



Tastevin

Jan Janusz Dębowski, 240187

Isabella Reggi, 240189

Henrik Kronborg Pedersen

Decanter SAS

72,692 characters

ICT

Bachelor (7th semester)

12.12.2018



Abstract

Tastevin is a software system, the main feature of which is matching different types of food to wine by means of data mining. Its other features strive to deliver an intuitive and modern user experience. It has been developed in various environments and incorporates multiple platforms in its deployment. The system components comprise of two data mining models (primary and auxiliary), a server and a client mobile application.

The following report describes the system in terms of analysis, design and implementation of these components and makes relevant distinctions between the system as designed and its resulting prototype. It furthermore discusses technical aspects of the development process and as such can be subject to academic scrutiny.

Appendix section contains full legacy code base, whereas standardized modular framework configurations (such as Visual Studio) are described within the document itself.

The assumed structure of chosen methods and design implications is written in a format meant to be easy to follow for readers taking interest in replication of the system.

Table of content

Abstract	2
1. Glossary	8
Abbreviations	8
Context and definitions	8
2. List of figures and tables	9
3. Acknowledgements	11
4. Introduction	11
5. Methods	12
5.1. Tools and environments	12
5.2. Data Mining Analysis	12
5.3. Database	13
5.3.1. Entities	14
Entities	14
5.3.2. Triggers	15
5.3.3. Prototype Entities	16
5.4. Primary Data Mining Model (<i>Matchmaking</i>)	17
5.4.1. Design	17
5.4.2. Implementation	17
5.5. Auxiliary Data Mining Model (<i>Recommendations</i>)	20
5.5.1. Design	20
5.5.2. Implementation	20
5.6. Android Analysis	23
5.7. Android Frontend	26
5.7.1. Design	26
Login Activity Diagram	26
Wine Info Navigation Activity Diagram	27
Add Review Activity Diagram	28
Get Matching Activity Diagram	29
Insert Matching Activity Diagram	30
Get Recommendation Activity Diagram	31
5.7.2. Implementation	32

5.8.	Android Backend	37
5.8.1.	Design	37
5.8.2.	Implementation	38
5.9.	Android SDK Integrations	51
5.9.1.	Facebook Login	51
5.9.2.	PayPal Integration	53
5.10.	Network Analysis	55
5.10.1.	GET	55
5.10.2.	Prototype GET	57
5.11.	Network Server	58
5.11.1.	Design	58
5.11.2.	Implementation	59
	• GET	59
	• INSERT	63
	• MATCHING	65
	• PURCHASE	66
	• REVIEW	66
	• USER	66
	• WINE	66
	• INGREDIENT	67
	• SERVERMODEL	67
	• HTTP	70
	• TastevinMain	70
5.12.	Testing	71
6.	Results	74
7.	Discussion	74
8.	Conclusion	79
	References	80
	Appendices	84
a)	Mockup	84
b)	User Guide	85
1.	Settings Prototype	85
a.	Server	85

b. Application	85
2. Log in	87
3. Dashboard	88
a. Navigation Menu	89
4. Get a matching	90
5. Get a matching	91
6. Navigate to Wine Info	92
a. Write a review	93
b. Insert a matching	93
7. Recommendations	94
8. Shop	95
a. Search	95
b. Filter	96
9. Cart	97
c) UML Server Class Diagram	98
d) UML Client Class Diagram	99
e) Server Code	99
GET	99
INSERT	114
Ingredient	118
Matching	119
Purchase	120
Review	122
User	124
Wine	125
HTTP	127
ServerModel	128
TastevinMain	132
f) Android Code	132
CartPresenter	132
Cart Presenter Interface	137
Cart View	137
Cart View Interface	140

Dashboard Presenter	140
Dashboard Presenter Interface	143
Dashboard View	143
Dashboard View Interface	146
Login Presenter	146
Login Presenter Interface	149
Login View	149
Login View Interface	152
MainActivity Presenter	152
MainActivity Presenter Interface	153
MainActivity View	154
MainActivity View Interface	157
MatchedList Presenter	157
MatchedList Presenter Interface	158
MatchedList View	158
MatchedList View Interface	162
Matching Presenter	162
Matching Presenter Interface	166
Matching View	166
Matching View Interface	169
Recommendation Presenter	170
Recommendation Presenter Interface	172
Recommendation View	173
Recommendation View Interface	174
Shop Presenter	175
Shop Presenter Interface	178
Shop View	178
Shop View Interface	183
WineInfo Presenter	183
WineInfo Presenter Interface	189
WineInfo View	189
SubWineInfo Presenter	196
SubWineInfo Presenter Interface	197

SubWineInfo View	197
SubWine View Interface	199
WineReviews Presenter	199
WineReviews Presenter Interface	202
WineReviews View	202
WineReviews View Interface	205
Activities	205
CheckoutAdapter	205
ClientRequests	206
Config	213
DashboardAdapter	214
Filter	215
Ingredient	217
Matching	218
Parser	220
Purchase	223
Review	225
User	227
Wine	228
WineAdapter	231

1. Glossary

Abbreviations

SQL (viz. Sequel) - Structured Query Language.

SSAS - SQL Server Analysis Services.

SSDT - SQL Server Data Tools.

SSMS - SQL Server Management Studio.

IIS - Internet Information Services.

UML - Unified Modeling Language.

PHP - Popular General-Purpose Scripting Language.

SDK - Software Development Kit.

XML - Extensible Markup Language.

IDE - Integrated Development Environment.

GUI - Graphical User Interface.

HTTP - Hypertext Transfer Protocol.

Context and definitions

Activity - used in this document in multiple contexts, “activity” can refer to:

- user interaction (activity diagram).
- Android Activity (Android Studio code fragmentation class).

Intent - used in this document in multiple contexts, “intent” can refer to:

- intention.
- Android Intent (Android Studio navigation class).

.xmls - (plural) XML files, containing layout structure and presentation elements.

Matching - refers to a match between a recipe and wine.

Matchmaking - refers to the act of creation of such a matching.

2. List of figures and tables

Figure 1 Database Entity Relationship model	13
Figure 2 UpdateReview Trigger	15
Figure 3 UpdateTastes Trigger	16
Figure 4 Example data row part 1	17
Figure 5 Example data row part 2	18
Figure 6 Algorithm setup part 1	18
Figure 7 Algorithm setup part 2	19
Figure 8 Decision Tree	20
Figure 9 Auxiliary example data rows	21
Figure 10 Auxiliary algorithm setup	22
Figure 11 Auxiliary Decision Tree	22
Figure 12 Use Case Diagram	24
Figure 13 Matching Use Case description	25
Figure 14 Login Activity Diagram	26
Figure 15 WinInfo Activity Diagram	27
Figure 16 Add Review Activity Diagram	28
Figure 17 Get Matching Activity Diagram	29
Figure 18 Insert Matching Activity Diagram	30
Figure 19 Get Recommendation Activity Diagram	31
Figure 20 Fronted content structure diagram	32
Figure 21 Example frontend XML	34
Figure 22 Example frontend View interface	34
Figure 23 Example frontend onCreate method	35
Figure 24 Example frontend onCreateView method	35
Figure 25 Example frontend createAlertDialogWines method	36
Figure 26	37
Figure 27 GetWinInfo method	39
Figure 28 InsertRating method	40
Figure 29 OnNavigationItemSelectedListener method	41
Figure 30 GetActivitiesList method	42
Figure 31 Ingredient switch case	44
Figure 32 AddToCart method	45
Figure 33 GetExtras method	47
Figure 34 AddReview method	48
Figure 35 DeleteWine method	49
Figure 36 Facebook manifest data	51
Figure 37 Facebook Gradle build data	51
Figure 38 Facebook Callback Method	52
Figure 39 Facebook Access Token	52
Figure 40 Facebook Callback Manager	52
Figure 41 PayPal Manifest Permission	53
Figure 42 PayPal Gradle build data	53

Figure 43 PayPal Config data.....	53
Figure 44 PayPal onDestroy and onCreate methods.....	54
Figure 45 HTTP GET Sequence Diagram)	55
Figure 46 HTTP POST Sequence Diagram.....	56
Figure 47 Prototype HTTP GET Sequence Diagram.....	57
Figure 48 Server UML Class Diagram	58
Figure 49 Server getRecommendedWine method.....	61
Figure 50 Server getMatchedWine method.....	62
Figure 51 Server insertRating method	63
Figure 52 Server inserPurchases method.....	64
Figure 53 Server insertMatching method	65
Figure 54 Server switch case	68
Figure 55 Server insert switch case.....	70
Figure 56 Singleton Query Test comparison	71
Figure 57 Testing table.....	74
Figure 58.....	76
Figure 59.....	77
Figure 60.....	77
Figure 61.....	77
Figure 62.....	78

3. Acknowledgements

The team would like to first and foremost acknowledge support provided by our supervisor. Throughout the process he has provided knowledge, guidance and inspiration. The project discussed in this report would not have been possible without his input.

4. Introduction

The world used to be a much different place throughout humanity's past centuries. All we have known were physical analogous media, information we acquired was stored on paper and processed by neurons of our brains. This however did not withhold us from progress. Knowledge was passed on and reinterpreted by new generations, inventions birthed by human minds, new associations found in natural, physical signals.

But times have changed. Through our way of striving to new horizons and broadening our understanding of the world around us, mankind, since its conception, created tools to ease its existence. What used to have been clay tablets, beads on a string and pictograms, blossomed into paper, mathematical algorithms and complex languages. Yet, in the reality we find ourselves, paper was substituted with series of transistors, algorithms switched their processing environments from real brains to ones made of silicone logic gates and state of the art languages are now written with two letters: 0 and 1.

As time has passed, these changes became more and more widespread. Information processing has moved to digital environments, forming data mining algorithms adjacent to machine learning in creation of artificial intelligence.

Those algorithms are widely used to find associations and patterns, much like human brains do, in data sets too great to otherwise process. Data mining finds applications in statistics, market basket analysis, surveillance, research and banking, to name a few. However, with the availability of these tools, processing large sets of information for pattern recognition has come closer to the consumer market in more direct ways.

Utilizing data mining techniques and mobile devices, some of humanity's earliest joys: food and wine, can now be optimally paired for enjoyment at the click of a virtual button. This report addresses technical matters involved in the creation of such a system, methodologies and technology employed in its development. It also shows why its structure had to have been limited and discusses what issues plague development in frameworks it utilizes, as well as this particular instance of heterogeneous software implementation on varied hardware platforms.

5. Methods

5.1. Tools and environments

- Visual Studio 2017 Community - was the IDE used for mining model configuration.
- Android Studio - mobile application development IDE used for client implementation.
- Eclipse - Java IDE used to develop and deploy the server.
- SSAS - analysis services used by Visual Studio for data mining purposes.
- SSDT - SQL toolkit required for SSAS setup.
- SSMS - framework used for deploying and managing databases.
- Postman - connection framework employed for testing process.
- XAMPP - server framework, the Apache feature of which had been used for Braintree SDK development.
- Astah - UML environment the diagrams in this document were designed with.
- Fluid UI - online prototype mockup designer.
- Sourcetree - source control Git GUI made by Atlassian, powered by GitHub.

5.2. Data Mining Analysis

The Data Mining Model was the initial system's component under development. Due to their intrinsically connected nature, the mining model and database were developed concurrently in the Ouroboros approach. Both had gone through gradual changes according to release versions, respectively increasing in complexity and consistency.

The process which they had undergone required cyclical analysis, design and implementation, making it somewhat difficult to structure their documentation in other ways. The final version of the database is discussed in detail in the *database* section, whereas each of the mining model's versions are discussed in their respective subchapters. As the core component of the system, *Matchmaking's* development had been versioned in order to guarantee highest quality of outcome the team is capable of.

Thus, every individual version of the mining model was separately analyzed and meant to achieve varying goals. This chapter contains only persistent changes affecting the final product. Please refer to *Process Report* document for additional insight into versioning.

Subsequently, the team had decided upon an auxiliary mining model to serve two purposes:

- Recommendations
- Matchmaking output filtering

With the former, somewhat self-explanatorily suggesting recommendations for the user. That functionality is based on an idea of simulating the user's taste buds based on their recent purchases, with each wine containing a flavor index, altering the user's average index for each of the base flavors (sweet, salty, umami etc.).

The latter functionality mentioned above uses the auxiliary mining model to filter *Matchmaking's* output based on the user's personal preferences in order to make every matching tailor-suited to each individual user.

5.3. Database

The database's final version caters to the primary and auxiliary mining models (*Matchmaking* and *Recommendations*).

