Technical University of Cluj-Napoca

Programming Techniques

Assignment 4

Food Delivery Service Manager

Teacher: Ioan Salomie

Laboratory Teacher: Cristina Pop

Student: Anitei Cristian-Daniel

Group: 30424

1) Objective

The objective of this assignment is to design and implement a food delivery management system for a catering company. The client can order products from the company's menu. The system should have three types of users that log in using a username and a password: administrator, regular employee, and client.

The administrator can:

• Import the initial set of products which will populate the menu from a .csv file.
• Manage the products from the menu: add/delete/modify products and create new products composed of several products (an example of composed product could be named "daily menu 1" composed of a soup, a steak, a garnish, and a dessert).
• Generate reports about the performed orders.

The client can:

• Register and use the registered username and password to log in within the system.
• View the list of products from the menu.
• Search for products based on one or multiple criteria such as keyword (e.g., "soup"), rating, number of calories/proteins/fats/sodium/prices.
• Create an order consisting of several products – for each order, the date and time will be persisted and a bill will be generated that will list the ordered products and the total price. of the order.

The employee is notified each time a new order is performed by a client so that it can prepare the delivery of the ordered dishes.

2) Problem analysis

Modeling

Our application has several models that must be persisted using serialization. The first model we have is the menu item. The menu item has two variants: **the base menu item** and **the composite menu item.**

The all menu items share a title, a price, and a rating. Besides that, the base menu item will also have extra attributes like fats, proteins, calories, and sodium. The composite menu item is still a menu item and should be treated like such, having a price, title, and rating. Besides that, instead of modeling nutritional values, it will hold a collection of items (composite or basic). This is chosen to model a business requirement like creating the daily menu or special offers.

Besides menu items, we need to model the information about an order. The order will have an uniquely identifiable ID, a uniquely identifiable email for the client who placed that order and the date and time it was placed. Besides that, the order will also hold the total price of the ordered items. The order will have associated to it a list of menu items, which will be the items the client has selected to be ordered.

The third main model of our application is the User. A user is modeled by a unique email and a secret password, as well as by a type. The users of our application can be of three types:

- Client: can place orders and will receive a bill for those orders
- Admin: can manage the delivery service, modify the products, and generate statistics
- Employee: will receive orders from the client and will prepare the meals. He can also mark orders as complete.

Based on these models, our system will implement all the functionality required in the objective, and a few soft features which make the user experience a pleasant one.

Use cases

The use cases of our applications differ depending on the users of our application. This being an application which behaves differently based on the user that logs in.

The following use case diagram will display all the actions our user can take while using the warehouse order management application:

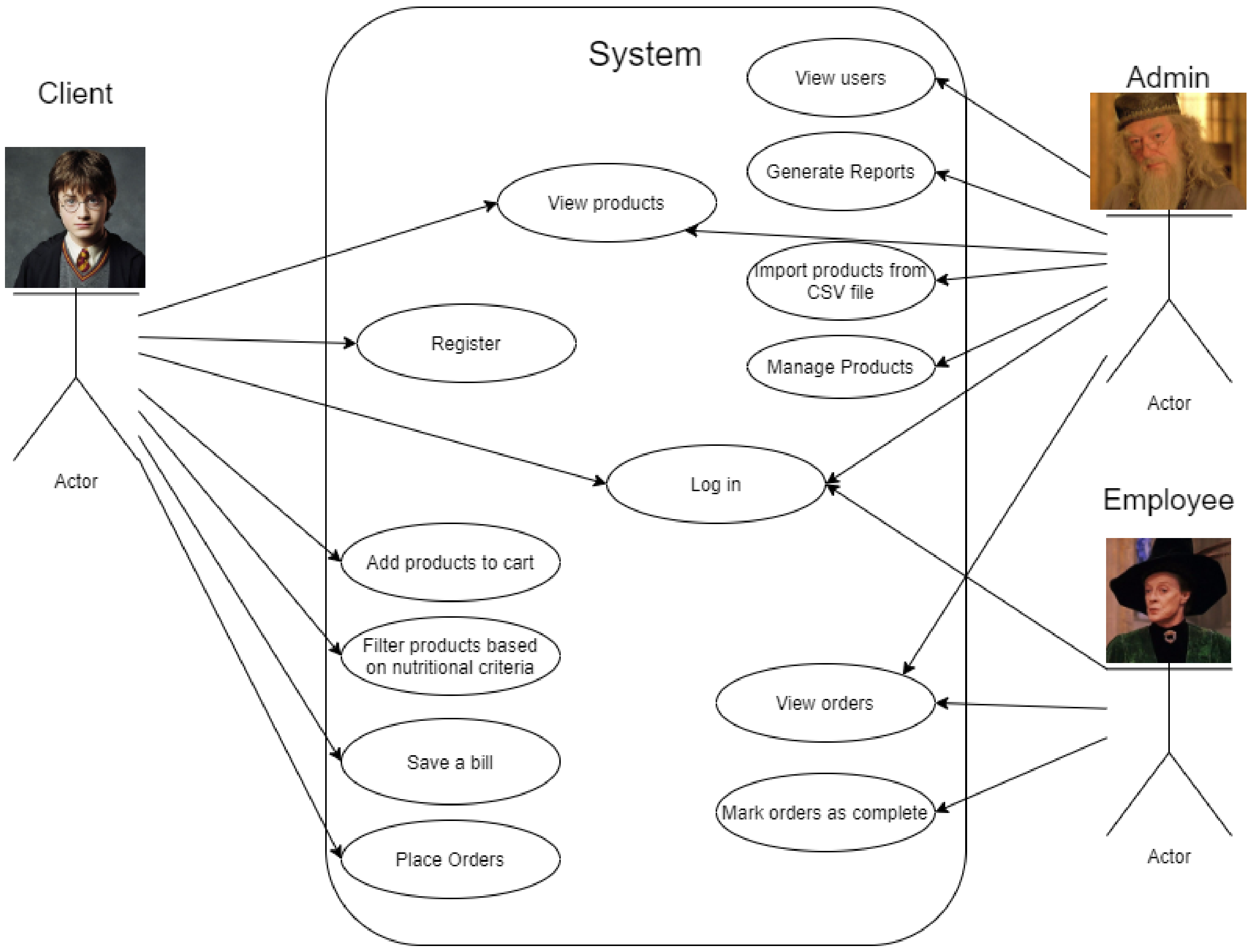


Figure 1. Use case diagram.

The client can register and view the products, add a few to cart, filter them by nutritional criteria like for example calories and place order, saving the bill to his desktop.

The employee can view place orders and can mark them as complete once they have finished preparing them.

The administrator (or more briefly: admin) can view all orders, products, and users. Can generate reports on all of them and can manage the products available to the client. He can add, edit, or remove both composite and base products from the product pool and can also import base products from a previously saved csv file.

All three actors that attempt to use our app must log in first in order to be recognized and granted access to their respective functionalities. Without their saved credentials, none of the actors can access their respective windows. When logging in, each user can have different use case scenarios. The scenarios of every actor are developed in the following figures:

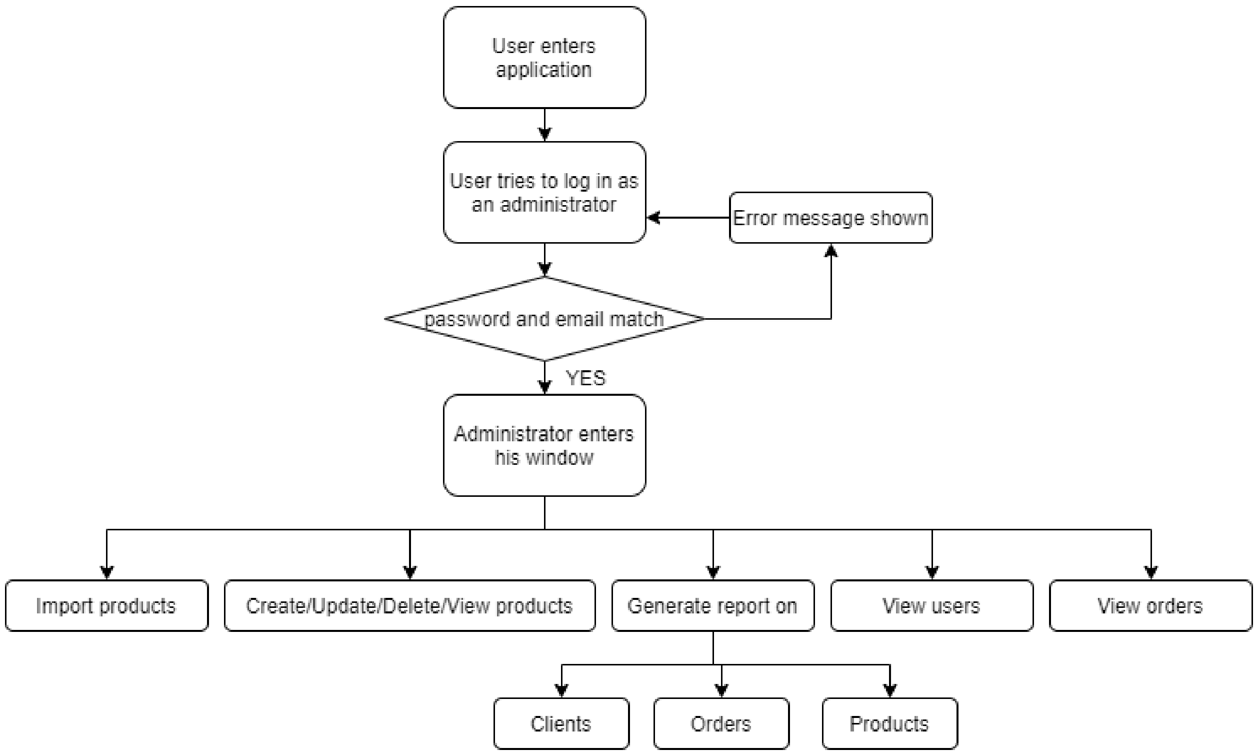


Figure 3. Administrators use case scenario.

