from\_group(@login, @role::UserRoles);

* C10, Remove post.

DELETE FROM plant\_post WHERE plant\_id = @postId;

To accomplish this goal a trigger post\_prevent\_unlawfull\_delete has been created that would enforce access rules for removal.

* C11, Update instruction.

CALL edit\_instruction(@InstructionId, @GroupId, @Text, @Title, @Description, @CoverImage);

* C12, Reject order.

DELETE FROM plant\_order WHERE post\_id = @orderId;

To accomplish this goal a trigger order\_prevent\_unlawfull\_delete has been created that would enforce access rules for removal.

* C13, Start Order delivery.

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

* D1, See popularity for plants based on their group.

SELECT \* FROM plant\_stats\_v;

* D2, See financial info for plant based on their group.

SELECT \* FROM get\_financial(@from, @to);

# 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

The implementation for plants 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 main function of Login page.

main : Program D.Value Model Msg

main =

baseApplication

{ init = init

, view = view

, update = update

, subscriptions = subscriptions

}

Listing 8.7 – login page main

# 9 USER GUIDE WITH ILLUSTRATIONS

## 9.1 Consumer

## 9.2 Producer

## 9.3 Manager

# 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

* Lock A. ASP.NET Core in Action. 2018 y.
* PostgreSQL: Documentation: 9.4: Constraints - <https://www.postgresql.org/docs/9.4/ddl-constraints.html>.
* Feldman R. Elm in Action. 2020 y.
* .NET Generic Host in ASP.NET Core: Microsoft Docs - <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/host/generic-host>.
* Common web application architectures: Microsoft Docs - <https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures>.
* Teplyakov S. .NET Software architecture patterns. 2015 y.
* Richter J. CLR via C#. 2012 y.
* PostgreSQL: Documentation: 9.4: Alter Role - <https://www.postgresql.org/docs/9.4/sql-alterrole.html>.
* Elm Core Documentation - <https://package.elm-lang.org/packages/elm/core/latest>.

# APPENDIX A Database tables creation

//Paste 001\_Initial here

# APPENDIX B Database objects creation

//paste 003 to 013

# APPENDIX C Access Grants

//paste 014\_Grants here

# APPENDIX D Database Diagram