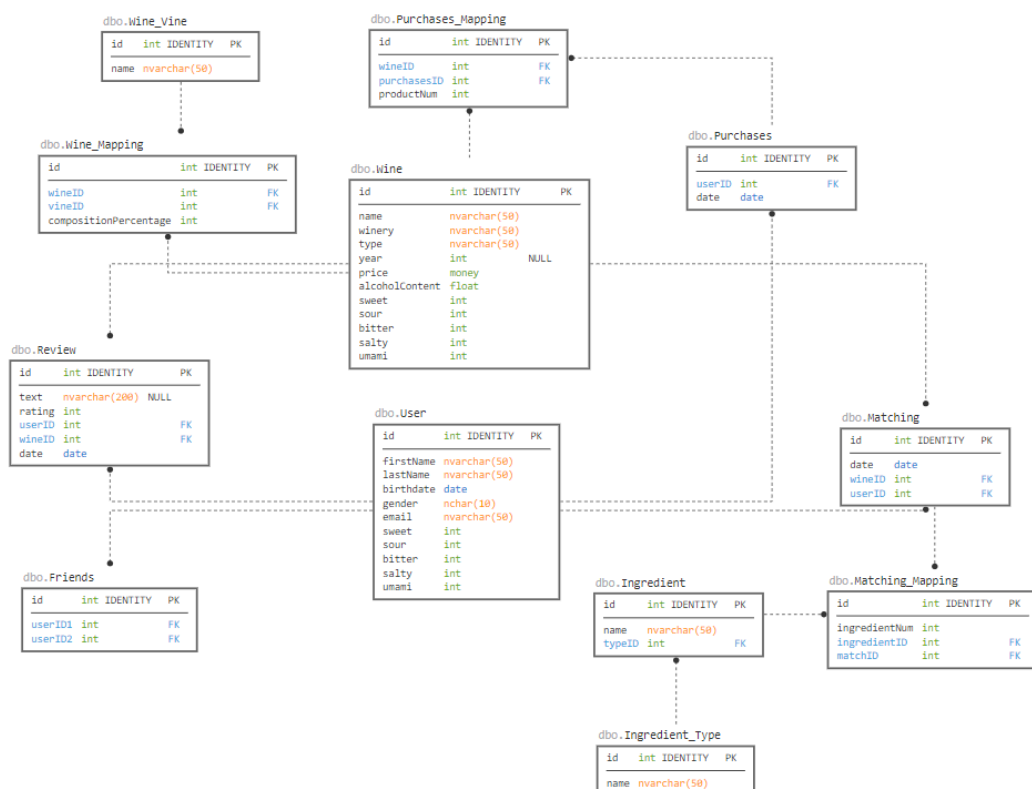


Figure 1 Database Entity Relationship model

5.3.1. Entities

Entities

- ***IngredientType*** - because ingredients of each meal can be segregated into different categories, each with a different wine affinity, it was crucial to represent that reality in digital form. Thus, this entity with a foreign key to *Ingredient*, serves that particular purpose. Its existence allows for ease of generalization and association for data-mining purposes and clarity in terms of human readability. Examples of its content include meat, vegetables, starches etc.
- ***Ingredient*** - existence of this entity is self-explanatory. It represents individual ingredients and thus user input into the algorithm.
- ***WineVine*** - albeit confusingly named, *WineVine* serves a broader purpose, in combination with *WineMapping*, than its conceptual predecessor *WineType*. It encompasses general derivation properties a given wine can possess, such as the name of the vines in their composition (*Pinot Noir*, etc.).
- ***Wine*** - represents a bottle/kind of wine. It is described using basic parameters a wine bottle entails, such as year of production, alcoholic content, price etc.
- ***WineMapping*** - using many-to-many relationship, this entity is crucial for representing wine objects. Its fields are foreign keys to *Wine* and *WineVine* as well as one other essential component: *CompositionPercentage*. Because in reality wines are more often than not blends of several types of wine, this field allows the system to mimic this mechanic by stating what amount of which wine had gone into the makings of the final product.
- ***Matching*** - the basis of Mining Model's content is stored inside of this entity, which creates a connection directly between a wine and a user, and indirectly with a recipe.
- ***MatchingMapping*** - a relative novelty in terms of previous versions of the system, this entity's role defines much of the idea behind the database's final functionality. It allows for multiple ingredients to be mapped to a single matching (one-to-many relationship), and paves way for a necessary mining model functionality through the *ingredientNum* field, allowing the system to sort ingredients in order in which they are processed by the Mining Model (functionality further discussed in *Primary Data Mining Model* chapter). Thus, in a way this entity represents the actual recipe a user creates.
- ***User*** - is one of the most standard entities in any given system. It represents its namesake and contains user data the system requires to function. *Tastevin's* "personal" spin on it is the addition of fields representing the users' taste buds i.a. *sweet*, *salty*, *umami* etc., forming the data basis of the auxiliary *Recommendations* data mining model (functionality further discussed in *Auxiliary Data Mining Model* chapter).

- **Friends** - this table is to register all the users who are friends with each other. In this way a user can easily track their friends' activities.
- **Purchases** - also one of industry standards, this entity represents a financial relationship between users and products. It contains a *date* field, which allows the product owner to log users' purchases.
- **PurchasesMapping** - a many-to-many relationship entity, this one connects purchases to individual products that had been purchased as well as their number.
- **Review** - this entity is connected to the Android functionality. It allows users to post text reviews of their purchased products, limited to 200 characters and complete with a numerical rating (five-star scale).

5.3.2. Triggers

- **updateReview** - This trigger is activated in case of an insert query in the table. It checks if the user already made a review about that wine. If it already exists, it will update the value of the rating, the textual content and the date.

```
ALTER TRIGGER [dbo].[updateReview]
ON [dbo].[Review]
INSTEAD OF INSERT
AS
BEGIN
    IF EXISTS(SELECT * FROM dbo.Review WHERE dbo.Review.wineID
        IN (SELECT inserted.wineID FROM inserted) AND
        dbo.Review.userID IN (SELECT inserted.userID FROM inserted))
    BEGIN
        UPDATE dbo.Review SET rating = (SELECT inserted.rating FROM inserted),
            date = GETDATE()
        WHERE dbo.Review.wineID
            IN (SELECT inserted.wineID FROM inserted) AND
            dbo.Review.userID IN (SELECT inserted.userID FROM inserted)
    END
    ELSE
    BEGIN
        INSERT INTO dbo.Review VALUES((SELECT inserted.text FROM inserted),
            (SELECT inserted.rating FROM inserted), (SELECT inserted.userID FROM inserted),
            (SELECT inserted.userID FROM inserted), (SELECT inserted.date FROM inserted))
    END
END
```

Figure 2 UpdateReview Trigger

- **updateTastes** - This trigger is activated when the elements of a purchase are inserted in the table Purchases_Mapping. It takes the value of the wine flavor indexes and sums them to the level of the user in order to update them. The user tastes are characterized by a constraint that restrains them from having a value between 0 and 100. In this way trigger will never change the user taste to a negative level or too high a value.

```
ALTER TRIGGER [dbo].[UpdadeTastes]
ON [dbo].[Purchases_Mapping]
AFTER INSERT
AS
BEGIN
    UPDATE dbo.[User] SET
        sweet += dbo.Wine.sweet,
        bitter += dbo.Wine.bitter,
        salty += dbo.Wine.salty,
        sour += dbo.Wine.sour,
        umami += dbo.Wine.umami
    FROM Inserted i
    INNER JOIN dbo.Wine ON dbo.Wine.id = i.wineID
    INNER JOIN dbo.Purchases ON dbo.Purchases.id = i.purchasesID
    WHERE dbo.[User].id = dbo.Purchases.userID
END;
```

Figure 3 UpdateTastes Trigger

5.3.3. Prototype Entities

These entities were added because, since the direct connection to the data mining structure was not possible, a storage of the predicted values and probabilities was needed.

- **Recommendation** – This entity contains the probabilities of the recommendations for the users. A row consists of the 15 probabilities of the wines used in the structure and the info of the user.
- **MatchProbs** - This entity contains the probabilities of the matchings for the users. A row consists of the 15 probabilities of the wines used in the structure and the some of the ingredients used. The user tastes were not added for a impossibility of storing all those values hand by hand. The filtering with the user's tastes is simulated using the results of the recommendations in the server (See Network Server).

5.4. Primary Data Mining Model (*Matchmaking*)

5.4.1. Design

The design is handled backstage by the *Microsoft Corporation*, creators of the chosen algorithm, *Microsoft Decision Trees* (Microsoft, n.d.)

The algorithm suits the project's purposes due to the fact it creates correlations between inputs, i.e. ingredients, user tastes and predictable i.e. wine by creating a tree of nodes, akin to Binary Tree design pattern. The data used in this case is provided by the matchings made by the users. The inner workings of the algorithm are somewhat obfuscated by mathematical complexity of entropy and Bayesian networks, yet in simple terms each node represents a split created by an input column being closely correlated to the predictable.

5.4.2. Implementation

For the implementation of the Matchmaking Mining Model, an Analysis Services Multidimensional and Data Mining project was created in Visual Studio.

The data needed for this project was collected from a database view called `dbo.Matchings_MM`. This view was obtained by an inner join of the tables `User`, `Matching` and the view `Matching_Raw`.

This view contains the following columns:

- MatchingID - id of the matching.
- WineName - name of the wine.
- Age - age calculated by the birthday of the user.
- Gender - user's gender.
- Sweet - the sweet tasting level of the user.
- Sour - the sour tasting level of the user.
- Bitter - the bitter tasting level of the user.
- Salty - the salty tasting level of the user.
- Umami - the umami tasting level of the user.
- Ingredient1 - the first ingredient of the recipe.
- Ingredient2 - the second ingredient of the recipe.
- Ingredient3 - the third ingredient of the recipe.
- Ingredient4 - the fourth ingredient of the recipe.

MatchingID	WineName	Age	gender	sweet	sour	bitter
1	Chianti Poggio ...	23	F	20	20	20

Figure 4 Example data row part 1

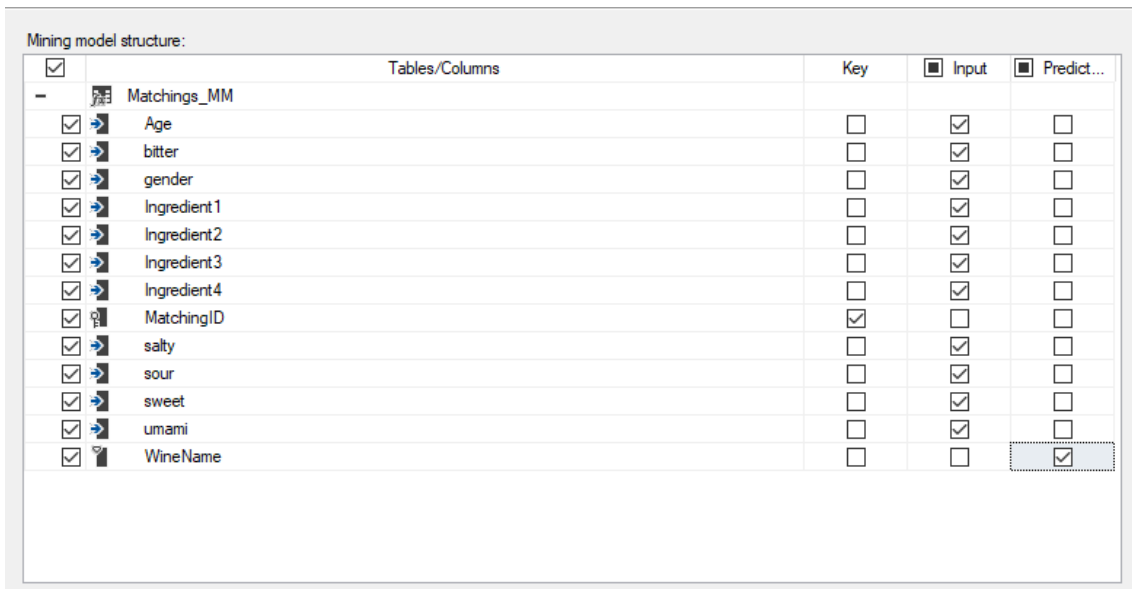
salty	umami	Ingredient1	Ingredient2	Ingredient3	Ingredient4
20	20	Pork Chops	Garlic	Honey	Soy Sauce

Figure 5 Example data row part 2

Since there was the need of a certain amount of data, a minor Java program was made to generate random data with which to populate the database regarding the matchings. The random was prepared using only a limited set of ingredients and the levels 20, 50 and 70 for all the user's tastes.

Post data generation, the connection of the project to the database was made. Then the data source view for the mining model was created from the view dbo.Matching_MM.

The last step was to create the mining model structure. The first step to follow was algorithm selection. In this case the algorithm selected was *Microsoft Decision Tree*. (Microsoft, n.d.) Then the data source could be selected and used as a 'case table'.



Tables/Columns	Key	Input	Predict...
Matchings_MM			
Age	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
bitter	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
gender	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Ingredient1	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Ingredient2	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Ingredient3	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Ingredient4	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
MatchingID	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
salty	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
sour	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
sweet	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
umami	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
WineName	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>

Figure 6 Algorithm setup part 1

The next step was to decide how the different data of the data source view was going to be used. The mining structure to be created needed a key value, that in this case was the MatchingID, a predictable value, that in this case was the WineName, and input value, that in this case were covered by all the remaining values.

After this the columns' content and type were specified: all values of which were Discrete except for MatchingID that, since it's the key in the mining model, has the content type Key.

Mining model structure:














	Columns	Content Type	Data Type
	Age	Discrete	Double
	Bitter	Discrete	Long
	Gender	Discrete	Text
	Ingredient1	Discrete	Text
	Ingredient2	Discrete	Text
	Ingredient3	Discrete	Text
	Ingredient4	Discrete	Text
	Matching ID	Key	Long
	Salty	Discrete	Long
	Sour	Discrete	Long
	Sweet	Discrete	Long
	Umami	Discrete	Long
	Wine Name	Discrete	Text

Figure 7 Algorithm setup part 2

The last step for the creation of the mining model structure was to decide the percentage of data that was going to be used for data training and in this case the 30% was chosen.

After this the mining model was created and the last thing to do was to process this structure. However since the data the it was used it's below the standards, the MINIMUM_SUPPORT and SCORE_METHOD needed to have the value equal to 1.

The MINIMUM_SUPPORT specifies the number of cases that a leaf/node must contain and the SCORE_METHOD usually uses the Bayesian Dirichlet Equivalent with Uniform Prior method but instead the Entropy method was selected.(Microsoft, n.d.)

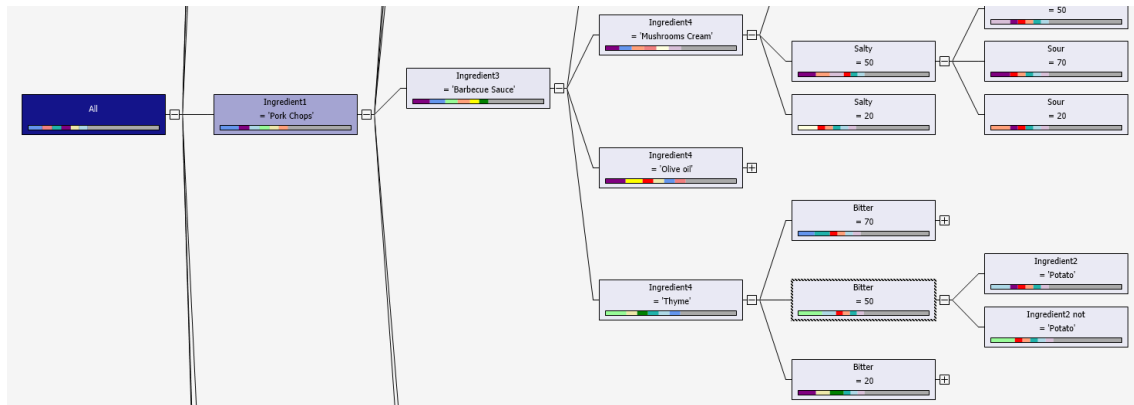


Figure 8 Decision Tree

Subsequently, the decision tree was completed and ready for testing.

5.5. Auxiliary Data Mining Model (*Recommendations*)

5.5.1. Design

This mining model also used the *Microsoft Decision Tree* algorithm (Microsoft, n.d.) and in this case the inputs are only the user info and the predictable is the wine. The data used in this case is provided by the purchases made by the users altering their own taste indexes, as per analysis description.

5.5.2. Implementation

For the implementation of the Auxiliary Mining Model, an Analysis Services Multidimensional and Data Mining project was created in Visual Studio.

The data needed for this project was collected from a database view called `dbo.Purchases_MM`. This view was obtained by an inner join of the tables `User`, `Purchases`, `Purchases_Mapping` and `Wine`.

This view contains the following columns:

- PurchaseID - id of the Purchase
- Name - name of the wine
- UserID - id of the user
- Age - age calculated by the birthday of the user.
- Gender - user's gender.
- Sweet - the sweet tasting level of the user.
- Sour - the sour tasting level of the user.
- Bitter - the bitter tasting level of the user.

- Salty - the salty tasting level of the user.
- Umami - the umami tasting level of the user.

	PurchaseID	name	UserID	gender	Age	sweet	sour	bitter	salty	umami
1	1	Chardonnay Riserva Castel Ringberg	1	F	23	20	20	20	20	20
2	1	Chianti Colli Senesi	1	F	23	20	20	20	20	20

Figure 9 Auxiliary example data rows

Also, in this case, since there was the need of a certain amount of data, a minor Java program was used to generate random data for the purchases. The random was prepared using only the levels 20, 50 and 70 for all the user's tastes.