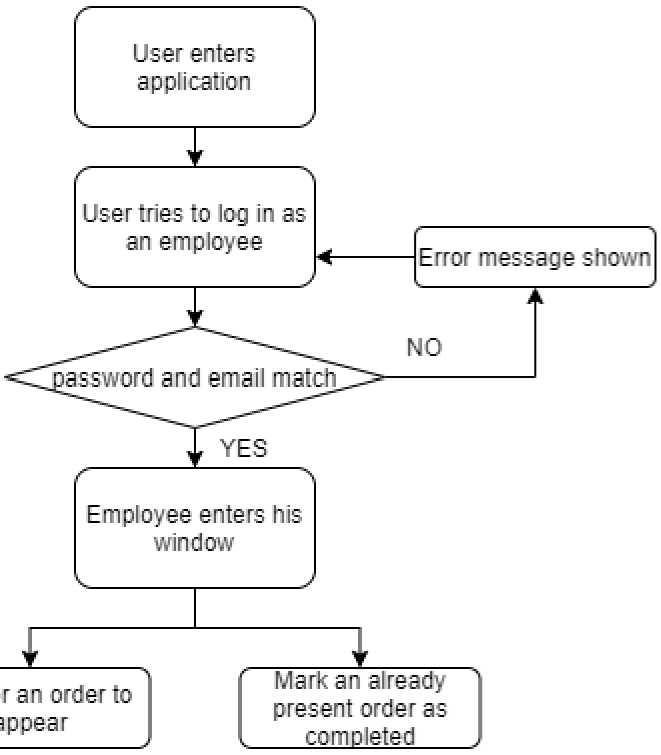

User enters application
User tries to log in as an employee
Error message shown
NO
password and email match
YES
Employee enters his window
Wait for an order to appear
Mark an already present order as completed

Figure 2. Employees use case scenario.

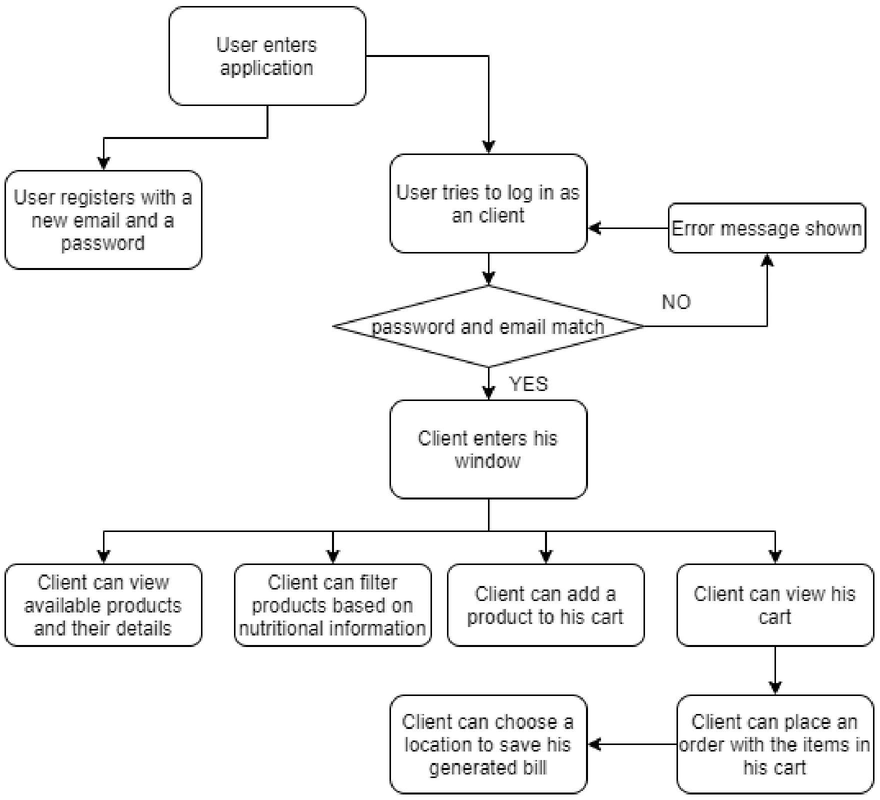

User enters application
User registers with a new email and a password
User tries to log in as an client
Error message shown
NO
password and email match
YES
Client enters his window
Client can view available products and their details
Client can filter products based on nutritional information
Client can add a product to his cart
Client can view his cart
Client can choose a location to save his generated bill
Client can place an order with the items in his cart

Figure 4. Clients use case scenario.

3) Application Design

The application is structured in an object-oriented manner, using a layered architectural pattern for the structure of its packages. This layered architecture allows for a separation between the graphical user interface, the application logic and the data access and management classes. User requests are propagated through the layers of the architecture, from the GUI down to the serialized data stored in files.

• Presentation

The presentation package contains all our front-end classes which compose the graphical user interface our user sees. In this package, pairs of fxml objects and their controller classes are stored and used to show the various lists, controls and dialogues which enable the users to interact with our application.

• Business Logic

The business logic layer holds all the data management logic required for our front-end to work properly. When the presentation layer requires certain data, it can call the business layer for that data, and the business layer will deliver it, either by requesting it from the data access layer (i.e. deserializing stored data) or by computing it on already existing data.

• Data Access

The data access offers a convenient abstractization from the file requesting and processing of serialized data, as well as the parsing of csv files.

To the left in figure 5, you can see the overall package schema.

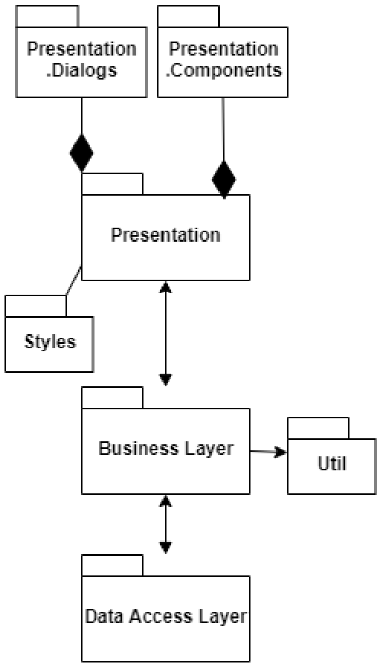


Figure 5. Package diagram

Data Access

The data access layer is composed of two main classes that are unrelated to each other but provide a crucial layer for different parts of the application. The two main functionalities of this data access layer is to perform serialization and deserialization on the models of our application. The main purpose of serialization is to take a snapshot of the data in our application and print it to a file in a manner that is later readable and reconstructible without losing data. The other attribute of the data access layer is to provide the functionality of parsing objects from a csv file. That functionality is implemented by the CSV parser class which exposes a static method for parsing and returning a list of BaseProduct.

The Serializator will be a generic class that will be able to serialize any array of type T if the type of that array is a class that implements the Serializable interface.

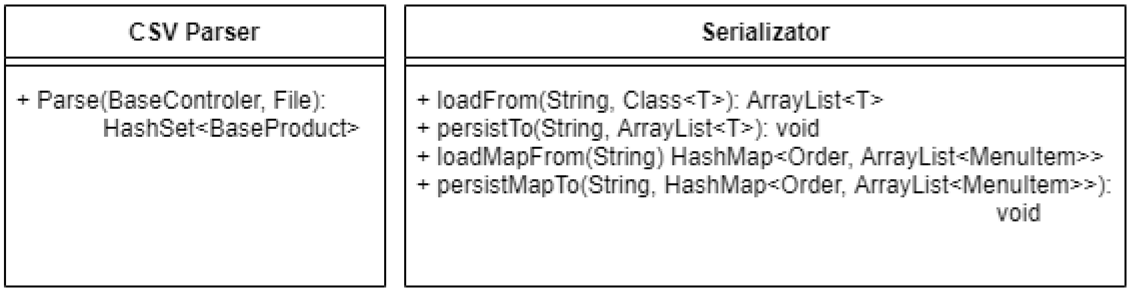

CSV Parser
+ Parse(BaseControler, File): HashSet<BaseProduct>
Serializator
+ loadFrom(String, Class<T>): ArrayList<T>
+ persistTo(String, ArrayList<T>): void
+ loadMapFrom(String) HashMap<Order, ArrayList<MenuItem>>
+ persistMapTo(String, HashMap<Order, ArrayList<MenuItem>>): void

Figure 6. Data Access diagram.

Business Logic Layer

The business logic layer represents the main logic of our application. The main class of this layer is the DeliveryService class. This class will manage all the data storage, manipulation, retrieval, and persistence, supplying that data to the presentation layer which will be presented in the next section. The delivery service class will expose methods for accessing the sets of all the models of our app. The presentation views can request data and submit modifications on the MenuItem, Orders and users of our app. To provide the upper layer with easy access to this data, the DeliveryService has been implemented as a singleton in our app. You cannot instantiate another delivery service, and you can only make request to the existing one. The delivery service implements the IDeliveryServiceProcessing interface, which exposes the main methods for interacting with the data. Using design by contract, each method has some preconditions and postconditions specified, conditions that the implementing class must respect in order to preserve the design. Besides these design patterns, another one was used to create an inversion of control between the presentation layer and the business logic. The DeliveryService class will also implement the Observable interface and will expose a method called Subscribe, such that the presentation classes can be notified to any relevant changes. An example of this pattern being of use in this application is the EmployeeController which needs to be notified when a client creates a new order, to render that order on the employee's screen.

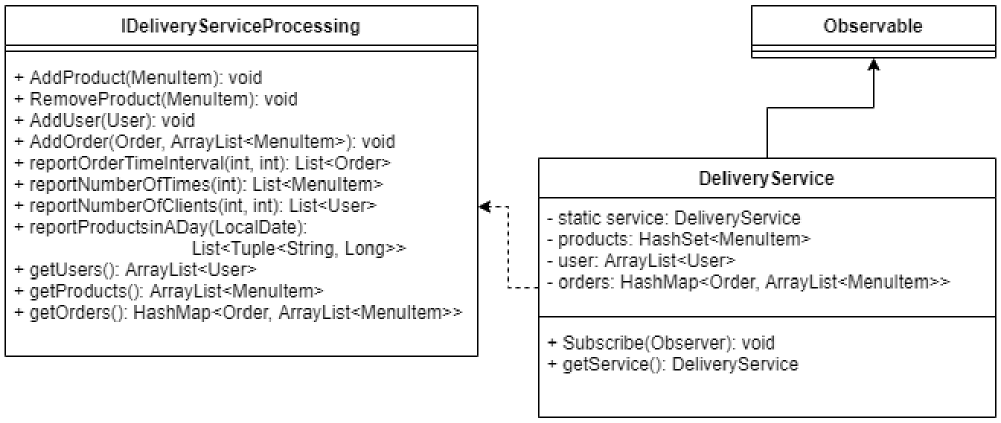

IDeliveryServiceProcessing
+ AddProduct(MenuItem): void
+ RemoveProduct(MenuItem): void
+ AddUser(User): void
+ AddOrder(Order, ArrayList<MenuItem>): void
+ reportOrderTimeInterval(int, int): List<Order>
+ reportNumberOfTimes(int): List<MenuItem>
+ reportNumberOfClients(int, int): List<User>
+ reportProductsinADay(LocalDate): List<Tuple<String, Long>>
+ getUsers(): ArrayList<User>
+ getProducts(): ArrayList<MenuItem>
+ getOrders(): HashMap<Order, ArrayList<MenuItem>>
Observable
DeliveryService
- static service: DeliveryService
- products: HashSet<MenuItem>
- user: ArrayList<User>
- orders: HashMap<Order, ArrayList<MenuItem>>
+ Subscribe(Observer): void
+ getService(): DeliveryService

Figure 7. Business Logic Delivery Service

Besides the main delivery service which encompasses most of the business layer, we also implement the model classes in this layer. The three main model classes are User, Order and MenuItem with its inheritors: BaseProduct and CompositeProduct. While the model of the user and Order are not too out of the ordinary and do not present any design challenges other than defining them and their fields, the MenuItem is a whole other story. The menu item can be both a BaseProduct and a Composite product, but they will be both stored as a menu item inside the delivery service. Also, the composite product aggregates its ancestor, menu item inside its structure, therefore creating an interesting type recursion where one could create a composite product which has in its composition another composite product and so on. The design pattern I used to build this kind of relationship is the composite design pattern, which achieves exactly this goal. This pattern is best illustrated in the following schema:

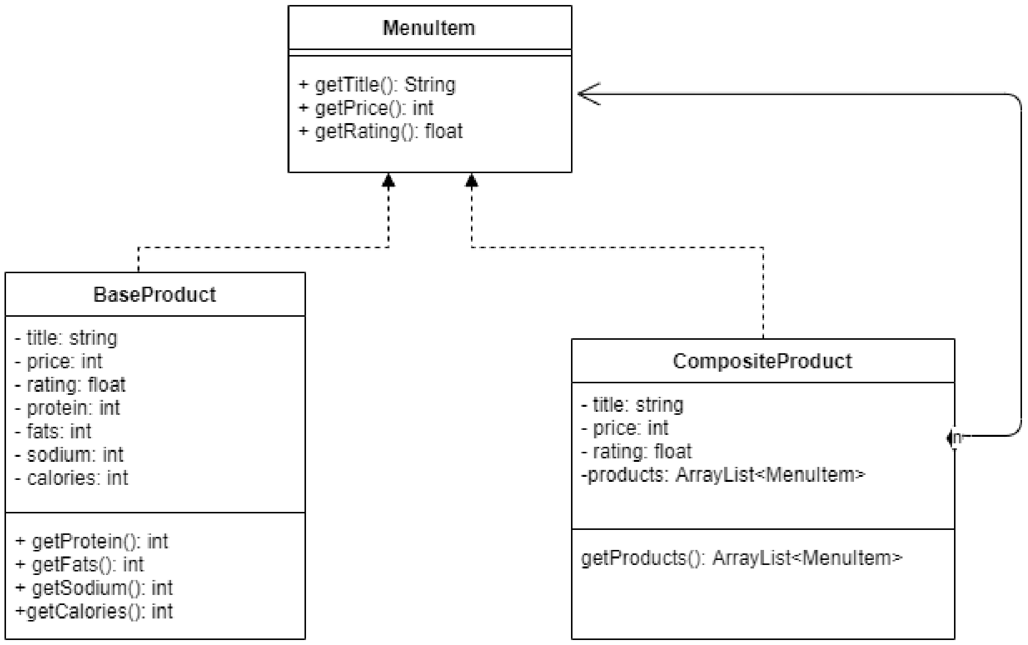


Figure 8. Menu Item Hierarchy.

And the design of the other two models:

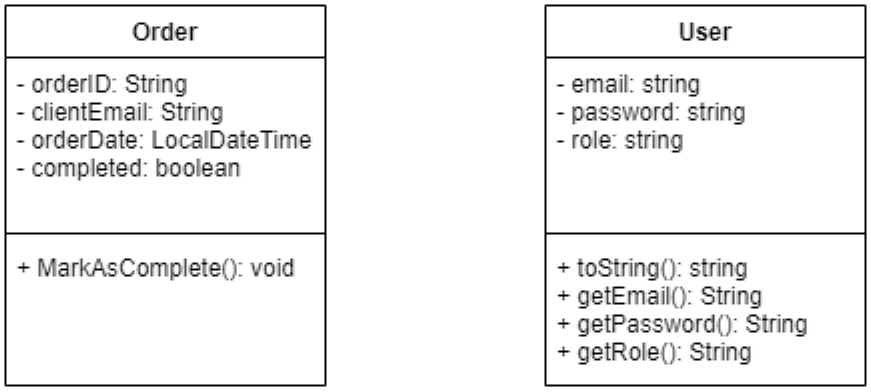


Figure 9. Order model and User model

All three models combined with our DeliveryService give us a robust backend which we can harness when we will design the presentation layer.

Presentation

The presentation is split between multiple screens. Firstly, we have the login screen which will appear when the user starts the application. When the user types of his valid credentials into the login form, a new window will open, housing the view appropriate for his role in the company. The administrator window will offer a wide range of inputs to perform the reports and the management on the products of the delivery application, as well have available three lists to view the users, orders and menu items stored in the memory of the application.

The employee will be greeted with an empty window, but once the client will make an order, that empty window will be populated with an order component, which will list the contents of the order and the information about the client who ordered it.

The client will be greeted with a list of menu items represented in the interface as cards. The cards will hold all information about a product, for example for the base product it will hold the nutritional value like calories and fats, while for a composite product it will hold a list of the menu items in its composition. We have divided these UI elements in views: ClientView, AdminView and EmployeeView, while the components which are aggregated in these views have been grouped under a package called Presentation.Components: OrderComponent, BaseProductComponent and CompositeProductComponent.

We also have a few dialogs to help with the input and confirmation of the products and orders respectively: BaseProductDialog, CompositeProductDialog and CheckoutDialog.

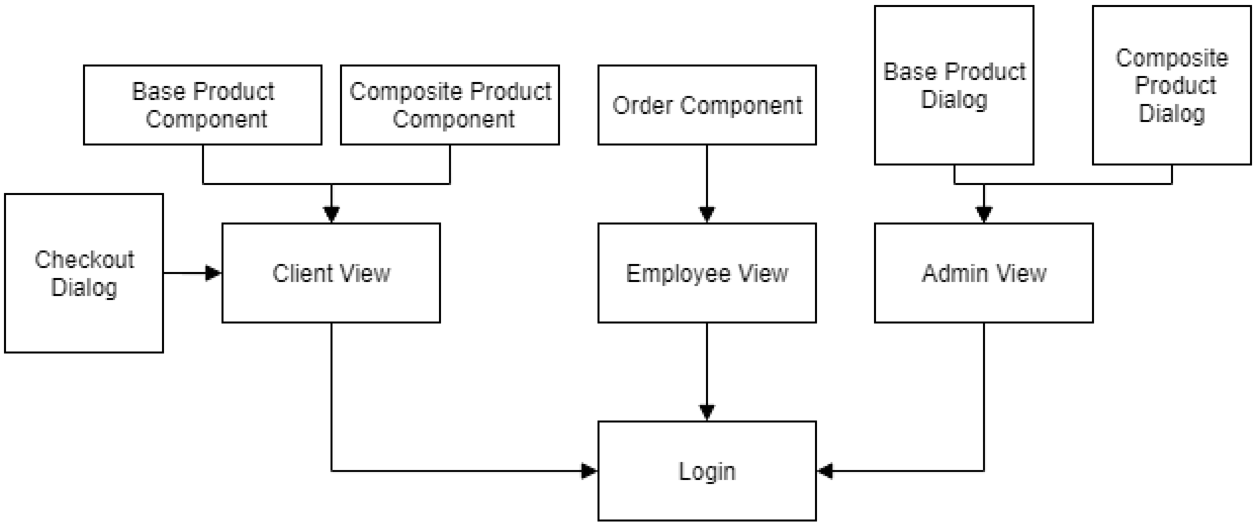


Figure 10. UI composition diagram

Utils

The utils package holds some utility classes that I had the need for during development, like the Tuple class which is nothing more than a pair of two values whose types are specified through generics.