Since the connection was already established with the previous mining model, the next step was to create a data source view for the mining model. It was called `dbo.Purchases_MM`.

Then the next step was to create the mining structure and, as the main mining model, the *Microsoft Decision Tree* as algorithm was selected and the data source view `Purchase_MM` was selected as case table.

Mining model structure:

	Tables/Columns	Key	Input	Predict...
-	Purchases_MM			
<input checked="" type="checkbox"/>	Age	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
<input checked="" type="checkbox"/>	bitter	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
<input checked="" type="checkbox"/>	gender	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
<input checked="" type="checkbox"/>	name	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/>	PurchaseID	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<input checked="" type="checkbox"/>	salty	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
<input checked="" type="checkbox"/>	sour	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
<input checked="" type="checkbox"/>	sweet	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
<input checked="" type="checkbox"/>	umami	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
<input type="checkbox"/>	UserID	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Recommend inputs for currently selected predictable:

Figure 10 Auxiliary algorithm setup

As mentioned above, the user info(age, gender, bitter, sweet, salty, sour, umami) were used as inputs, the name of the wine as predictable and the purchaseID as key.

After this, the process for the columns' content and type and the percentage of data for the testing followed the same procedure as the main mining structure.

Following, the decision tree for the recommendations was completed and ready for the testing.

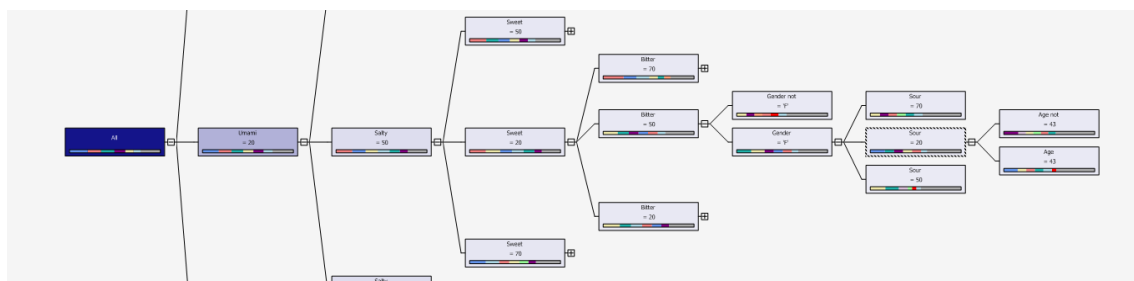


Figure 11 Auxiliary Decision Tree

5.6. Android Analysis

The client-side environment of the application along with a GUI had been chosen to be Android 8.1 (Oreo) API. Market-analysis provided by the chosen IDE - Android Studio - reveals that the second larger majority of users (21.5% marketable devices)(Microsoft, n.d.) are capable of running client application without the necessity of backwards compatibility.

The next several subchapters discuss the final version of the Android application in context of the prototypical implementation, along with suggestions for potential future market-oriented implementation of the product at hand. Due to the nature of the framework, versioning had dealt mainly with SDK choices and is further analyzed in the *Discussion* section of this document.

Android Frontend chapter provides insight into the presentation layer, its elements, layout and decisions governing the user experience, which are fully fleshed out in the User Guide (see Appendix).

Android Backend and *SDK Integrations* concern the logical layer of the application, its design patterns and implementation. Please note that these last two aforementioned sections do not include layout .xmls and their respective descriptions, thus presentation elements should be taken into consideration only within the context of the scope.

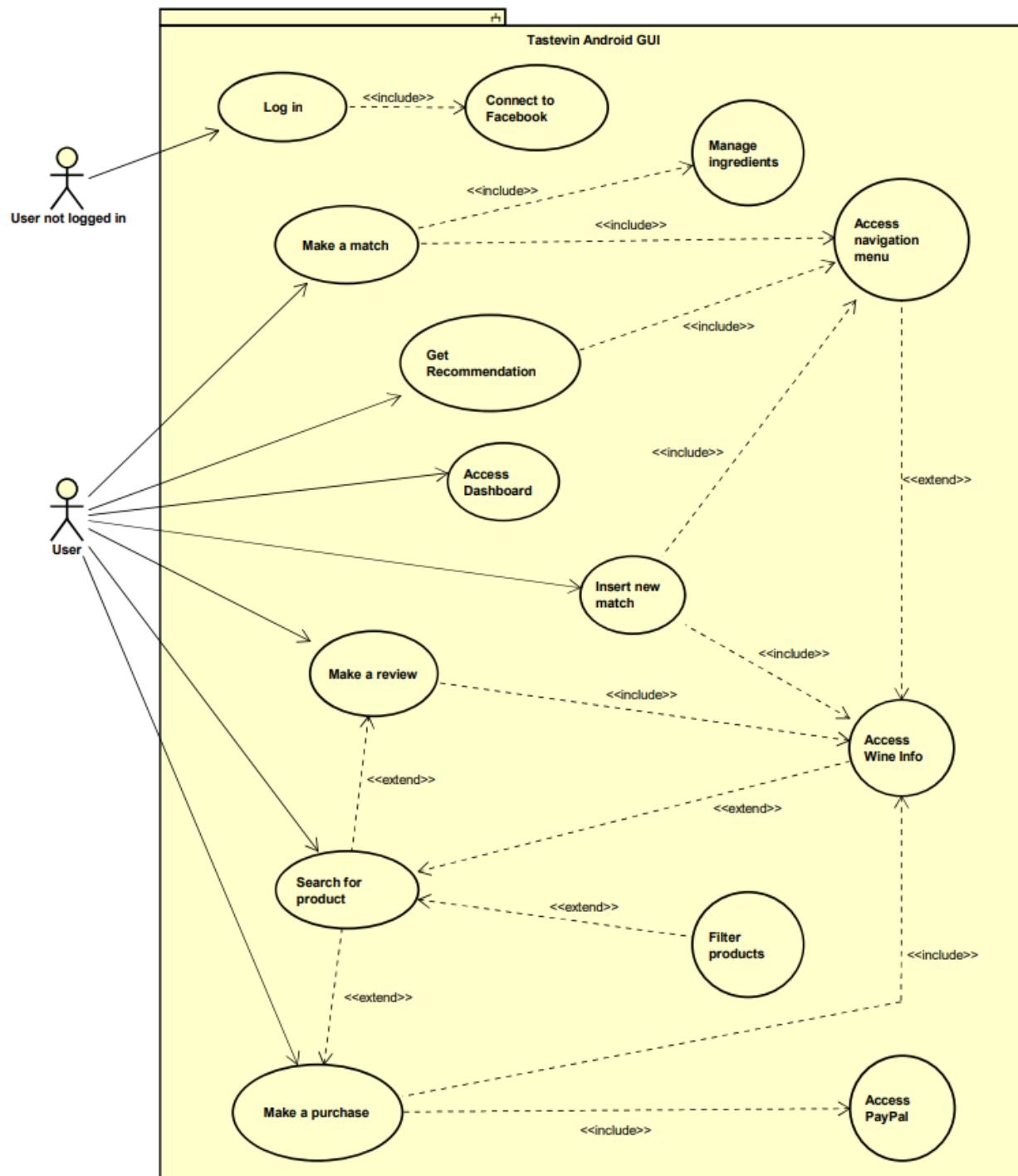


Figure 12 Use Case Diagram



ITEM	VALUE
UseCase	Make a match
Summary	User accesses the system's primary functionality: food-to-wine matching.
Actor	User
Precondition	User has navigated to matching.
Postcondition	User receives a matching to a wine.
Base Sequence	1. User accesses Navigation Drawer Menu. 2. User clicks Matchmaking button. 3. System navigates to Matchmaking. 4. User inputs their ingredients. 5. User presses Ok button. 6. System returns a valid matching.
Branch Sequence	
Exception Sequence	A. No valid matching found -> system notifies the user of their matching having low probability/no matches being found.
Sub UseCase	Access navigation menu, Manage ingredients
Note	

Figure 13 Matching Use Case description

5.7. Android Frontend

5.7.1. Design

The GUI had been designed using several tools and techniques, namely Fluid UI to design a mock-up (please refer to *Appendix*) and Astah framework to create activity diagrams to describe user interaction. All presentation layer elements are handled by *View* (MVP design pattern) section of the code, further discussed in *Android Backend Design*.

Login Activity Diagram

This diagram shows the user's first interaction with the system; login. Very much straightforward in its course, logins rely on access to Facebook to retrieve user data and credential confirmation. This activity serves as the precondition for all other user interactions.

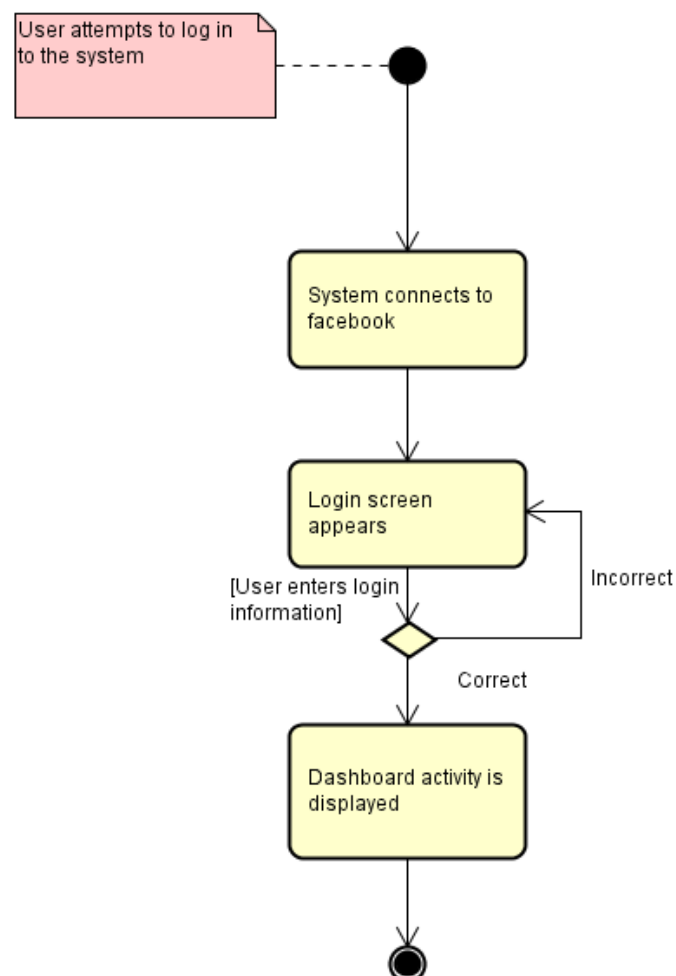


Figure 14 Login Activity Diagram

Wine Info Navigation Activity Diagram

Albeit intuitive in its purpose, the *Wine Info* page requires a more elaborate diagram due to the fact it can be accessed via several routes. The page serves as a product overview fragment, with all data the system holds (aside from the flavor index required for calculations) on an individual item, displayed for the user. This is however not the only purpose of it, as from here users can engage in all product interaction the system holds the capacity for.

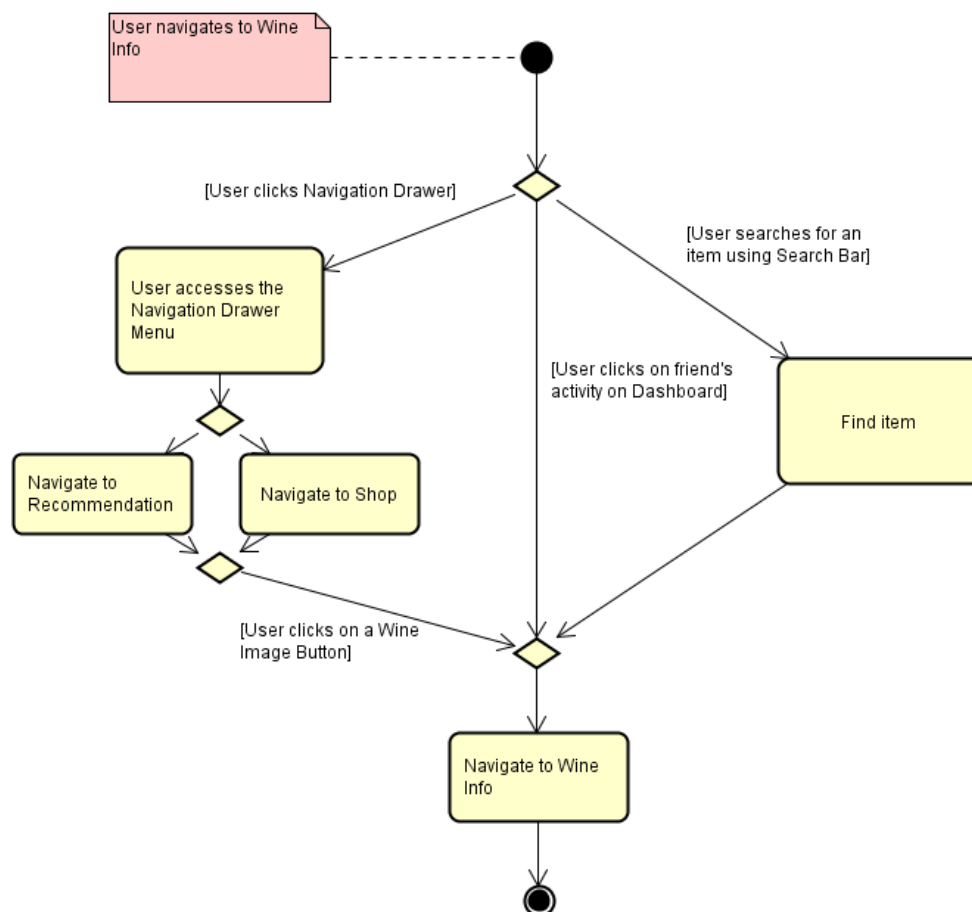


Figure 15 WineInfo Activity Diagram

Add Review Activity Diagram

User reviews hold an important place in modern consumer media, and as such *Tastevin* includes the review functionality. Precondition to reviewing products is the user needs to first navigate to *Wine Info*.