Styles

As before in my other projects, the styles class contains CSS files that get injected into the view to style it in a more pleasant way. In this project, like in the others before it, I have expanded those styles to extend to list views and other smaller UI elements like labels and some buttons as well.

4) Application Implementation

To orderly present the implementation, I will keep using the layered format and present how each package was implemented, but before that, what is important is the structure of the project.

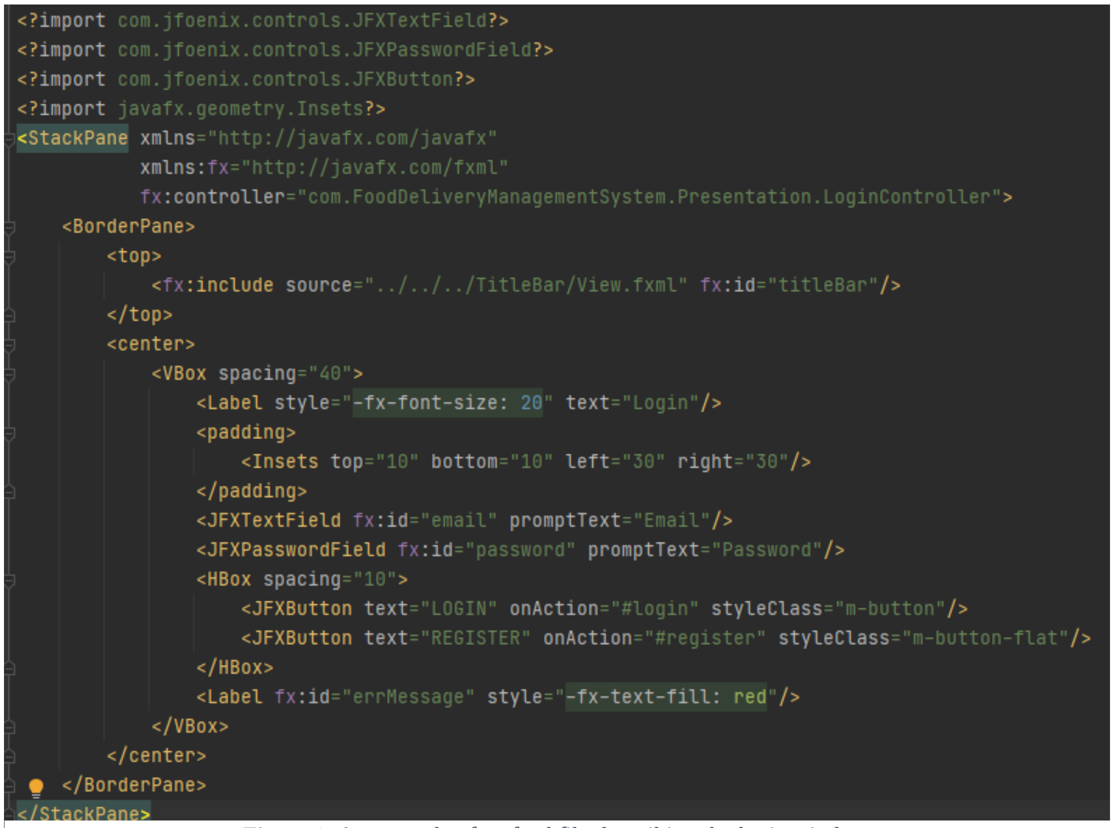
The project makes use of several packages to run. Those packages are:

- **Java FX** – is a UI framework that allows for development of more modern desktop visual applications. This was mostly used for the user interface and its connections with the controller. (1)
- **JFoenix** – is a front-end framework created for Java FX that brings with it some beautiful material styles to make our app more pleasing to the eye and create more interactive interfaces. (2)
- **FontAwesome** – is a front-end icon font which allows use of a various range of icons in the application. (5)

To easily manage these packages, the application uses Gradle (3) which is a popular tool for automating building processes and downloading and loading depended on packages. To the user, Gradle is invisible, being used only to build apps and load packages without the programmer having to manually add jars to their project. Gradle also uses the maven package repository to automatically retrieve the dependencies, making project development an ease.

These packages have been crucial to the development of the application and user interface, which we will explore now.

The user interface was realized using FXML which is a special kind of XML markup which the Java FX platform can interpret and instantiate as class objects inside our app. Developing in FXML, we do not have to write java code necessarily, but we can structure the same classes we had to instantiate manually into HTML like tags to produce our UI. While doing so, the same JavaFX classes are used like the BorderPane or the Button, but it allowed for a more intuitive view of the front end.



```
<?import com.jfoenix.controls.JFXTextField?>
<?import com.jfoenix.controls.JFXPasswordField?>
<?import com.jfoenix.controls.JFXButton?>
<?import javafx.geometry.Insets?>
<StackPane xmlns="http://javafx.com/javafx"
           xmlns:fx="http://javafx.com/fxml"
           fx:controller="com.FoodDeliveryManagementSystem.Presentation.LoginController">
    <BorderPane>
        <top>
            <fx:include source="../../../TitleBar/View.fxml" fx:id="titleBar"/>
        </top>
        <center>
            <VBox spacing="40">
                <Label style="-fx-font-size: 20" text="Login"/>
                <padding>
                    <Insets top="10" bottom="10" left="30" right="30"/>
                </padding>
                <JFXTextField fx:id="email" promptText="Email"/>
                <JFXPasswordField fx:id="password" promptText="Password"/>
                <HBox spacing="10">
                    <JFXButton text="LOGIN" onAction="#login" styleClass="m-button"/>
                    <JFXButton text="REGISTER" onAction="#register" styleClass="m-button-flat"/>
                </HBox>
                <Label fx:id="errMessage" style="-fx-text-fill: red"/>
            </VBox>
        </center>
    </BorderPane>
</StackPane>
```

Figure 1. An example of an fxml file describing the login window.

Having discussed that, here is the implmentation of our application, starting with the login window. Here an user can log in as any of the three types of accounts, or create a new account of type client.

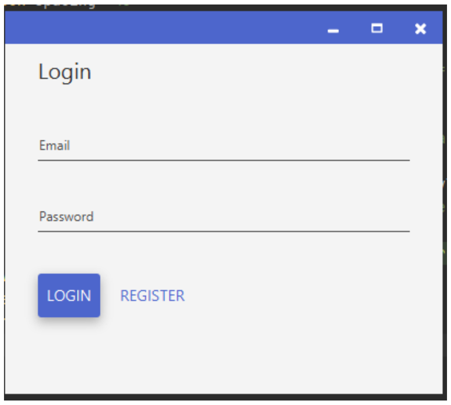
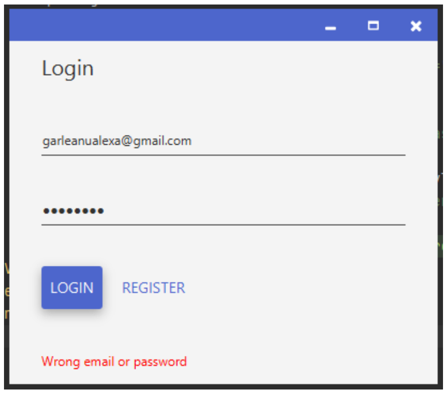


Figure 3. The log in window.



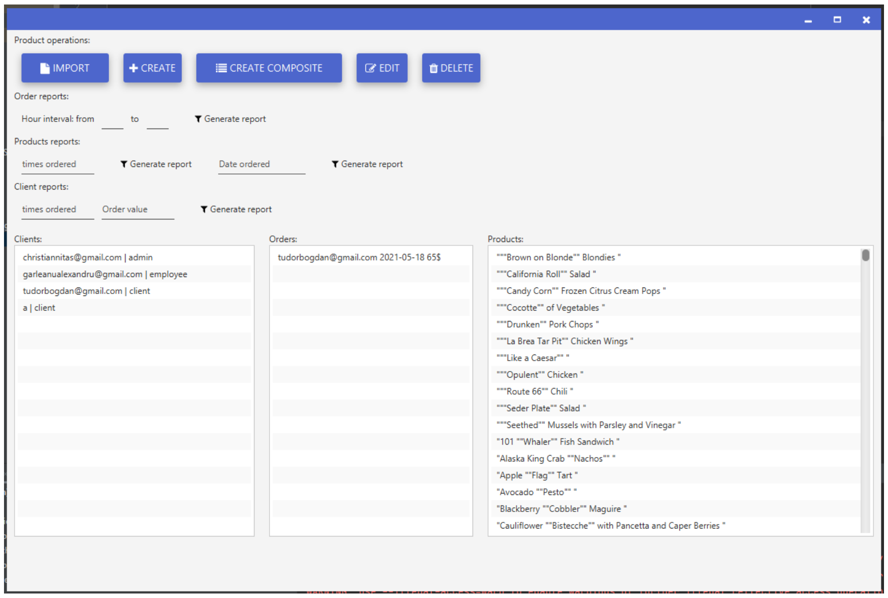
garleanualexa@gmail.com

••••••••

Wrong email or password

Figure 2. Failed log in attempt.

If the credentials are not matched to any of the saved user data, the window will display an error message. If an administrator logs in successfully, he will see the following screeen:



Clients:

christiannitas@gmail.com | admin
garleanualexandru@gmail.com | employee
tudorbogdan@gmail.com | client
a | client

Orders:

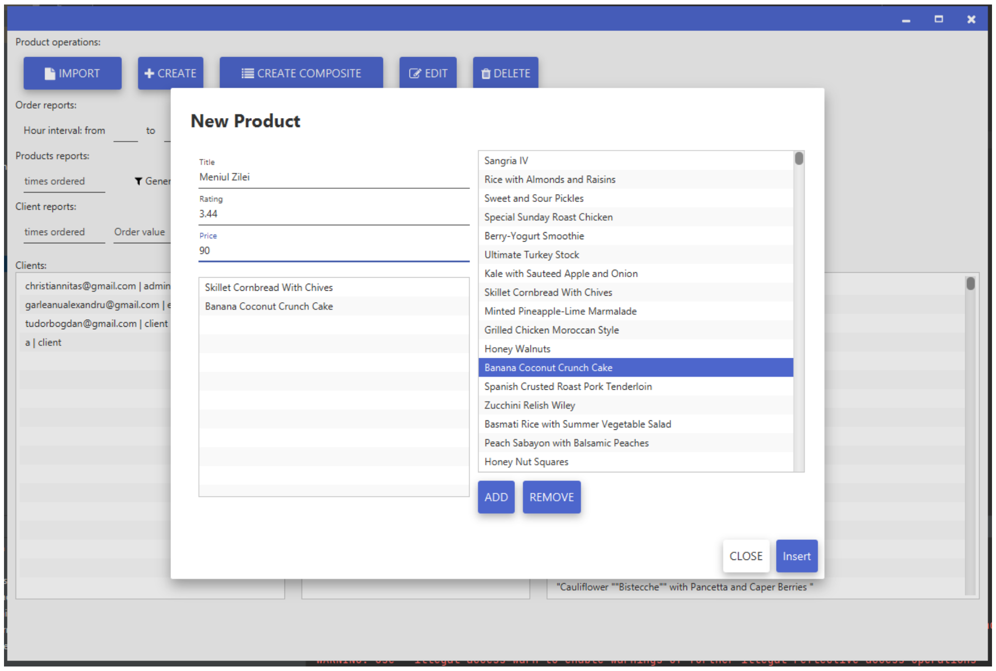
tudorbogdan@gmail.com 2021-05-18 65$

Products:

"""Brown on Blonde"" Blondies "
"""California Roll"" Salad "
"""Candy Corn"" Frozen Citrus Cream Pops "
"""Cocotte"" of Vegetables "
"""Drunken"" Pork Chops "
"""La Brea Tar Pit"" Chicken Wings "
"""Like a Caesar"" "
"""Opulent"" Chicken "
"""Route 66"" Chili "
"""Seder Plate"" Salad "
"""Seethed"" Mussels with Parsley and Vinegar "
"101 ""Whaler"" Fish Sandwich "
"Alaska King Crab ""Nachos"" "
"Apple ""Flag"" Tart "
"Avocado ""Pesto"" "
"Blackberry ""Cobbler"" Maguire "
"Cauliflower ""Bistecche"" with Pancetta and Caper Berries "

Figure 4. The administrator window.

The administrator can import products, create or edit both base and composite product and view all the models inside the app. He can as well, using the textboxes in the upper region, create new reports and save them locally.



christiannitas@gmail.com | admin
garleanualexandru@gmail.com | e
tudorbogdan@gmail.com | client
a | client

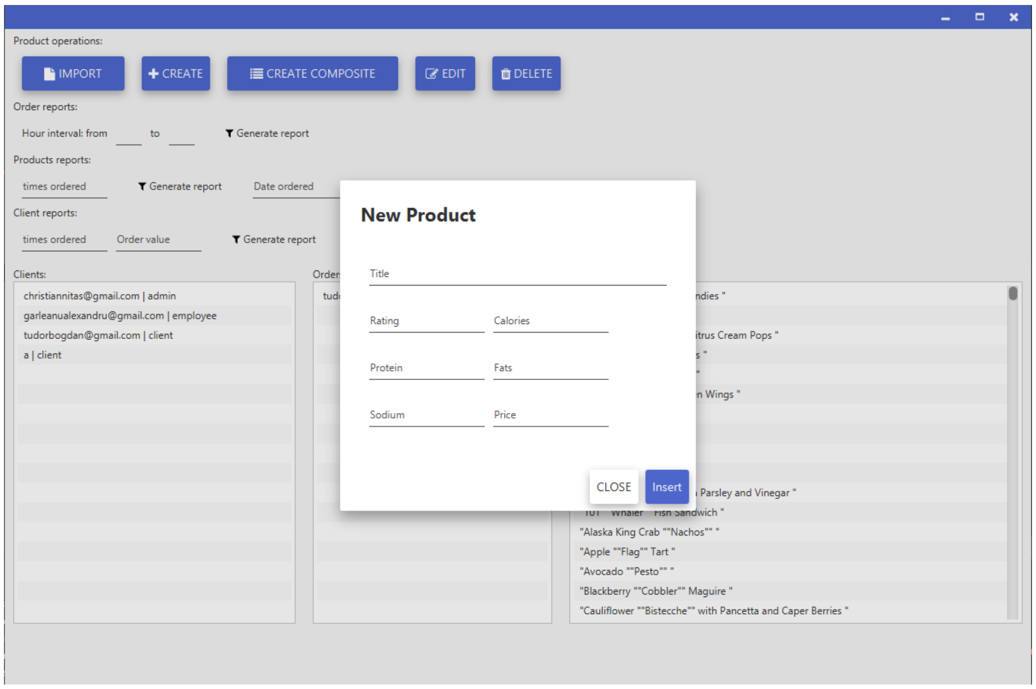
Title: Meniul Zilei
Rating: 3.44
Price: 90

Skillet Cornbread With Chives
Banana Coconut Crunch Cake

Sangria IV
Rice with Almonds and Raisins
Sweet and Sour Pickles
Special Sunday Roast Chicken
Berry-Yogurt Smoothie
Ultimate Turkey Stock
Kale with Sauteed Apple and Onion
Skillet Cornbread With Chives
Minted Pineapple-Lime Marmalade
Grilled Chicken Moroccan Style
Honey Walnuts
Banana Coconut Crunch Cake
Spanish Crusted Roast Pork Tenderloin
Zucchini Relish Wiley
Basmati Rice with Summer Vegetable Salad
Peach Sabayon with Balsamic Peaches
Honey Nut Squares

"Cauliflower ""Bistecche"" with Pancetta and Caper Berries "

Figure 6. The create new composite dialog.



christiannitas@gmail.com | admin
garleanualexandru@gmail.com | employee
tudorbogdan@gmail.com | client
a | client

ndies "
itrus Cream Pops "
s "
n Wings "
Parsley and Vinegar "
"101 ""Whaler"" Fish Sandwich "
"Alaska King Crab ""Nachos"" "
"Apple ""Flag"" Tart "
"Avocado ""Pesto"" "
"Blackberry ""Cobbler"" Maguire "
"Cauliflower ""Bistecche"" with Pancetta and Caper Berries "

Figure 5. The create base product dialog.

The administrator can also choose a file to import into the project, as well as to generate reports and save them locally to another file:

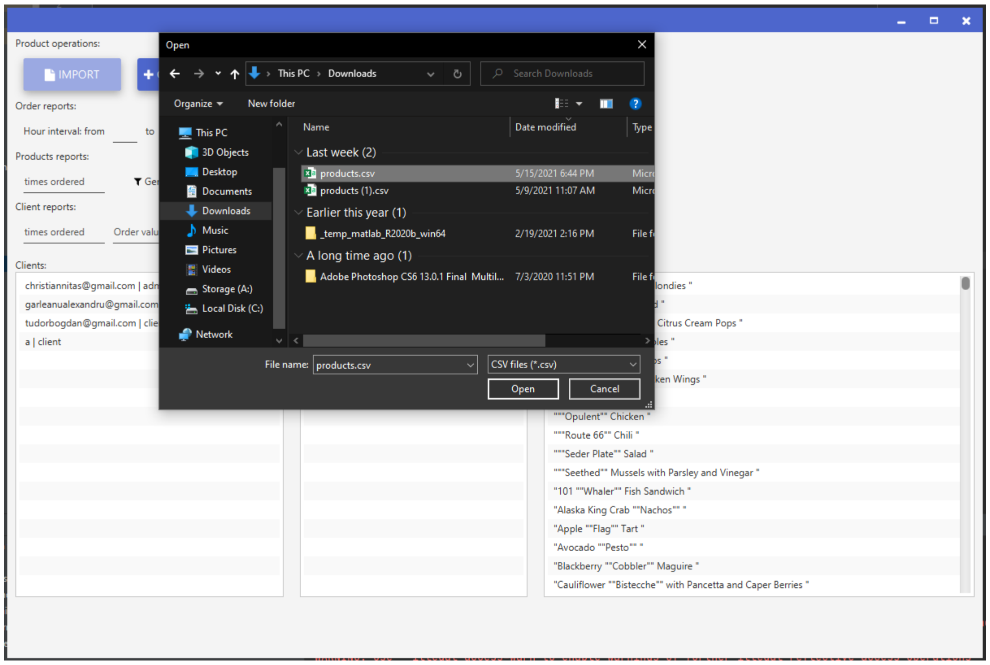
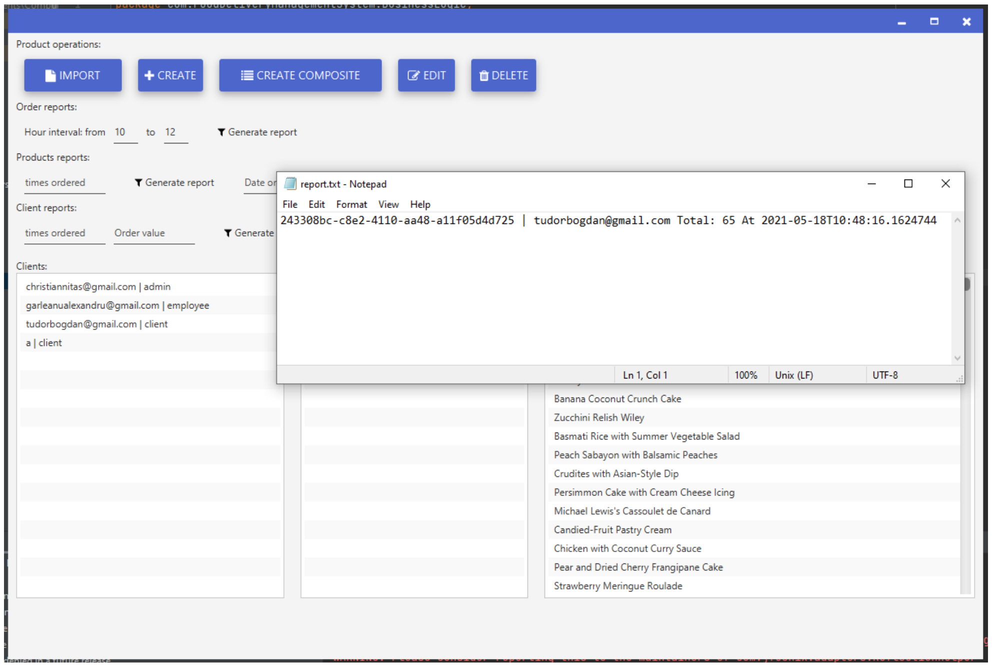