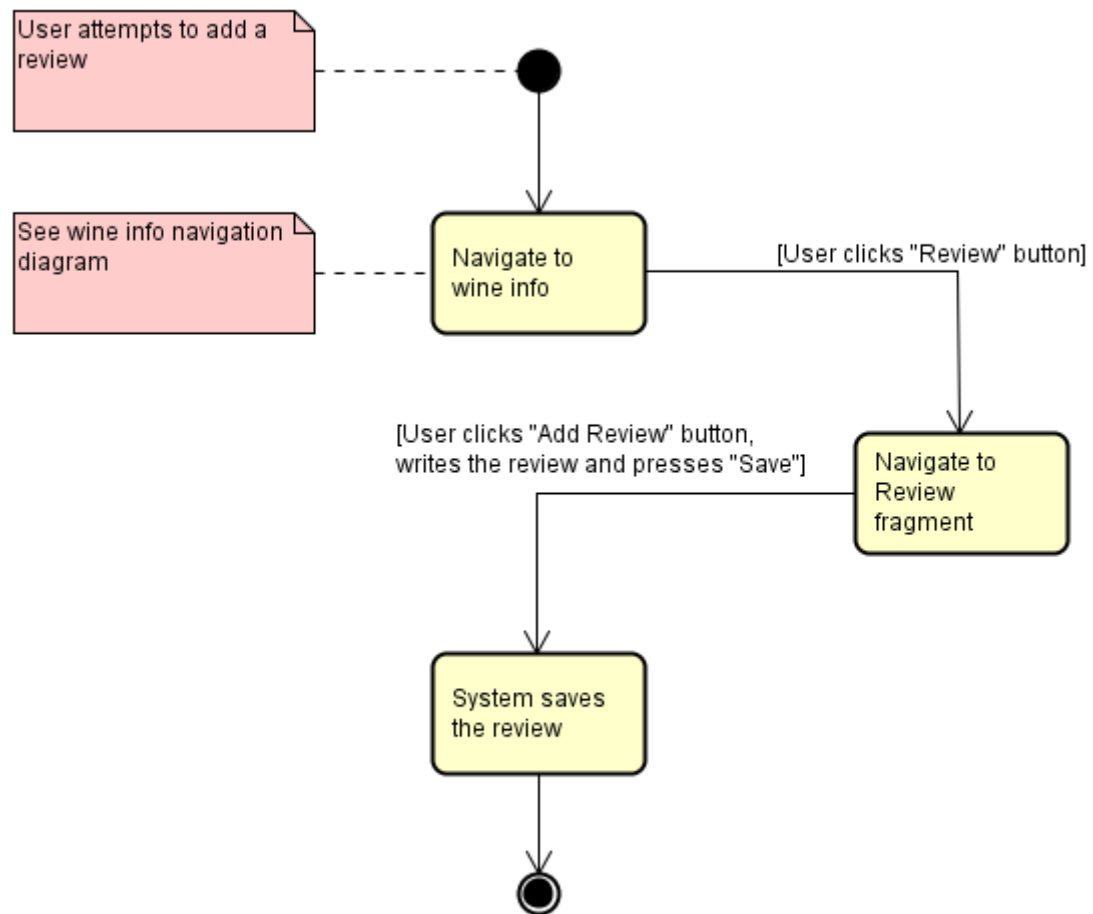


Figure 16 Add Review Activity Diagram

Get Matching Activity Diagram

The system's core feature is used in the *Matchmaking* section of the application. It enables the user to employ the server-side data mining model to match food with wine.

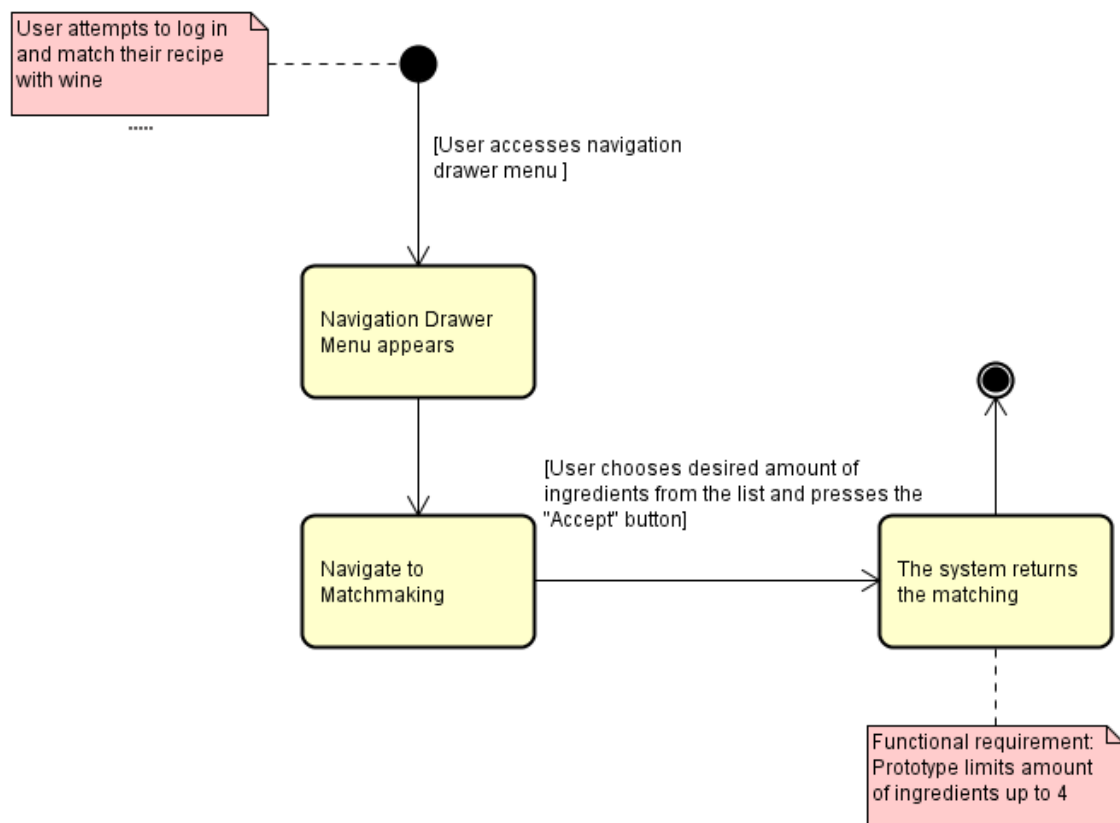


Figure 17 Get Matching Activity Diagram

Insert Matching Activity Diagram

The other side of the proverbial coin for *Matchmaking* is insert matching. This feature allows the users to create matchings they deem appropriate and insert them into the database for themselves and others to access.

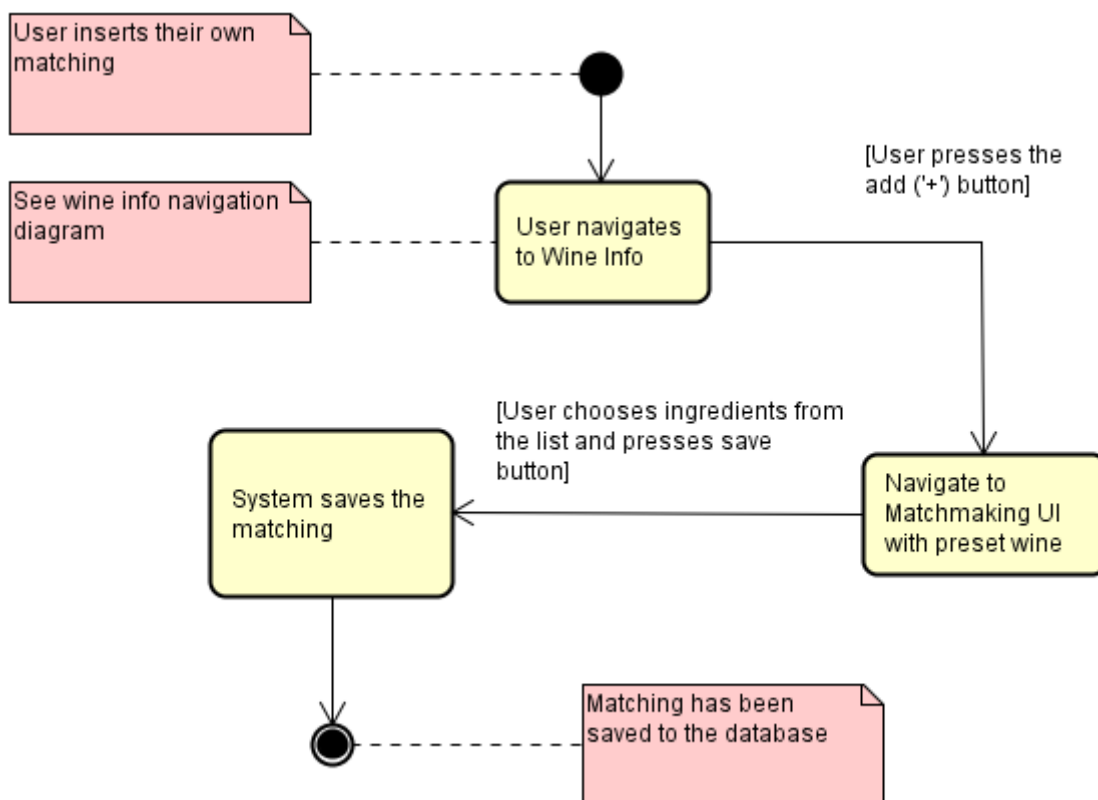


Figure 18 Insert Matching Activity Diagram

Get Recommendation Activity Diagram

The auxiliary mining model equips *Tastevin*'s users with the ability to retrieve products recommended to them by the system, based on their interactions.

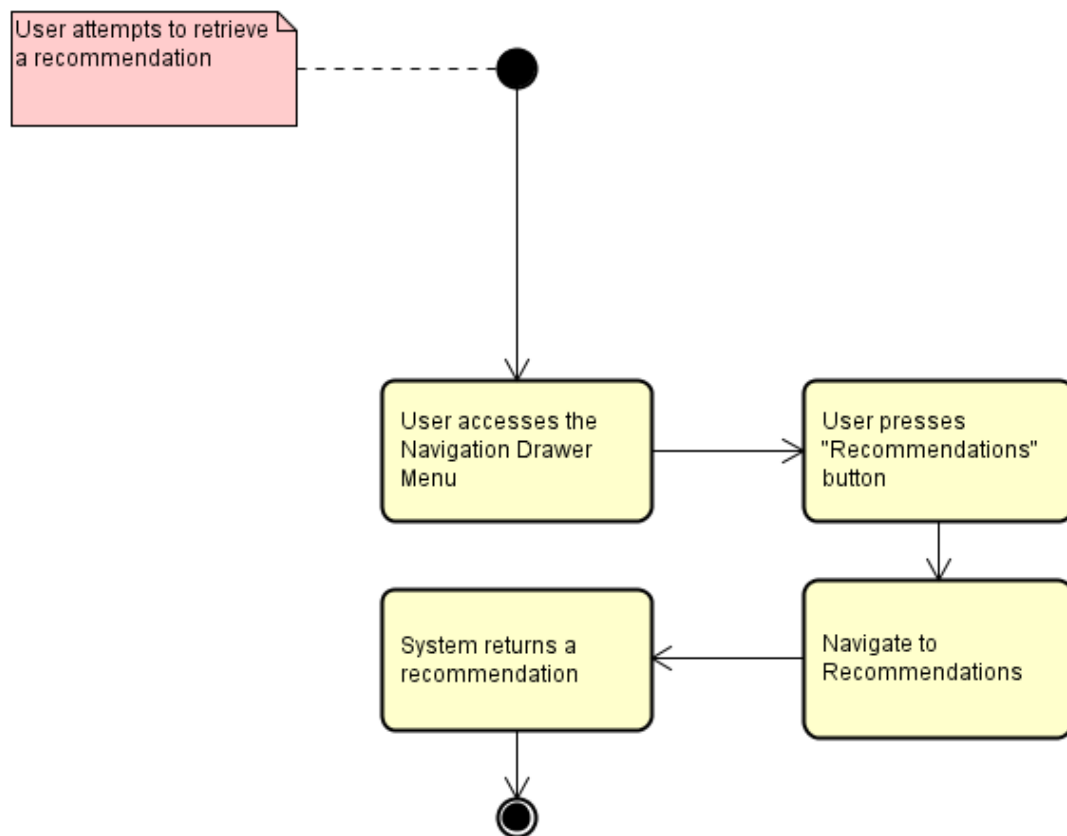


Figure 19 Get Recommendation Activity Diagram

5.7.2. Implementation

The View aspect is divided into several subsections that formulate the final front-end content, generated from layout .xmls. The prevalent chosen layout was *LinearLayout*, often in nested structures. Thanks to this measure, the team could ensure that the elements remain in relative positions intact during perspective change, as well as organize individual containers with more ease.

The most important configuration arrangements stemmed from overlapping design choices, resulting in the structure presented in this chapter. Foremost, since the interface is accessed through activities, and then secondly by adjoining fragments which serve as sheets of colored paper one can slide onto the activity as though it was a clean slate. This is the purpose of the *Main* activity. By itself, *Main* is very much a barren.

A notable example of this is that the .xmls are arranged in a specific order due to the inclusion of a *Navigation Drawer Menu*. Resulting content placement is built by declaring three layers of extension; *content_main.xml*, *app_bar_main.xml*, *activity_main*. (Android Developer, n.d.)

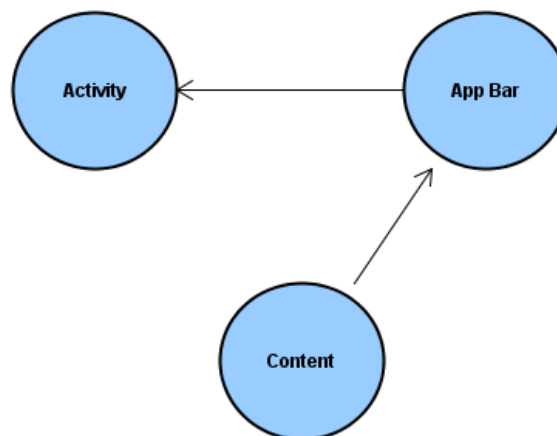


Figure 20 Fronted content structure diagram

These layers act like a metaphorical Matryoshka dolls, with the activity containing the app bar and it in turn containing the content (this is of course achieved by *include* statements within those files).

As a side note, the fragment .xmls act in the way the lastly mentioned content files, however they are passed within the back code. The remaining files concern the presentation of *listItems* included, being the smallest doll in the stack. They appear in several fragments and require to be adapted into *lists*.

Full list of frontend XML files:

Activities:

- activity_cart
- activity_login
- activity_main
- activity_shopping_cart
- activity_wine_info

Fragments:

- fragment_add_matching
- fragment_add_review
- fragment_bottles_nbr_dialog
- fragment_dashboard
- fragment_filter
- fragment_get_matching
- fragment_sub_wine_info
- fragment_reviews
- fragment_recommendations
- fragment_shop

List items:

- list_item_matching
- list_item_purchases
- list_item_recommendations
- list_item_reviews
- list_item_wine
- spinner_item

Menu:

- nav_header_main

Due to the amount of content and frequent fidelity thereof, this chapter will from now on cover only an example of the implementation;

Dashboard is the first fragment loaded into *Main* activity, with the .xml depicted on the figure below.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <ListView
        android:id="@+id/list_dashboard"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:divider="@color/search_layover_bg"
        android:dividerHeight="8dp"/>

</LinearLayout>
```

Figure 21 Example frontend XML

Taking a look in the *DashboardView* class inflating it, one can see that it is being implemented, as in the case of all other *View* classes, by implementing its respective interface.

```
public interface DashboardViewInterface {

    Activity getActivity();
    void displayToast(String text);
    void showActivities(ArrayList<Object> activities);
}
```

Figure 22 Example frontend View interface

The class itself sustains on these extended interface methods as well as:

- *onCreateView()*
- *onViewCreated()*
- *createAlertDialogWines()*

Where, the first method handles the class behavior upon being called. The layout is being inflated, *listView* holding user activities (such as reviews and purchases) and the *Presenter* object are being declared.