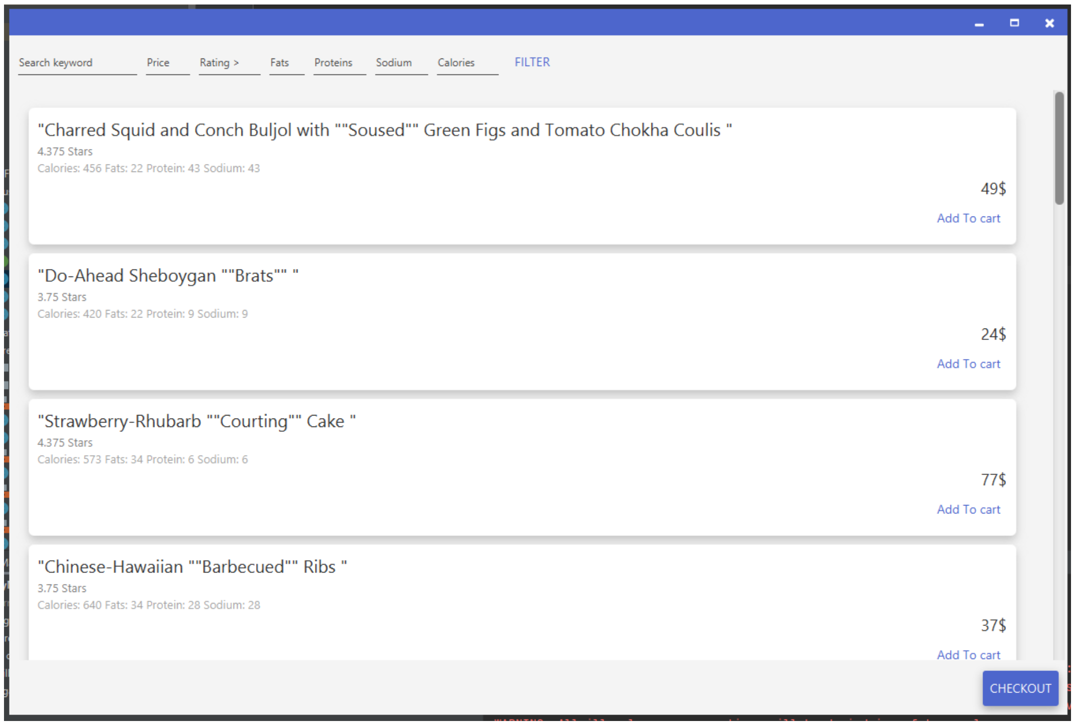
Figure 8. Import from CSV.


Product operations:
IMPORT
CREATE
CREATE COMPOSITE
EDIT
DELETE
Order reports:
Hour interval: from 10 to 12
Generate report
Products reports:
times ordered
Generate report
Date or
Client reports:
times ordered Order value
Generate
Clients:
christiannitas@gmail.com | admin
garleanualexandru@gmail.com | employee
tudorbogdan@gmail.com | client
a | client
report.txt - Notepad
File Edit Format View Help
Ln 1, Col 1 100% Unix (LF) UTF-8
Banana Coconut Crunch Cake
Zucchini Relish Wiley
Basmati Rice with Summer Vegetable Salad
Peach Sabayon with Balsamic Peaches
Crudites with Asian-Style Dip
Persimmon Cake with Cream Cheese Icing
Michael Lewis's Cassoulet de Canard
Candied-Fruit Pastry Cream
Chicken with Coconut Curry Sauce
Pear and Dried Cherry Frangipane Cake
Strawberry Meringue Roulade

```
243308bc-c8e2-4110-aa48-a11f05d4d725 | tudorbogdan@gmail.com Total: 65 At 2021-05-18T10:48:16.1624744
```

Figure 7. Create a report of the orders.

When a client enters his credentials, he will be greeted with the following window, displaying the available products:



Search keyword | Price | Rating > | Fats | Proteins | Sodium | Calories | FILTER

"Charred Squid and Conch Buljol with ""Soused"" Green Figs and Tomato Chokha Coulis "
4.375 Stars
Calories: 456 Fats: 22 Protein: 43 Sodium: 43
49$
Add To cart

"Do-Ahead Sheboygan ""Brats"" "
3.75 Stars
Calories: 420 Fats: 22 Protein: 9 Sodium: 9
24$
Add To cart

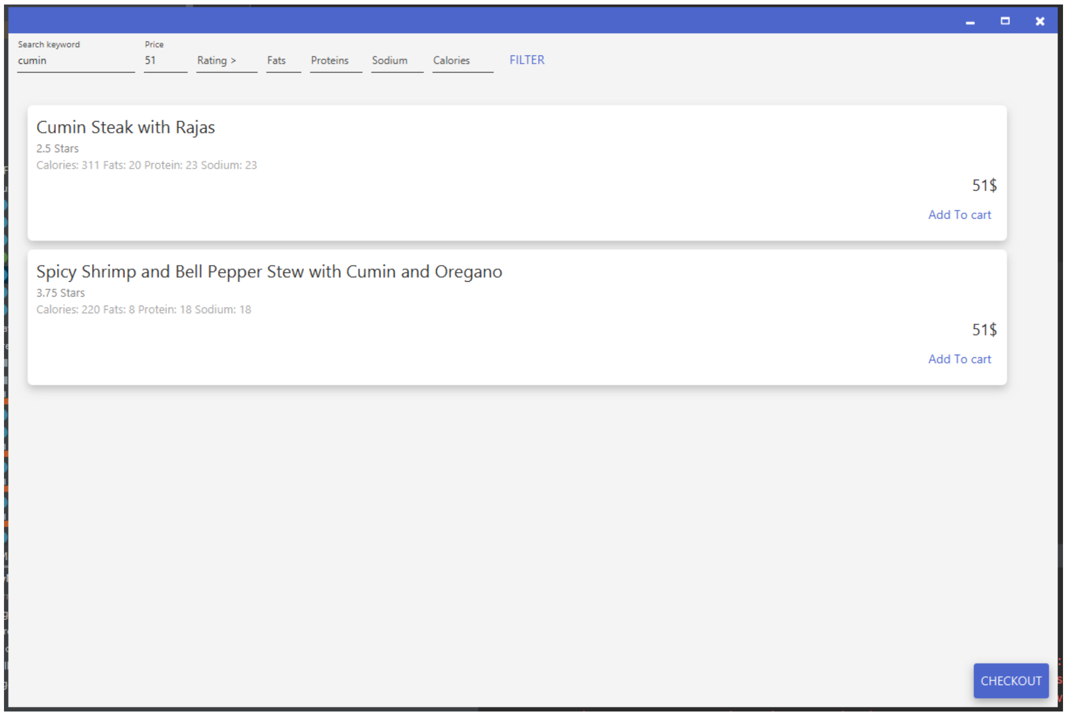
"Strawberry-Rhubarb ""Courting"" Cake "
4.375 Stars
Calories: 573 Fats: 34 Protein: 6 Sodium: 6
77$
Add To cart

"Chinese-Hawaiian ""Barbecued"" Ribs "
3.75 Stars
Calories: 640 Fats: 34 Protein: 28 Sodium: 28
37$
Add To cart

CHECKOUT

Figure 10. The client interface.

Here, the client can filter the products using the various textfields:



Search keyword: cumin | Price: 51 | Rating > | Fats | Proteins | Sodium | Calories | FILTER

Cumin Steak with Rajas
2.5 Stars
Calories: 311 Fats: 20 Protein: 23 Sodium: 23
51$
Add To cart

Spicy Shrimp and Bell Pepper Stew with Cumin and Oregano
3.75 Stars
Calories: 220 Fats: 8 Protein: 18 Sodium: 18
51$
Add To cart

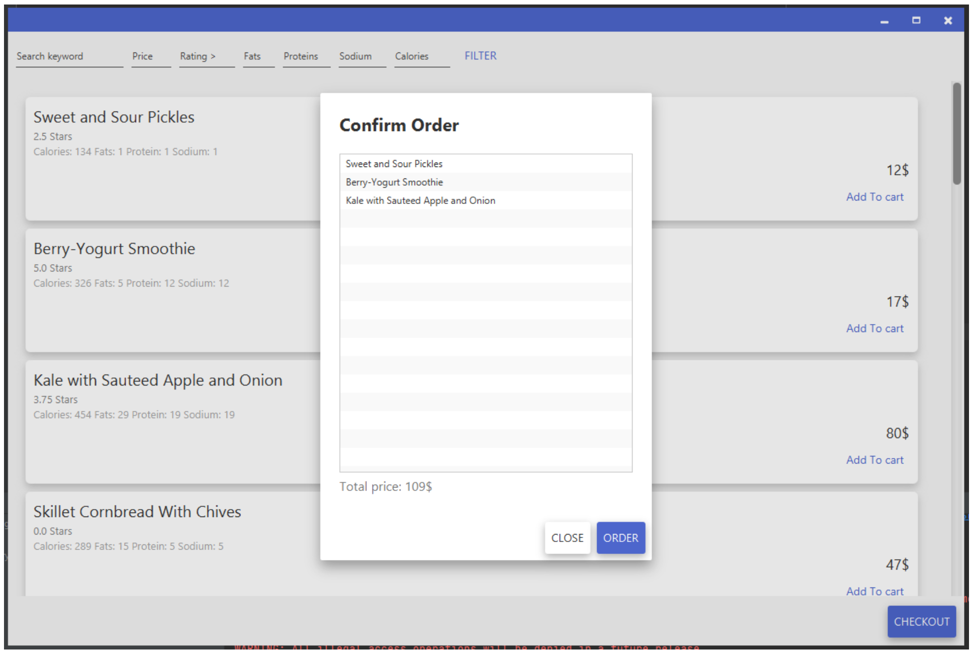
CHECKOUT

Figure 9. Filtering for keyword and price.

Because of the great load of data parsed from the products file, we cannot load all of it simultaneously. Knowing this, the interface was design in such a way to support lazy loading and pagination. The user can scroll through a list of twenty products (that list be it filtered or not) and once he reaches the bottom of this list, if there are still products available, twenty more products will be generated and rendered inside the list. This mechanism assures small load times and minimal processing work for the application to load products because the user will never (or is unlikely) to scroll through the 12000 products available to him.

Note this approach is better suited for distributed applications where the view of the application is opened on one computer, while the backend and database access is hosted on another. The pagination will reduce the network load and the resources necessary to run the app smoothly.*

The user can add products to his cart and when he is done shopping, he can click the checkout button to see the total cost and to place the order, recieving a bill afterwards.



Sweet and Sour Pickles
2.5 Stars
Calories: 134 Fats: 1 Protein: 1 Sodium: 1
12$

Berry-Yogurt Smoothie
5.0 Stars
Calories: 326 Fats: 5 Protein: 12 Sodium: 12
17$

Kale with Sauteed Apple and Onion
3.75 Stars
Calories: 454 Fats: 29 Protein: 19 Sodium: 19
80$

Skillet Cornbread With Chives
0.0 Stars
Calories: 289 Fats: 15 Protein: 5 Sodium: 5
47$

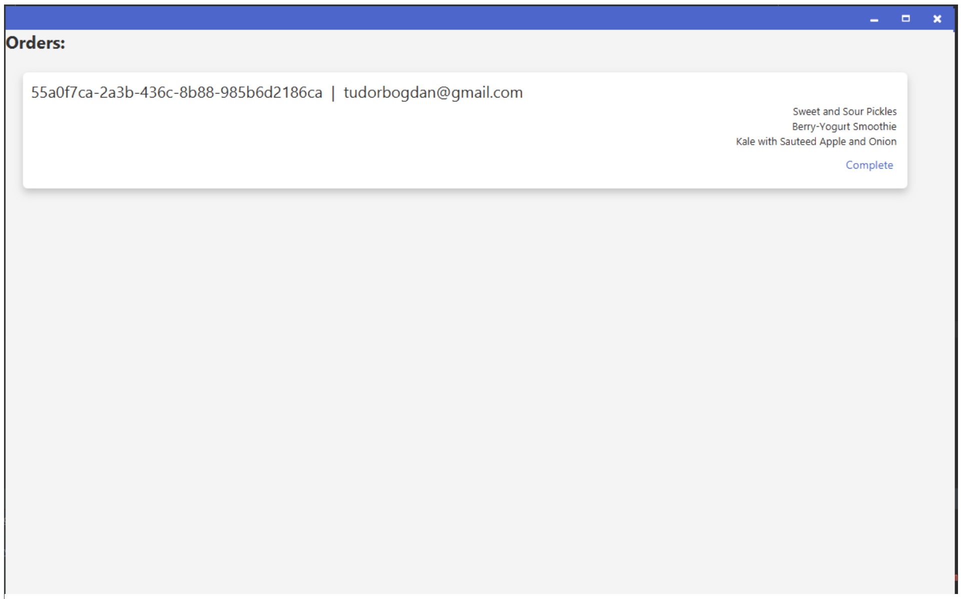
Confirm Order

Sweet and Sour Pickles
Berry-Yogurt Smoothie
Kale with Sauteed Apple and Onion

Total price: 109$

Figure 11.Checkout dialog.

When the employee logs into the system, he will wait until a client will make an order. When that client will finalize the order, he will recieve a new card listing the id of the order, the client who ordered it and the contents of the order to be prepared:



Orders:

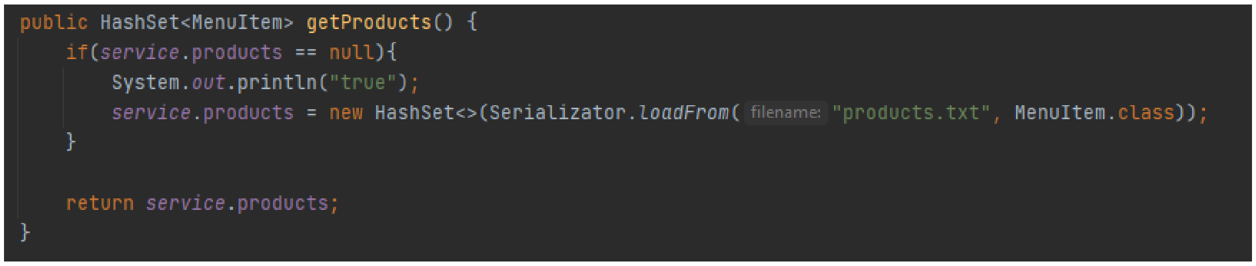
55a0f7ca-2a3b-436c-8b88-985b6d2186ca | tudorbogdan@gmail.com

Sweet and Sour Pickles
Berry-Yogurt Smoothie
Kale with Sauteed Apple and Onion

Figure 12. The employee view.

The implementation of the business logic revolves around the Delivery service class. The delivery service holds multiple types of data including users, products and orders, but not all parts of the application use all these models in their composition. That is why, when implementing the methods for retriving the data from the service, I choose an approach which will lazily load the requested data only when it is necessary for the application.

For example: A client doesn't need to see all the users, therefore he shouldn't load them from the files, while an employee doesn't need to access the products, he only needs acccess to the orders and the products specifically assigned to that particular order and so on. An example of this lazy loading is given here:

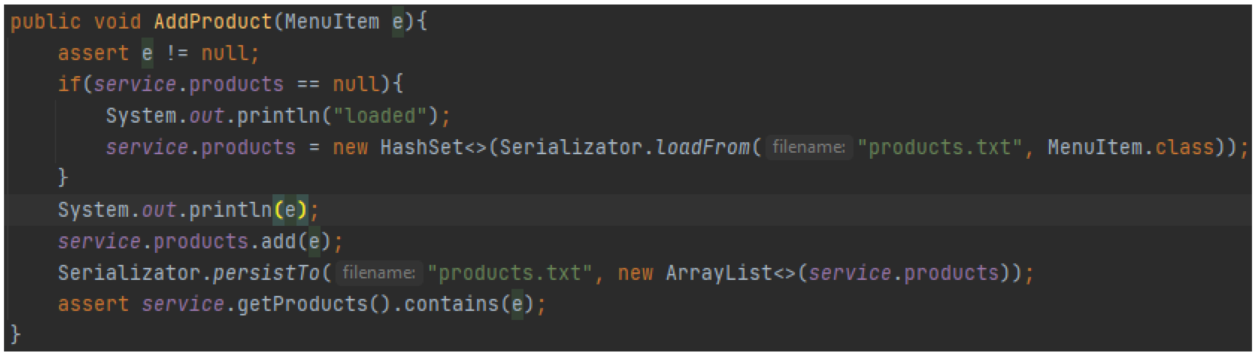


```java
public HashSet<MenuItem> getProducts() {
    if(service.products == null){
        System.out.println("true");
        service.products = new HashSet<>(Serializator.loadFrom( filename: "products.txt", MenuItem.class));
    }

    return service.products;
}
```

Figure 13. Lazy loading of resources.