```
@Nullable
@Override
public View onCreateView(@NonNull LayoutInflater inflater, @Nullable ViewGroup container, @Nullable Bundle savedInstanceState)
{
    View layout = inflater.inflate(R.layout.fragment_dashboard, root: null);
    listView = layout.findViewById(R.id.list_dashboard);
    dashboard_presenter_interface = new DashboardPresenter( dashboard_view_interface: this);
    return layout;
}
```

Figure 23 Example frontend OnCreate method

In *onViewCreated*, these objects come into play, with the *Presenter* retrieving the list of activities and *listView* onclick event handlers being defined.

```
@Override
public void onViewCreated(@NonNull View view, @Nullable Bundle savedInstanceState) {
    super.onViewCreated(view, savedInstanceState);

    dashboard_presenter_interface.getActivitiesList();

    listView.setOnItemClickListener((parent, view, position, id) -> {
        if(parent.getItemAtPosition(position) instanceof Review)
            dashboard_presenter_interface.checkWine(((Review) parent.getItemAtPosition(position)).getId());
        else if(parent.getItemAtPosition(position) instanceof Purchase)
        {
            final Purchase item = (Purchase) parent.getItemAtPosition(position);
            createAlertDialogWines(item);
        }
        else if(parent.getItemAtPosition(position) instanceof Matching)
            dashboard_presenter_interface.checkWine(((Matching) parent.getItemAtPosition(position)).getWineID());
    });
}
```

Figure 24 Example frontend OnViewCreated method

Whereas, the final method outside of the interface scope handles class behavior upon an event of user attempting to access a wine from another user's purchase, in case of that purchase containing references to multiple acquired products. This situation does not occur with any other types of user activities, since both review and matching can only refer to one wine. Hence, the method parameter is of the type *Purchase*. As one might infer from the name of the method, this is handled by displaying an alert dialog with a list of items purchased, allowing the user to choose the one they were interested in.

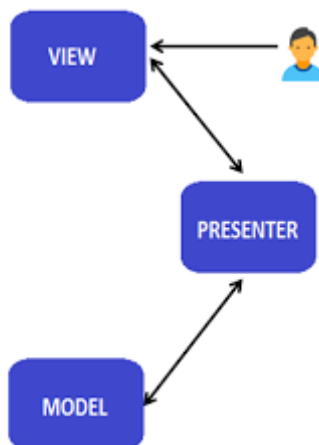
```
public void createAlertDialogWines(final Purchase purchase)
{
    builder = new AlertDialog.Builder(getContext());
    final int[] checked = {0};
    builder.setTitle("Do you want to check a wine of this purchase?");
    ArrayList<String> temp = new ArrayList<>();
    for(int i = 0; i < purchase.getWineList().size(); i++)
    {
        if(!temp.contains(purchase.getWineList().get(i)))
            temp.add(purchase.getWineList().get(i));
    }
    String[] items = new String[temp.size()];
    temp.toArray(items);
    builder.setSingleChoiceItems(items, checked[0], (dialog, which) -> {
        checked[0] = which;
    });
    builder.setPositiveButton( text: "Ok", (dialog, which) -> {
        dashboard_presenter_interface.checkWine(purchase.getWineArrayList().get(checked[0]).getId());
    });
    builder.setNegativeButton( text: "Back", (dialog, which) -> { dialog.dismiss(); });
    AlertDialog alertDialog = builder.create();
    alertDialog.show();
}
```

Figure 25 Example frontend createAlertDialogWines method

The remaining methods from interface handle three basic utilities: returning the entire *View*, displaying informative *Toast* if need be and most importantly retrieving the list of user activities retrieved from *DashboardAdapter*, which provides all three varieties in a neat package, and most importantly allows for them to be added to the *listView* interchangeably.

5.8. Android Backend

5.8.1. Design



The client logic has been implemented in adherence to the Model-View-Presenter design pattern, allowing for higher modularization of the code by decoupling individual components, easier debugging and organization of the architectural structure at hand. At the base of class hierarchy of any MVP extending system is the Model, acting in a similar way as a data source. Further up the ladder, the Presenter handles the logical aspects of code functionality handling background tasks, invoking methods, events, in-app navigation and transferring data to be displayed to the user towards the next component: the View. This final element handles displaying and initialization of presentation elements, thus controlling the actual GUI. Each of these elements will have been to be outfitted with interfaces for cross-component communication (blog.roketinsights, 2017). With the outlines of the system being too large for a readable class diagram, the figure below models the system using a

Figure 26

Package Diagram.(modeliosoft, n.d.)

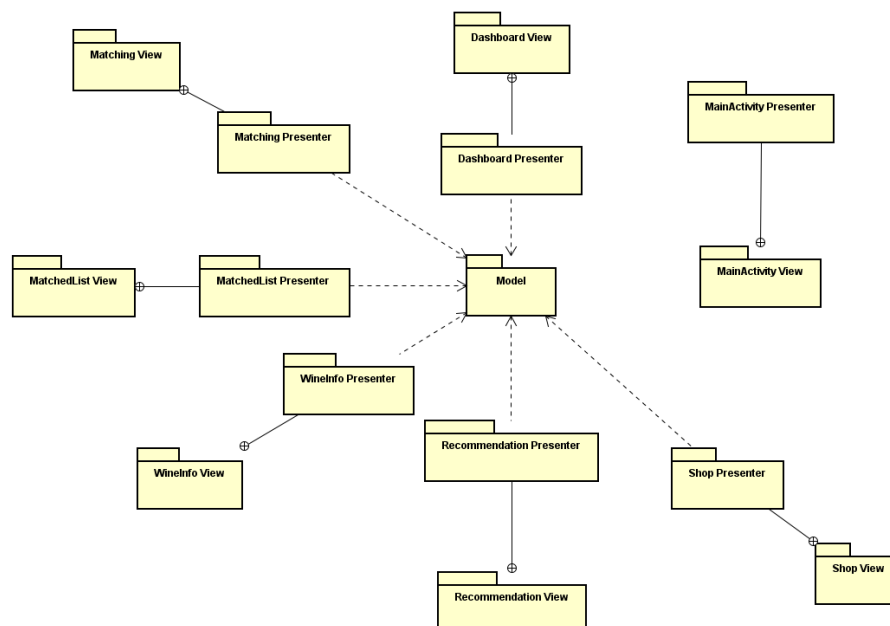


Figure 27 Client Package Diagram

5.8.2. Implementation

- **ClientRequests**

For the requests to the server, the ClientRequests class was created. This class makes use of an HTTP library called Volley and specifically of the StringRequest, JsonRequest and RequestQueue.

The RequestQueue is initialized in the constructor of the class and it gets the Context of the Activity/Fragment. The requests that will be made from all the other methods will be put in this queue and then they will be executed.

In this class there are basically two types of requests: GET and POST. For the GET request, the StringRequest is used and for the POST request the JsonRequest is used.

The methods that this class contains are:

- login
- getWineInfo
- getWineList
- getRecommendationList
- getMatchedList
- getReviewList
- getFriendList
- insertRating
- insertMatching
- insertPurchase

In this section, two methods, one for each request, will be taken as example and explained.

- **getWineInfo**

The getWineInfo method is used to retrieve all the information of a particular wine. As arguments it takes the id of the wine, a response listener and an error listener. The two listeners are always implemented in the classes where this method is used, in order to avoid synchronization problems.

The method contains a String array of the headers needed for the requests. In this array at the even indexes (e.g. 0, 2 etc.) there are the key values and at the odd positions there are their values. All the requests have the key value "REQUEST" because it's the value used to identify the type of request when it arrives to the server (See Network Server).

Then some of the request, as the one taken in case, have other headers like "NBR" that is used in the database query. In this case it's the id of the wine that the user would like to see. The String url contains the IP address of the server and the port. With all these elements the StringRequest is created and added to the Requestqueue.

```
public void getWineInfo(int id, Response.Listener<String> listenerResponse,
                        Response.ErrorListener errorListener){
    final String[] headers = {"REQUEST", "WINEINFO", "NBR", String.valueOf(id)};
    String url = "http://68.66.253.202:8080/";
    StringRequest stringRequest = new StringRequest(Request.Method.GET, url, listenerResponse, errorListener)
    {
        @Override
        public Map<String, String> getHeaders() {
            Map<String, String> params = new HashMap<>();
            params.put("Authorization", "Your authorization");
            for(int i = 0; i < headers.length; i++)
                params.put(headers[i], headers[++i]);
            params.put("cache-control", "no-cache");
            return params;
        }
    };

    queue.add(stringRequest);
}
```

Figure 28 GetWinInfo method

- **insertRating**

The insertRating method is used to insert a review of the user in the database. As before, it takes three arguments but this time it takes a Review object, instead of an id. In this method, the JsonRequest is used and that is the reason why the Review info are taken and transferred to a JSONObject that will then be passed to the request that will then be passed to the RequestQueue.

```
public void insertRating(Review review, Response.Listener<JSONObject> listenerResponse,
                        Response.ErrorListener errorListener)
                        throws JSONException
{
    final String[] headers = {"REQUEST", "INSERTRATING"};
    String url = "http://68.66.253.202:8080/";
    JSONObject json = new JSONObject();
    json.put( name: "text", review.getComment());
    json.put( name: "rating", review.getRating());
    json.put( name: "userID", review.getUserID());
    json.put( name: "wineID", review.getWineID());

    JSONObjectRequest jsonObjectRequest = new JSONObjectRequest(Request.Method.POST, url, json,
                                                                listenerResponse, errorListener)
    {
        public Map<String, String> getHeaders()
        {
            Map<String, String> params = new HashMap<>();
            params.put("Authorization", "Your authorization");
            for(int i = 0; i < headers.length; i++)
                params.put(headers[i], headers[i+1]);
            params.put("cache-control", "no-cache");
            return params;
        }
    };
    queue.add(jsonObjectRequest);
}
```

Figure 29 InsertRating method

- **MainActivity**

The MainActivity is the activity that follows the successful login and it acts as a container to 5 different fragments:

- Dashboard
- Matching
- Recommendation
- Shop
- Profile


```
public boolean onNavigationItemSelected(MenuItem item) {
    int id = item.getItemId();
    if (id == R.id.nav_dashboard)
    {
        base_fragment = new DashboardView();
        floatingActionButton.setVisibility(View.GONE);
        fragmentStart();
    }
    else if (id == R.id.nav_matching)
    {
        base_fragment = new MatchingView();
        floatingActionButton.setVisibility(View.VISIBLE);
        fragmentStart();
    }
    else if(id == R.id.nav_recommendation)
    {
        base_fragment = new RecommendationView();
        floatingActionButton.setVisibility(View.VISIBLE);
        fragmentStart();
    }
    else if (id == R.id.nav_shop)
    {
        base_fragment = new ShopView();
        floatingActionButton.setVisibility(View.VISIBLE);
        fragmentStart();
    }
    else if (id == R.id.nav_profile)
    {
        base_fragment = new ProfileView();
        floatingActionButton.setVisibility(View.GONE);
        fragmentStart();
    }

    DrawerLayout drawer = findViewById(R.id.drawer_layout);
    drawer.closeDrawer(GravityCompat.START);
    return true;
}

public void fragmentStart()
{
    if(base_fragment != null)
    {
        FragmentManager fragmentManager = getSupportFragmentManager();
        FragmentTransaction ft = fragmentManager.beginTransaction();
        ft.replace(R.id.screen_area, base_fragment);
        ft.commit();
    }
}
```

Figure 30 OnNavigationItemSelected method

This activity contains a “Base” fragment that can be changed to the 5 different listed above. In order to move from fragment to fragment, a navigation drawer was implemented with 5 different options. When the user selects one of the navigation options, it will be directed to the chosen fragment. This activity was developed with the help of Navigation Drawer Activity Demo provided by Android Studio.

- **Dashboard**

The functionality of the Dashboard Fragment is to display the latest activity of friends to the user. The activities can have three types: Review, Purchase and Matching.

The Dashboard Presenter job is to send requests to the server, retrieve information and then send them to the Dashboard View. For the requests it uses the ClientRequests class and for the information stored locally it uses the SharedPreferences.

- **getActivitiesList**

The job of the method getActivitiesList is to retrieve the activities list.

The method used from the ClientRequests is getFriendList and this method gets the id of the user, stored in the Shared Preferences, and a response listener and an error listener as parameters. When the request gets the response, the JSONObject obtained is parsed by the class Parser and then the ArrayList returned is filtered by the Filter class so that all the activities will be in organized by the date in a descending order. The ArrayList at the end is passed to the view that will then pass it to the ListView. If some errors were to occur, the method will ask the view to display the message "Network Error".

```
public void getActivitiesList()
{
    Response.Listener<String> listenerResponse = (response) -> {
        try {
            ArrayList<Object> items = Filter.sortActivities(Parser.
                parseJSONActivities(response));
            dashboard_view_interface.showActivities(items);
        } catch (JSONException e) {
            dashboard_view_interface.displayToast( text: "Network Error");
        } catch (ParseException e) {
            dashboard_view_interface.displayToast( text: "Network Error");
        }
    };
    Response.ErrorListener errorListener = (error) -> {
        dashboard_view_interface.displayToast( text: "Network Error");
    };
    client.getFriendList(sharedPref.getInt( s: "userID", i: 0), listenerResponse,
        errorListener);
}
```

Figure 31 GetActivitiesList method

- **checkWine**

The method checkWine is used when the user wants to see the additional info of a specific wine listed in the activities and it takes as arguments the id of the wine that will then be passed to the Intent that will be used to start the activity WineInfo.

- **Matching**

The functionality of the Matching Fragment is to give to the user the instruments in order to be able to select a max of 4 ingredients and ask for 3 matched with those ingredients. Those matched wines will change from user to user based on their tastes.

The Matching Presenter job is to retrieve the ingredients selected from the user and then requests the server for the 3 matched wines.

In this prototype the few available ingredients were inserted in 4 Ingredients array inside the presenter of the matching.

- **getMatching**

This method is used to send the request to the server for the matching. The first thing that it does is getting the positions of the items selected by the user from the view. It then passes them through a for loop that will retrieve the id of the ingredients from the arrays present in this class. If a position is equal to 0 ("Missing"), then the number 0 is taken as ID.

It then proceeds into the building of the creation of the response listener and error listener that will then be passed to the method getMatchedList of the class ClientRequest.

In the response the JSONObject is parsed to obtain an arraylist of the wines and puts the 3 wines into an intent that will start the activity MatchedList.

```
Integer[] ingr = new Integer[position.length];
int wasSelected = 0;

for(int i = 0; i < position.length; i++)
{
    if(position[i] != 0)
    {
        switch (i)
        {
            case 0:
                ingr[i] = ingredients_1[position[i]-1].getId();
                wasSelected++;
                break;
            case 1:
                ingr[i] = ingredients_2[position[i]-1].getId();
                wasSelected++;
                break;
            case 2:
                ingr[i] = ingredients_3[position[i]-1].getId();
                wasSelected++;
                break;
            case 3:
                ingr[i] = ingredients_4[position[i]-1].getId();
                wasSelected++;
                break;
        }
    }
    else
        ingr[i] = 0;
}
```

Figure 32 Ingredient switch case

- **Recommendations**

The Recommendation Presenter's job is to retrieve the 3 wines from the server.

- **getWinesList**

This method is used to send the request to the server in order to retrieve the wines. The method gets only the id of the user and creates the Response Listener and Error Listener these three will be passed to the ClientRequest method getRecommendationList.

In the Response Listener the JSONObject obtained is then parsed to get the ArrayList of wines and then it's passed to the view in order to be displayed.

- **addToCart**

This method is used when the user wants to add a wine to the cart. All the information of the cart are stored inside the Shared References.

The method will create or update the value of the number of wines present in the cart and it will store or update the info of the wines that includes: number of bottles selected, name of the wine, id of the wine, price of the wine. At the end will create or update the value of the total price of the cart.

```
public void addToCart(Wine wine, int nbr)
{
    SharedPreferences.Editor editor = sharedPref.edit();
    int index = sharedPref.getInt( S: "winesNbr", 0);
    boolean wasPresent = false;
    for(int i = 0; i < index; i++)
    {
        if(sharedPref.getString( S: "wineName"+(i+1), S: "Null").equals(wine.getName()))
        {
            wasPresent = true;
            int number = sharedPref.getInt( S: "wineNbr"+(i+1), 0);
            editor.putInt( S: "wineNbr"+(i+1), number+nbr);
            float tot = sharedPref.getFloat( S: "winesTOT", 0);
            editor.putFloat( S: "winesTOT", tot+(wine.getPrice()*nbr));
            editor.commit();
            break;
        }
    }
    if(wasPresent == false)
    {
        editor.putInt( S: "winesNbr", (index+1));
        editor.putInt( S: "wineID"+(index+1), wine.getId());
        editor.putInt( S: "wineNbr"+(index+1), nbr);
        editor.putString( S: "wineName"+(index+1), wine.getName());
        editor.putFloat( S: "winePrice"+(index+1), wine.getPrice());
        float tot = sharedPref.getFloat( S: "winesTOT", 0);
        tot += wine.getPrice() * nbr;
        editor.putFloat( S: "winesTOT", tot);
        editor.commit();
    }
}
```

Figure 33 AddToCart method

- **Shop**

The functionality of the Shop Fragment extends the one from the Recommendation, with the only difference that here the user has access to all the wines available in the store.

- **filter & filterSearch**

These two methods are used when the user wants to filter the list by options or searchText. These methods simply created a new ArrayList, filtered by the Filter class and then passes it to the view.

- **WineInfo**

The job of the WineInfo Presenter is to send all the information of the wine to the view and to manage the adding matching funtion.

- **addMatching**

This method is similar to the getMatching of the Matching Fragment, but the only difference is that in this case the method it's doing a POST request in order to insert in the database the matching of the user.

In the Response Listener, if the JSONObject obtained contains the word "Success" then is will ask the view to display "Matching added", otherwise it will display "Network Error".

- **getExtras**

```
public void getExtras(Intent intent)
{
    Response.Listener<String> listenerResponse = (response) → {
        try {
            wine = Parser.parseJSONWine(response);
            wine_info_view_interface.createInfo(wine);
            wine_info_view_interface.fragmentOne();
        } catch (JSONException e) {
            wine_info_view_interface.displayToast( text: "Network Error");
        }
    };
    Response.ErrorListener errorListener = (error) → {
        wine_info_view_interface.displayToast( text: "Error Network");
    };
    int id = intent.getExtras().getInt( key: "id");
    parse.getWineInfo(id, listenerResponse, errorListener);
}
```

Figure 34 GetExtras method

Since this activity is called when the user wants to check out a particular wine, this method simply gets the extras from the previous activity that contains the id of the wine and then it requests the server for the information related to that wine.

- **fragments**

This method is used when the user wants to travel from one sub-fragment to another. It prepares a bundle for that activity with all the information they need: the SubWineInfo will need all the remaining info of the wine to display and the WineReviews will take the id of then wine in order to be able to requests the list of reviews to the server.

- **SubWineInfo**

The job of the SubWineInfo Presenter is to simply to get the data from the budle received from the start and then send it to the view.

- **WineReviews**

The job of the WineReviews Presenter is to do a GET request for the list of reviews from the server and to send a POST request for the insert of a review.

- **addReview**

This method takes the rating and the eventual comment inserted from the user and send them to the server through a POST request. The method checks for any mistake first (e.g. length of

the comment too long or the rating was not selected). Passed the first checks, the methods then build the Review object and then sends it to class ClientRequest, with the two listeners.

If the Response listener retrieves a JSONObject that contains the word "Success" then the presenter asks to the view to display "Review Added", otherwise it will display "Network Error".

```
public void addReview(int rating, String comment)
{
    if(comment.length() > 200)
        fragment_wine_reviews_view_interface.displayToast( text: "Too many words");
    else if(rating == 0)
        fragment_wine_reviews_view_interface.displayToast( text: "Select a rating");
    else
    {
        SharedPreferences sharedPref = fragment_wine_reviews_view_interface.getActivity().
            getSharedPreferences( name: "com.tastevin.android.tastevin.PREFERENCE_FILE_KEY",
                Context.MODE_PRIVATE);
        int userID = sharedPref.getInt( s: "userID", i: 0);
        Review review = new Review(rating, comment, id, userID);
        Response.Listener<JSONObject> listenerResponse = (response) -> {
            if(response.toString().contains("Success"))
                fragment_wine_reviews_view_interface.displayToast( text: "Review added");
            else
                fragment_wine_reviews_view_interface.displayToast( text: "Network Error");
        };
        Response.ErrorListener errorListener = (error) -> {
            fragment_wine_reviews_view_interface.displayToast( text: "Network Error");
        };
        try {
            parse.insertRating(review, listenerResponse, errorListener);
        } catch (JSONException e) {
            fragment_wine_reviews_view_interface.displayToast( text: "Network Error");
        }
    }
}
```

Figure 35 AddReview method

- **getReviews**

As mentioned before, this fragment receives the id of the wine in order to be able to get the review list and this method retrieves the id and then sends the request to the server. In the response listener, the JSONObject is parsed into an ArrayList of reviews and it's then passed to the view.

- **Cart**

The job of the cart is to manage the items in the cart, stored in the Shared Preferences, and to be able to send a POST requests to the server in order to add a purchase to the database.

- **getWinesList**

This method takes all the wine info stored in the cart and it creates an ArrayList of wines that will be then send to the View in order to be displayed.

- **deleteWine**

This method is used when the user wants to delete an item from the cart. It updates the number of wines and the total and deleted the info of that wine in the Shared Preferences.

```
public void deleteWine(int pos)
{
    int nbr, index;
    float price;

    SharedPreferences.Editor editor = sharedPref.edit();
    price = sharedPref.getFloat( S: "winePrice"+(pos+1), 0);
    nbr = sharedPref.getInt( S: "wineNbr"+(pos+1), 0);
    tot = sharedPref.getFloat( S: "winesTOT", 0);
    tot -= price*nbr;
    editor.remove("wineName"+(pos+1));
    editor.remove("wineID"+(pos+1));
    editor.remove("wineNbr"+(pos+1));
    editor.remove("winePrice"+(pos+1));
    index = sharedPref.getInt( S: "winesNbr", 0);
    editor.putInt( S: "winesNbr", (index-1));
    editor.putFloat( S: "winesTOT", tot);
    editor.commit();
    wines.remove(pos);
    cart_view_interface.showWineList(wines, tot);
}
```

Figure 36 DeleteWine method

- **Filter**

This class has static methods that are used to filter ArrayList of type Wine and sort ArrayList of type Object.

- **filter**

This method is used by the Shop Presenter when the user wants to filter the shop list. The ArrayList is copied and then the copy is passed through a series of conditional statements where the wines that don't meet the prerequisites are deleted from the list.

- **filterSearch**

This method has the methodology of the previous one. Here there is a String value that is then checked with the first letters of the wine names and if they are not equals, they are deleted from the ArrayList.

- **sortActivities**

This method uses the sort of Collections where there is a custom Comparator that compare the date of the activity 2 with the date of activity 1 and then returns the sorted ArrayList.

- **Parser**

This class has static methods that are used when the requests made from the client receives a JSONObject. It transforms the JSONObject to the destined one (e.g. Wine) that then it's going to be returned.

All of these classes listed below are used to create the object of the types used in the application.

- Wine
- User
- Review
- Purchase
- Matching
- Ingredient
- Activities

The Activities class is an abstract class and it is extended by the Matching class, the Review class and the Purchase class. The abstract method in this case is get date that gives the possibility, when the user retrieves the list of friend activities, to reorder them in a descending chronological order.

5.9. Android SDK Integrations

There are two notable SDK integrations within the system at hand:

- Facebook Login
- PayPal Checkout

Both of them were analytically straight-forward choices and don't warrant a complex chapter structure. There were multiple reasons as to why inclusion of external libraries overshadows self-implemented custom solutions, as these are well-tested and community approved development tools, simple to integrate and created by teams of experienced professionals. It is worth noting that originally the prototypical PayPal integration was to be conducted via inclusion of Braintree which, unfortunately was not possible (for further information please refer to the *Discussion* chapter of this document). It is strongly recommended that this project's replication for market purposes to be conducted using the Braintree SDK, as at the time of development, PayPal SDK is deprecated and not supported for new applications.

5.9.1. Facebook Login

The Android application's login and signup functionalities were handled using this SDK. Its integration requires modification of only a few files. Firstly, one has to setup a Facebook developer account for their application and retrieve the hash key (Facebook, n.d.). Secondly, the manifest has to allow access to internet through the addition of these lines;

```
<uses-permission android:name="android.permission.INTERNET" />
<meta-data
    android:name="com.facebook.sdk.ApplicationId"
    android:value="@string/facebook_app_id" />
```

Figure 37 Facebook manifest data

The latter figure references meta-data taken from the Facebook Developer page.

Continuing on, one must add library references to their Gradle build files;

```
implementation 'com.facebook.android:facebook-android-sdk:4.28.0'
implementation 'com.squareup.picasso:picasso:2.5.2'
```

Figure 38 Facebook Gradle build data

with the latter being responsible for displaying user avatar on the login screen. The login page design requires a button for the user to initiate a connection to Facebook, which is handled by the `loginButton.registerCallback()` taking `FacebookCallback` as parameter method.

```
loginButton.registerCallback(callbackManager, new FacebookCallback<LoginResult>() {

    @Override
    public void onSuccess(LoginResult loginResult) {

        GraphRequest request = GraphRequest.newMeRequest(loginResult.getAccessToken(),
            new GraphRequest.GraphJSONObjectCallback() {
                @Override
                public void onCompleted(JSONObject object, GraphResponse response) {

                    Log.d( tag: "response", response.toString());

                    //accessing the picture and name
                    getFacebookData(object);
                    Intent intent = new Intent( packageContext: LoginActivity.this,
                        MainActivity.class);
                    startActivity(intent);
                }
            }
        );
    }
});
```

Figure 39 Facebook Callback Method

As shown on the figure above, the *LoginResult* object retrieves the access token supplied in a JSON object in nested callback. *getFacebookData()* method is called to store user data such as their picture and name, for the login screen. Upon a successful login attempt, the system navigates to *MainActivity*, from where the user can access the rest of the system, and the full response is stored in log files. In case a generated access token has already been received for this user, the following code is executed:

```
if(AccessToken.getCurrentAccessToken() != null){
    nameTextView.setText(AccessToken.getCurrentAccessToken().getUserId());
}
```

Figure 40 Facebook Access Token

where, the textView is populated actually with username, referred within the response JSON as *UserId*. The entire process is handled by the *callbackManager* declared in the creator method.

```
callbackManager = CallbackManager.Factory.create();
```

Figure 41 Facebook Callback Manager

5.9.2. PayPal Integration

Firstly, one is required to create a PayPal business account. In case of the prototype, a sandbox account has been used (Paypal, n.d.). This account provides a bundle of information necessary to implement PayPal integration (client id and environmental specification). Please note, as with Facebook integration, the manifest requires internet permissions as well as;

```
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

Figure 42 PayPal Manifest Permission

Secondly, the Gradle build files require references to packages used within the code;

```
implementation 'com.paypal.sdk:paypal-android-sdk:2.15.3'
implementation 'com.android.volley:volley:1.1.1'
implementation 'com.google.code.gson:gson:2.8.2'
```

Figure 43 PayPal Gradle build data

Where the first line references PayPal SDK, the second is an HTTP library used in Android environment and the last is a serialization library used to convert Java objects into JSON and back.

The account data is located into global final variables in a *Config.java* class, for ease of access (purely optional design choice), and retrieved henceforth in the following declaration/initialization;

```
private static final int PAYPAL_REQUEST_CODE = 7171;
private static PayPalConfiguration config = new PayPalConfiguration()
    .environment(PayPalConfiguration.ENVIRONMENT_SANDBOX)
    .clientId(Config.PAYPAL_CLIENT_ID);
```

Figure 44 PayPal Config data

Next, the code needs to run the PayPal service, as shown in both the creator and destructor, thus concluding the setup section;

```
@Override
protected void onDestroy() {
    stopService(new Intent( packageContext: this, PayPalService.class));
    super.onDestroy();
}

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_pay_pal2);

    //Start PayPal service, note the importance of Config.java
    Intent intent = new Intent( packageContext: this, PayPalService.class);
    intent.putExtra(PayPalService.EXTRA_PAYPAL_CONFIGURATION, config);
    startService(intent);
}
```

Figure 45 PayPal onDestroy and OnCreate methods

The PayPal button *OnClickListener()* calls the *processPayment()* method, which specifies:

- the intent of the payment (confusingly intent meaning to sell an item, rather than an Android intent),
- the amount and currency of money being transferred, i.e. product price,
- connection configuration elements taken from *Config.java*,
- type of payment,
- execution call of the *PaymentActivity*, provided in PayPal SDK.

The final method required for this integration is *onActivityResult*, which acts as the *PaymentActivity* return statement. Its structure is based on nested conditional statements, wherein if the request code provided is correct, the logic analyzes three possible outcomes; ok, cancelled, extras invalid, where “extras” pertain to data describing the transaction in the previous method. In case of the latter two outcomes, the system displays Toasts describing either a “cancelled” or “invalid” transaction. Upon the final, interesting outcome a *PaymentConfirmation* object is being created and converted to a JSON object, which is in turn converted to a String object and dissected to display to the user.

5.10. Network Analysis

Thanks to standardized industry protocols, network analysis on minor systems is relatively straightforward. From the outset it had been clear that the network had to connect the client and the server via HTTP. For data-related reasons only two methods were needed, and they were chosen to be GET and POST, to respectively retrieve and send data to the server.

Their usage is represented on the diagrams below:

5.10.1. GET

The purpose of this method is requisitioning responses carrying over data to the client. The prime example of that is depicted here, with the system's acquisition of matching data.