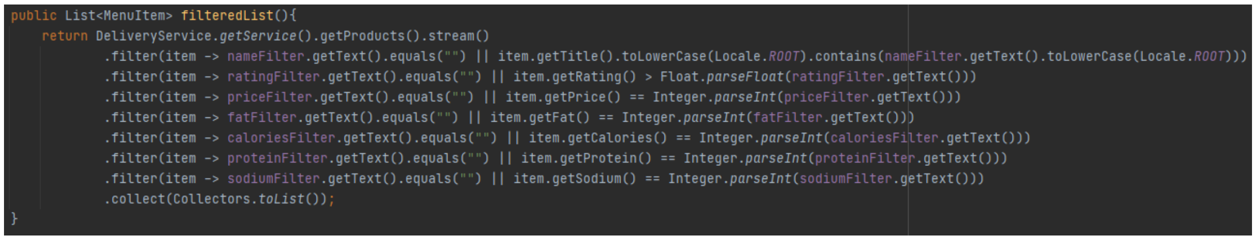
When a modification is issued to the set of products in our delivery service, the internal memory of the delivery service will change and every change will be persisted to a file in serialized format.



```java
public void AddProduct(MenuItem e){
    assert e != null;
    if(service.products == null){
        System.out.println("loaded");
        service.products = new HashSet<>(Serializator.loadFrom( filename: "products.txt", MenuItem.class));
    }
    System.out.println(e);
    service.products.add(e);
    Serializator.persistTo( filename: "products.txt", new ArrayList<>(service.products));
    assert service.getProducts().contains(e);
}
```

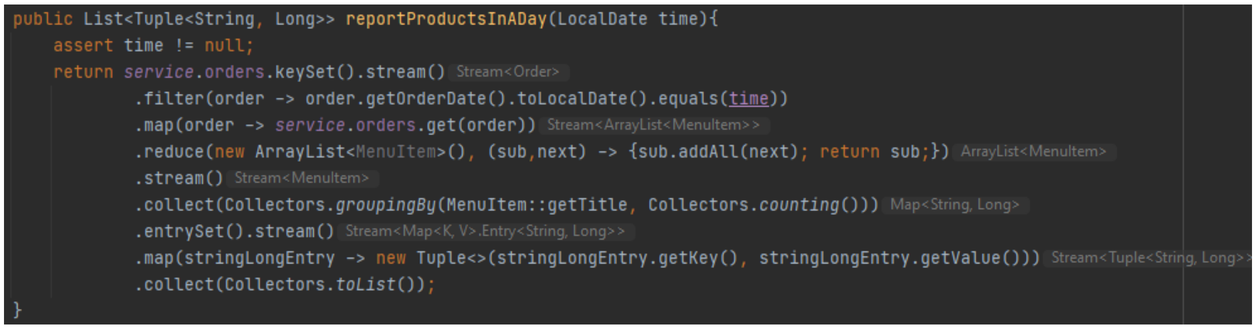
Figure 14. Data persistence.

The filtering capabilities of the client and the administrator have been implemented using java stream processing:



```java
public List<MenuItem> filteredList(){
    return DeliveryService.getService().getProducts().stream()
            .filter(item -> nameFilter.getText().equals("") || item.getTitle().toLowerCase(Locale.ROOT).contains(nameFilter.getText().toLowerCase(Locale.ROOT)))
            .filter(item -> ratingFilter.getText().equals("") || item.getRating() > Float.parseFloat(ratingFilter.getText()))
            .filter(item -> priceFilter.getText().equals("") || item.getPrice() == Integer.parseInt(priceFilter.getText()))
            .filter(item -> fatFilter.getText().equals("") || item.getFat() == Integer.parseInt(fatFilter.getText()))
            .filter(item -> caloriesFilter.getText().equals("") || item.getCalories() == Integer.parseInt(caloriesFilter.getText()))
            .filter(item -> proteinFilter.getText().equals("") || item.getProtein() == Integer.parseInt(proteinFilter.getText()))
            .filter(item -> sodiumFilter.getText().equals("") || item.getSodium() == Integer.parseInt(sodiumFilter.getText()))
            .collect(Collectors.toList());
}
```

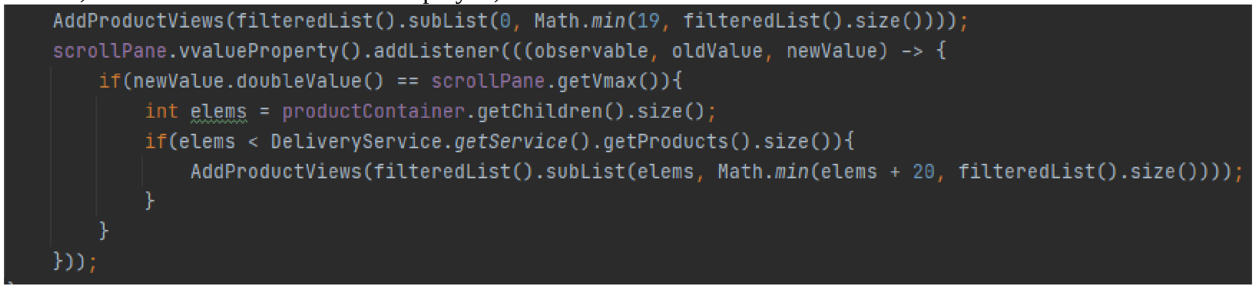
Figure 15. Filtering of menu items.



```java
public List<Tuple<String, Long>> reportProductsInADay(LocalDate time){
    assert time != null;
    return service.orders.keySet().stream() Stream<Order>
            .filter(order -> order.getOrderDate().toLocalDate().equals(time))
            .map(order -> service.orders.get(order)) Stream<ArrayList<MenuItem>>
            .reduce(new ArrayList<MenuItem>(), (sub,next) -> {sub.addAll(next); return sub;}) ArrayList<MenuItem>
            .stream() Stream<MenuItem>
            .collect(Collectors.groupingBy(MenuItem::getTitle, Collectors.counting())) Map<String, Long>
            .entrySet().stream() Stream<Map<K, V>.Entry<String, Long>>
            .map(stringLongEntry -> new Tuple<>(stringLongEntry.getKey(), stringLongEntry.getValue())) Stream<Tuple<String, Long>>
            .collect(Collectors.toList());
}
```

Figure 16. Filtering of products ordered in a day and computing of the frequency of the ordered product.

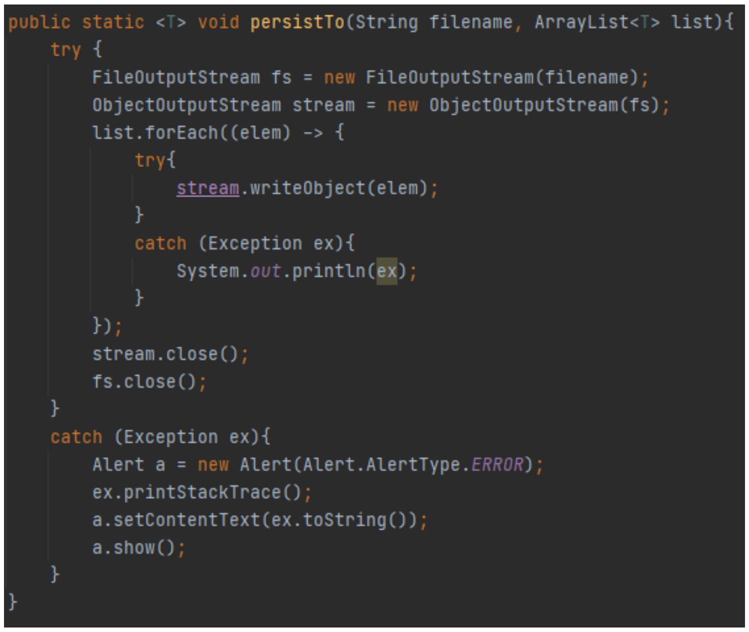
The lazy loading and pagination was done using the subList() method of the lists. To trigger the load of the next page, an event listener has been attached to the scrollpane holding the products. Each time the user will scroll to the bottom, if there is still content to be displayed, it will be rendered.



```java
    AddProductViews(filteredList().subList(0, Math.min(19, filteredList().size())));
    scrollPane.vvalueProperty().addListener(((observable, oldValue, newValue) -> {
        if(newValue.doubleValue() == scrollPane.getVmax()){
            int elems = productContainer.getChildren().size();
            if(elems < DeliveryService.getService().getProducts().size()){
                AddProductViews(filteredList().subList(elems, Math.min(elems + 20, filteredList().size())));
            }
        }
    }));
```

Figure 17. Pagination of the products.

The serializator class exposes two static methods for persisting and loading data. These methods are generic, thus any object can be passed to them to be serialized, if that object is serializable.



```java
public static <T> void persistTo(String filename, ArrayList<T> list){
    try {
        FileOutputStream fs = new FileOutputStream(filename);
        ObjectOutputStream stream = new ObjectOutputStream(fs);
        list.forEach((elem) -> {
            try{
                stream.writeObject(elem);
            }
            catch (Exception ex){
                System.out.println(ex);
            }
        });
        stream.close();
        fs.close();
    }
    catch (Exception ex){
        Alert a = new Alert(Alert.AlertType.ERROR);
        ex.printStackTrace();
        a.setContentText(ex.toString());
        a.show();
    }
}
```

Figure 18. The serializator peristance method.

5) Results

The application behaves as intended. It will allow all users to perform all the specified business requirements on the models of our app.

6) Conclusions

This project has been one of the greatest challenges faced while learning programming techniques in java. In its composition, it required 4 different design patterns to be implemented and used to create a beautiful and functional application. Because of that difficulty, I was required to grow and to expand my programming knowledge of both techniques used in the field, but also my knowledge of the javaFX framework. Now that we are nearing the end of the semester and the fact that this is the last project for this subject, I look back at the things I have learned this year and I can say that I grew as a programmer in many ways by implementing these "toy" projects. Even though they cannot be compared to real production application, I like to think that our efforts to create them have taught us much about the field of programming and I can say that I am proud to have completed all of them, be they as they are small and simple.

7) Bibliography

1) JAVA FX - JavaFX (openjfx.io)

2) JFOENIX UI library - JFoenix

3) GRADLE - Gradle Build Tool

4) FONT AWESOME icon font library - Font Awesome

5) FontAwesomeFX package - Jerady / fontawesomefx — Bitbucket