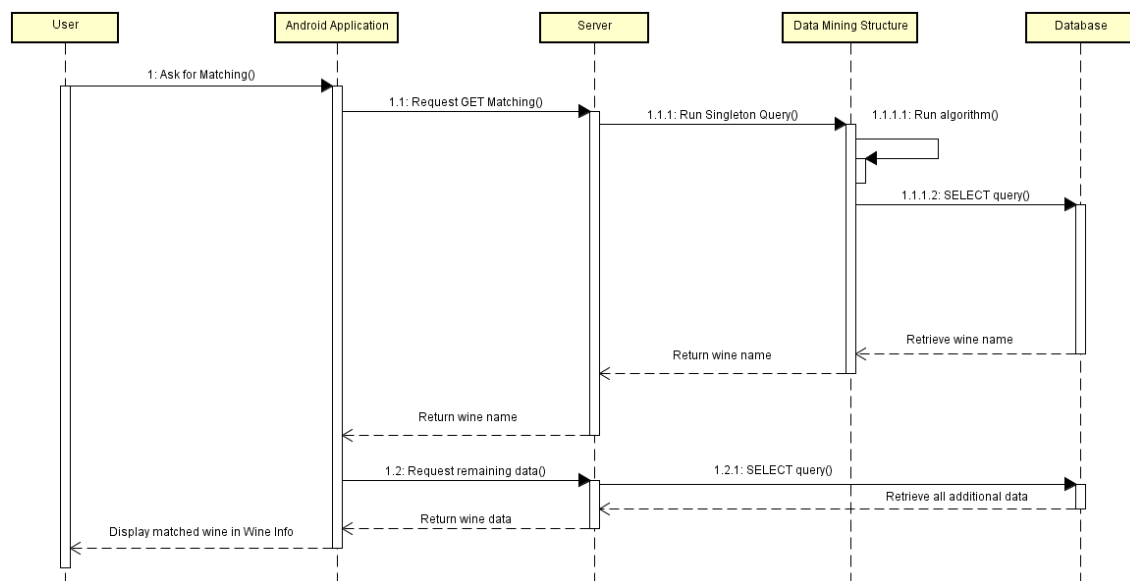


Figure 46 HTTP GET Sequence Diagram)

POST

Carrying over data from the client side made this method come in handy. The insert matching functionality discussed in previous chapters required the network to support this type of communication.

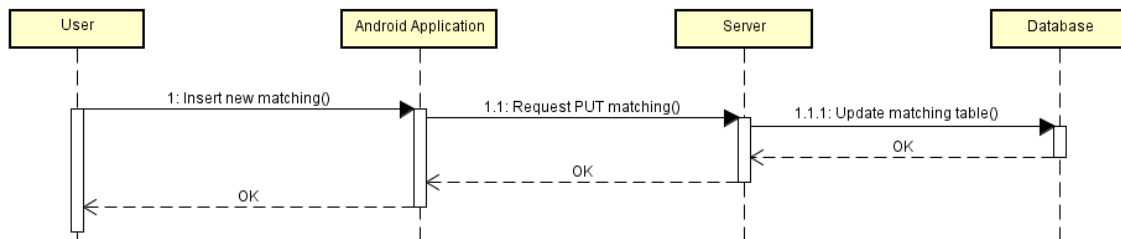


Figure 47 HTTP POST Sequence Diagram

5.10.2. Prototype GET

Factoring into this particular aspect of the system, financial constraints did not allow for IIS setup, a prerequisite for the ideal GET implementation. Because of that, the team had to cut out the SSAS connection and come up with a feasible work around by hardcoding the SSAS functionality into the database.

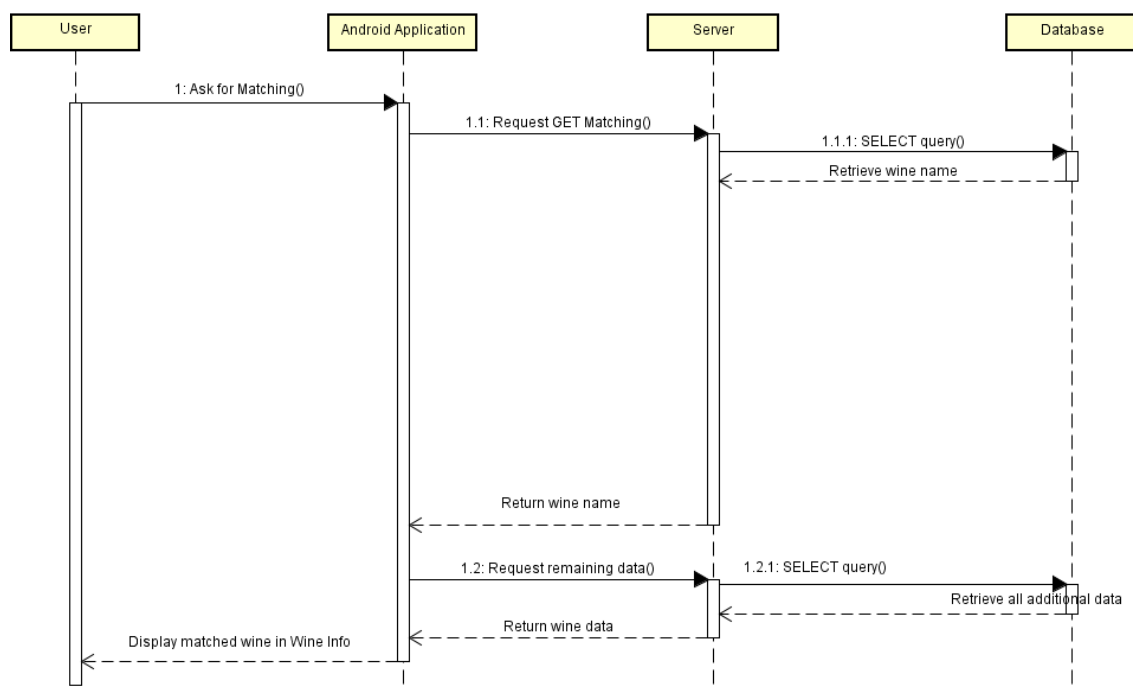


Figure 48 Prototype HTTP GET Sequence Diagram

5.11. Network Server

5.11.1. Design

The server was designed using Three-Tier-Architecture and Data Access Object design pattern.

- *DB* - class used as Data Tier and Access Object; handles all data manipulation.
- *Util* - Business Logic Tier.
- *Server* - Logic Tier handling connections to the client, which in turn acts as the Presentation Tier.

Please refer to the *Appendix* for detailed server-side UML diagrams.

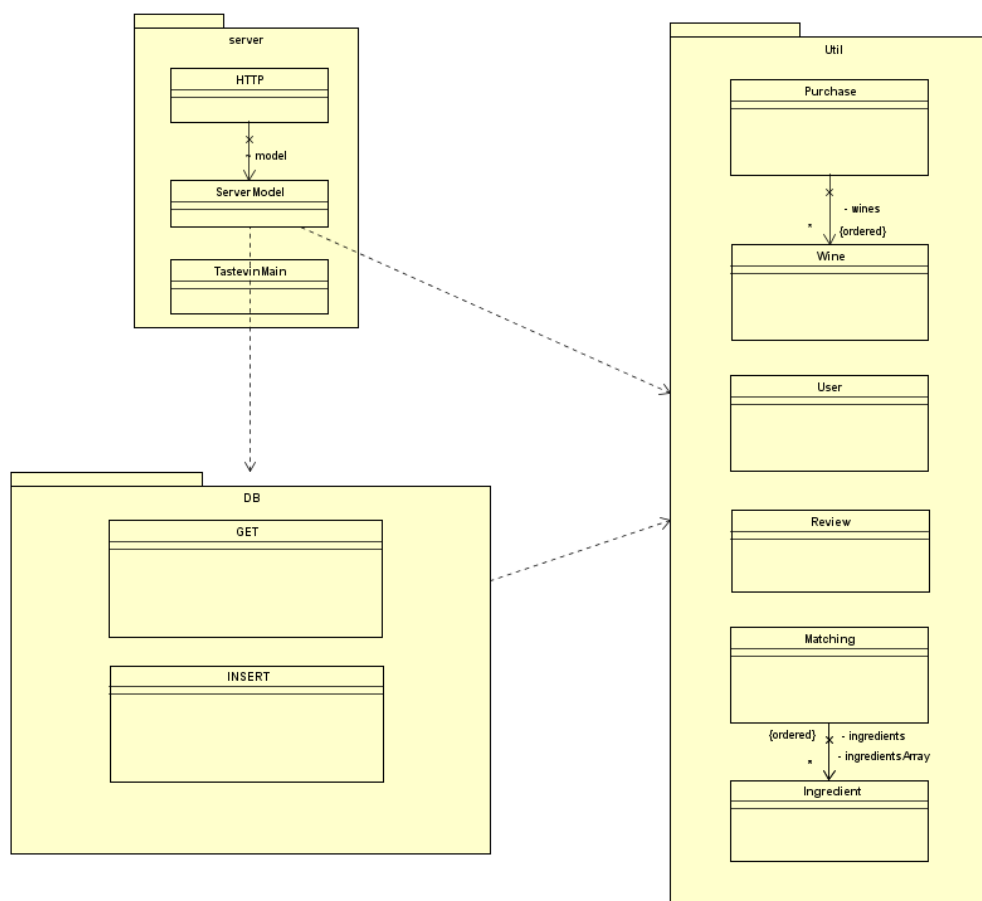


Figure 49 Server UML Class Diagram

5.11.2. Implementation

The server uses the protocol HTTP to communicate with the client. It waits for a request (GET or POST), it processes it and then send the proper response back to the client.

Below are listed the classes used by the server.

- **GET**

The GET class is used for all the SELECT queries to the database.

- **getUserInfo**

The getUserInfo takes the id of the user as parameter and it's then used in the SQL query

*SELECT * FROM dbo.[User] WHERE id = id(parameter).*

After getting the data, an object of type User is returned by the function.

- **getWineList**

The getWineList asks simply to the database the list of the wines to be then passed for the shop in the application using the query

*SELECT * FROM dbo.Wines.*

At the end, an ArrayList of type Wine is returned.

- **getWine**

The getWine takes the id of the wine as parameter and it's then used in the query

*SELECT * FROM dbo.Wines WHERE id = id(parameter).*

After getting the data, an object of type Wine is returned by the function.

- **login**

The login takes the mail sent by the client when he/she logs in as parameter and is used in the query

*SELECT * FROM dbo.[User] WHERE email = email(parameter).*

After getting the data, an object of type User is returned by the function.

- **getReviewList**

The getReviewList takes the id of the wine to get a list of its reviews using the query

```
SELECT * FROM dbo.Rating_Info WHERE wineID = id(parameter).
```

At the end, an ArrayList of type Review is returned.

- **getLatestFriendsReviewList**

The getLatestFriendsReviewList takes the id of the user to get the latest friends activities regarding the reviews using the query

```
SELECT TOP 5 dbo.Rating_Info.* FROM dbo.Friends INNER JOIN dbo.Rating_Info ON  
dbo.Friends.userID2 = dbo.Rating_Info.userID WHERE dbo.Friends.userID1 =  
id(parameter) ORDER BY dbo.Rating_Info.date DESC.
```

At the end, an ArrayList of type Review is returned.

- **getLatestFriendsPurchasesList**

The getLatestFriendsPurchasesList takes the id of the user to get the latest friends activities regarding the purchases using the query

```
SELECT TOP 5 dbo.Purchases_Info.id,  
dbo.Purchases_Info.userID, dbo.Purchases_Info.wineID, dbo.Purchases_Info.firstName,  
dbo.Purchases_Info.lastName, dbo.Purchases_Info.name, dbo.Purchases_Info.date,  
dbo.Purchases_Info.price FROM dbo.Friends INNER JOIN dbo.Purchases_Info ON  
dbo.Friends.userID2 = dbo.Purchases_Info.userID WHERE dbo.Friends.userID1 =  
id(parameter) ORDER BY dbo.Purchases_Info.date DESC.
```

At the end, an ArrayList of type Purchases is returned.

- **getLatestFriendsMatchingList**

The getLatestFriendsMatchingList takes the id of the user to get the latest friends activities regarding the purchases using the query

```
SELECT TOP 16 dbo.Matchings.* FROM dbo.Friends INNER JOIN dbo.Matchings ON  
dbo.Friends.userID2 = dbo.Matchings.userID WHERE dbo.Friends.userID1 =  
id(parameter) ORDER BY dbo.Matchings.date DESC.
```

At the end, an ArrayList of type Matching is returned.

- **getRecommendedWine**

The getRecommendedWine takes the id of the user to get the 3 recommended wines based on his/her tastes. It first uses the method getUserInfo to get a user object. Then it runs the query

```
SELECT Prob1, Prob2, Prob3, Prob4, Prob5, Prob6, Prob7, Prob8, Prob9, Prob10,
Prob11, Prob12, Prob13, Prob14, Prob15 FROM dbo.Recommendation WHERE sour =
sour(user parameter) AND sweet = sweet(user parameter) AND salty = salty(user
parameter) AND bitter = bitter(user parameter) AND umami = umami(user parameter)
```

in order to get the prediction probability of all the wines for the specific tastes of the user. They are then stored in a bidimensional array that stores the probability and the id of the wine connected to it and, when it's finished, the method returns the array.

```
public static Float[][] getRecommendedWine(int userID)
{
    Connection con = null;
    Statement stmt = null;
    ResultSet rs = null;
    Float[][] probs = new Float[15][2];
    User user = getUserInfo(userID);

    try {
        Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");

        int sour = setValue(user.getSour());
        int sweet = setValue(user.getSweet());
        int salty = setValue(user.getSalty());
        int bitter = setValue(user.getBitter());
        int umami = setValue(user.getUmami());
        con = DriverManager.getConnection(url);
        String sql = "SELECT Prob1, Prob2, Prob3, Prob4, Prob5, Prob6, Prob7, Prob8, Prob9, Prob10,"
            + "Prob11, Prob12, Prob13, Prob14, Prob15 FROM dbo.Recommendation "
            + "WHERE sour = "+sour+" AND sweet = "+sweet+" AND salty = "+salty+" "
            + "AND bitter = "+bitter+" AND umami = "+umami+";";

        stmt = con.createStatement();
        rs = stmt.executeQuery(sql);
        while (rs.next())
        {
            for(int i = 0; i < 15; i++)
            {
                probs[i][0] = rs.getFloat("Prob"+(i+1));
                probs[i][1] = (float) i+1;
            }
        }

    } catch (Exception e) {
        e.printStackTrace();
    }
    return probs;
}
```

Figure 50 Server getRecommendedWine method

- **getMatchedWine**

The getMatchedWine takes the id of the user and the id of the 4 ingredients that it's going to use for the matching to get the 3 recommended wines based on his/her tastes. It first uses the method getUserInfo to a user object. The user tastes, before that they are used, they are adapted to the data available in the database. That means that all the values that are included in the range 0 – 33 will be changed to 20, the ones in the range 33 – 66 will be changed to 50 and the remaining will be 70.

After this, it runs the query

```
SELECT AVG(PROB1) AS Prob1, AVG(PROB2) AS Prob2, AVG(PROB3) AS Prob3,
AVG(PROB4) AS Prob4, AVG(PROB5) AS Prob5, AVG(PROB6) AS Prob6, AVG(PROB7) AS
Prob7, AVG(PROB8) AS Prob8, AVG(PROB9) AS Prob9, AVG(PROB10) AS Prob10,
AVG(PROB11) AS Prob11, AVG(PROB12) AS Prob12, AVG(PROB13) AS Prob13,
```

*AVG(PROB14) AS Prob14, AVG(PROB15) AS Prob15 FROM dbo.MatchProbs WHERE
(ingredients)*

in order to get the prediction probability of all the wines for those particular ingredients. They are then stored in a bidimensional array that stores the probability and the id of the wine connected to it. The method then asks for the recommended wines for the user tastes. The two probabilities are then mixed in order to simulate the predictable from the mining structure because, as it was mentioned before, the direct connection to the structure it was not possible in this prototype.

```

sql = "SELECT AVG(PROB1) AS Prob1, AVG(PROB2) AS Prob2, AVG(PROB3) AS Prob3, AVG(PROB4) AS Prob4, AVG(PROB5) AS Prob5, "
      + "AVG(PROB6) AS Prob6, AVG(PROB7) AS Prob7, AVG(PROB8) AS Prob8, AVG(PROB9) AS Prob9, AVG(PROB10) AS Prob10, AVG(PROB11) AS Prob11, "
      + "AVG(PROB12) AS Prob12, AVG(PROB13) AS Prob13, AVG(PROB14) AS Prob14, AVG(PROB15) AS Prob15"
      + " FROM dbo.MatchProbs "
      + "WHERE ";

int i;

if(ingrID.get(0) != 0)
    sql += "Ingredient" + (0+1) + " = '" + ingrName.get(0) + "' ";

for(i = 1; i < ingrID.size()-1; i++)
{
    if(ingrID.get(i) != 0)
        sql += "AND Ingredient" + (i+1) + " = '" + ingrName.get(i) + "' ";
}

if(ingrID.get(i) != 0)
    sql += "AND Ingredient" + (i+1) + " = '" + ingrName.get(i) + "' ";

stmt = con.createStatement();
rs = stmt.executeQuery(sql);

while (rs.next())
{
    for(int j = 0; j < 15; j++)
    {
        probs[j][0] = rs.getFloat("Prob" + (j+1));
        probs[j][1] = (float) j+1;
    }

    Float[][] recWine = getRecommendedWine(userID);
    Float[][] result = new Float[probs.length][2];
    for(int k = 0; k < probs.length; k++)
    {
        result[k][0] = (probs[k][0]) * recWine[k][0];
        result[k][1] = probs[k][1];
    }
    Arrays.sort(result, (a, b) -> Float.compare(b[0], a[0]));
    wines.add(getWine(Math.round(result[0][1])));
    wines.add(getWine(Math.round(result[1][1])));
    wines.add(getWine(Math.round(result[2][1])));
}

} catch (Exception e) {
    e.printStackTrace();
}
return wines;

```

Figure 51 Server getMatchedWine method

- **getRecommendations**

This method is only used to get the 3 recommended wines, using the float array given by getRecommendedWine.

- **getReviewList**

The getReviewList takes the id of the wine to get its reviews using the query

*SELECT * FROM dbo.Rating_Info WHERE wineID = id(parameter).*

After getting the data, an ArrayList of reviews is returned.

- **INSERT**

The INSERT file is used for all the INSERT queries to the database.

- **insertRating**

The insertRating takes an object of type Review as parameter and it's then used in the SQL query

INSERT INTO dbo.Reviews VALUES (text, rating, userID, wineID, date).

The function returns a String "Success" if the query was successful and a String "Failure" is the query had some errors.

```
public static String insertRating(Review review)
{
    Connection con = null;
    Statement stmt = null;
    int rows = 0;
    String result = "";

    try {
        Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");

        con = DriverManager.getConnection(url);
        String sql = "INSERT INTO dbo.Review VALUES (" + review.getComment() + ", " + review.getRating() + ", " +
            review.getUserID() + ", " + review.getWineID() + ", GETDATE())";
        stmt = con.createStatement();
        rows = stmt.executeUpdate(sql);
        if (rows > 0)
            result = "Success";
    } catch (Exception e) {
        result = "Failure";
    }
    return result;
}
```

Figure 52 Server insertRating method

- **insertPurchases**

The insertPurchases takes an object of type Purchase as parameter and it's then used in the SQL query

INSERT INTO dbo.Purchases_Mapping VALUES (wineID, purchaseID, wineNbr).

The function returns a String "Success" if the query was successful and a String "Failure" is the query had some errors.



```
public static String insertPurchases(Purchase purchase, ArrayList<Integer> wines)
{
    Connection con = null;
    Statement stmt = null;
    ResultSet rs = null;
    int id = 0;
    int rows = 0;
    String result = "";

    try {
        Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");

        con = DriverManager.getConnection(url);
        String sql = "INSERT INTO dbo.Purchases VALUES( "+purchase.getUserID()+" , GETDATE())";
        stmt = con.createStatement();
        rows = stmt.executeUpdate(sql);
        if(rows == 0)
            return "Failure";
        rows = 0;
        sql = "SELECT TOP 1 id FROM dbo.Purchases WHERE userID = "+purchase.getUserID()+" ORDER BY date DESC";
        stmt = con.createStatement();
        rs = stmt.executeQuery(sql);
        while (rs.next())
            id = rs.getInt("id");
        sql = "";
        for(int i = 0; i < wines.size(); i++)
            sql += "INSERT INTO dbo.Purchases_Mapping VALUES( "+wines.get(i)+" , "+id+" , "+(i+1)+" );\n";
        rows = stmt.executeUpdate(sql);
        if(rows > 0)
            result = "Success";
    } catch (Exception e) {
        return "Failure";
    }
    return result;
}
```

Figure 53 Server insertPurchases method

- **insertMatching**

The insertMatching takes an object of type Matching as parameter and it's then used in the SQL query

INSERT INTO dbo.Mapping VALUES (ingredientID, matchingID, ingredientNbr).

The function returns a String "Success" if the query was successful and a String "Failure" is the query had some errors.

```
public static String insertMatching(Matching matching, ArrayList<Ingredient> ingr)
{
    Connection con = null;
    Statement stmt = null;
    ResultSet rs = null;
    int id = 0;
    int rows = 0;
    String result = "";

    try {
        Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");

        con = DriverManager.getConnection(url);
        String sql = "INSERT INTO dbo.Mapping VALUES (" + matching.getWineID() + ", " + matching.getUserID() + ", GETDATE());";
        stmt = con.createStatement();
        rows = stmt.executeUpdate(sql);
        if (rows == 0)
            return "Failure";
        rows = 0;
        sql = "SELECT TOP 1 id FROM dbo.Mapping WHERE userID = " + matching.getUserID() + " ORDER BY date DESC";
        stmt = con.createStatement();
        rs = stmt.executeQuery(sql);
        while (rs.next())
            id = rs.getInt("id");
        sql = "";
        for (int i = 0; i < ingr.size(); i++)
            sql += "INSERT INTO dbo.Mapping_Mapping VALUES (" + ingr.get(i).getId() + ", " + id + ", " + (i+1) + ");\n";
        rows = stmt.executeUpdate(sql);
        if (rows > 0)
            result = "Success";
    } catch (Exception e) {
        return "Failure";
    }
    return result;
}
```

Figure 54 Server insertMatching method

- **MATCHING**

The Matching class takes information from the tables dbo.Mapping, dbo.Mapping_Mapping and dbo.Ingredient and it contains the following information:

- id, first name and last name of the User
- id of the Matching
- id and name of the Wine
- an ArrayList of the ingredients of the recipe
- date of the Matching.

The class has two constructors, the first one insert all the info of the matching and the second is used by the client for the first step of the insert of a matching.

- **PURCHASE**

The Purchase class takes information from the tables dbo.Purchases and dbo.Purchases_Mapping and it contains the following information:

- id of the purchase
- id, firstName and lastName of the User
- date of the Purchase

- **REVIEW**

The Review class takes information from the tables dbo.Review and it contains the following information:

- id and name of the User
- id and name of the Wine
- id, rating, date and comment of the review

- **USER**

The User class takes information from the table dbo.User and it contains all the basis information of a user and in addition, the personal tastes (bitter, salty, sour, sweet and umami).

- **WINE**

The Wine class takes information from the tables dbo.Wine, dbo.WineMapping and dbo.WineVines. It contains the following information:

- id
- name
- winery
- type
- price
- alcohol Content
- composition
- rating
- year
- number of reviews

- **INGREDIENT**

The Ingredient class takes information from the tables dbo.Ingredient. It contains the id and the name of the ingredient

- **SERVERMODEL**

The ServerModel analyzes the requests that receives from the client.

- **checkAction**

This method gets as arguments a String for the method, a Map<String, List<String>> for the headers and a String for the body.

It first checks if the method is a “POST” method or a “GET” method. In the first case it will call the checkActionGET and in the second case it will call the checkActionPOST. If the method is not recognized, it will just return to the user “ERROR METHOD”.

- **checkActionGET**

This method analyzes the headers sent in the request. All of the options inside this method will use only the GET class and will attach to the response that is to be sent to the client the String of JSONObject obtained.

```
switch (header.get("REQUEST").get(0).toString())
{
    case "WINEINFO":
        i = header.get("NBR").get(0).toString();
        wine = GET.getWine(Integer.parseInt(i.toString()));
        object = new JSONObject(wine);
        response += object.toString();
        break;

    case "RECOMMENDATIONLIST":
        i = header.get("NBR").get(0).toString();
        wines = GET.getRecommendations(Integer.parseInt(i.toString()));
        object = new JSONObject();
        object.append("wines", wines);
        response += object.toString();
        break;

    case "MATCHEDLIST":
        int i0 = Integer.parseInt(header.get("NBR1").get(0).toString());
        int i1 = Integer.parseInt(header.get("NBR2").get(0).toString());
        int i2 = Integer.parseInt(header.get("NBR3").get(0).toString());
        int i3 = Integer.parseInt(header.get("NBR4").get(0).toString());
        int i4 = Integer.parseInt(header.get("NBR5").get(0).toString());
        wines = GET.getMatchedWine(i0, i1, i2, i3, i4);
        object = new JSONObject();
        object.append("wines", wines);
        response += object.toString();
        break;

    case "WINELIST":
        wines = GET.getWinesList();
        object = new JSONObject();
        object.append("wines", wines);
        response += object.toString();
        break;

    case "REVIEWLIST":
        i = header.get("NBR").get(0).toString();
        reviews = GET.getReviewList(Integer.parseInt(i.toString()));
        object = new JSONObject();
        object.append("reviews", reviews);
        response += object.toString();
        break;
}
```

Figure 55 Server switch case

First of all, the first header key, is equal to "REQUEST" and its value is then checked in a switch statement, and each of the options have a different set of code to run:

- WINEINFO - This option state that the client asked for the information of a wine. It takes a second header that contains the id of the particular wine and then the method getWine.
- RECOMMENDATIONLIST - This option state that the client asked for the recommended wines for the user. This method also takes a second header that contains this time the id of the user. This time the getRecommendations.

- **MATCHEDLIST** - This option state that the client asked for some wines that could be the best match for some chosen ingredients and user's tastes. This method takes multiples additional headers that represents the 4 ingredients and the id of the user. Then all this data is passed to the method `getMatchedWine`.
- **WINELIST** - This option states that the client asked for the list of all the wines in the database. This method doesn't have additional headers and it calls the `getWinesList` method.
- **REVIEWLIST** - This option states that the client asked for the list of reviews of a specific wine. It gets an additional header that is the id of the wine and then it calls the method `getReviewList`.
- **LISTFRIENDS** - This option states that the client asked for the list of activities of his/her friends. It takes an additional method that contains the id of the user and then if calls three methods from the GET class that will return an ArrayList of type Review, an ArrayList of type Purchase and ArrayList of type Matching. It then appends them to the same JSONObject.

- **checkActionPOST**

This method analyzes the headers sent in the request. All of the options inside this method will use only the INSERT class and will attach to the response a JSONObject that will contain the word "Success" or "Failure".

- **INSERTRATING** - This option states that the client wants to add a new rating to the database. The request contains the object that will then be passed to the method `insertRating`.
- **INSERTMATCHING** - This option states that the client wants to add a new matching to the database. The request contains the object that will then be passed to the method `insertPurchases`.
- **INSERTPURCHASING** - This option states the the client wants to add a new purchase to the database. The request contains the object that will then be passed to the method `insertMatching`.



```
switch (header.get("REQUEST").get(0).toString())
{
    case "INSERTATING":
        i = body;
        object = parser.parse((String) i);
        review = gson.fromJson(object, Review.class);
        jsonObject = new JSONObject();
        result = INSERT.insertRating(review);
        response += jsonObject.append("result", result);
        break;

    case "INSERTPURCHASING":
        i = body;
        jsonObject2 = new JSONObject(body);
        array1 = jsonObject2.getJSONArray("winesArray");

        ArrayList<Integer> wines = new ArrayList<>();
        for(int j = 0; j < array1.length(); j++)
            wines.add(array1.getJSONObject(j).getInt("id"));
        Purchase purchase = gson.fromJson(body, Purchase.class);
        jsonObject = new JSONObject();
        result = INSERT.insertPurchases(purchase, wines);
        response += jsonObject.append("result", result);
        break;

    case "INSERTMATCHING":
        i = body;
        jsonObject2 = new JSONObject(body);
        array1 = jsonObject2.getJSONArray("ingredientsArray");
        ArrayList<Ingredient> ingr = new ArrayList<>();
        for(int j = 0; j < array1.length(); j++)
        {
            JSONObject jsonObject3 = array1.getJSONObject(j);
            Ingredient ingredient = gson.fromJson(jsonObject3.toString(), Ingredient.class);
            ingr.add(ingredient);
        }
        Matching matching = gson.fromJson(body, Matching.class);
        jsonObject = new JSONObject();
        result = INSERT.insertMatching(matching, ingr);
        response += jsonObject.append("result", result);
        break;
}
```

Figure 56 Server insert switch case

- **HTTP**

This class implements the `HttpHandler` and it handles the first stage of the request received from the client. It gets all the data that then it's passed to the `ServerModel` and then returns the response to the client with the results. (Feng, 2015)

- **TastevinMain**

This class only contains the main that initializes and starts the server.

5.12. Testing

The chosen testing model was manual testing. Each of the system's components has been tested using several approaches.

The mining models' accuracy of output has been tested initially via hardcoding expected outcomes, whereas further on, by directing purchase data in such a way as to arrive at preset conclusions in the algorithm output e.g. stating that 7 out of 10 available matchings for beef were to Merlot, thus the Singleton Query with 'beef' input was to yield 'Merlot' as output. Testing conditions naturally raised in their complexity as so did the model, resulting in a level of confidence with which the team had populated the database with randomized data.

The subsequent tests were simple to undertake. Connections inherently exist in two states: working and not working. This fact greatly simplified testing server connections by allowing for writing and configuring an applet to test connections between the server and the client.

The main testing issue has however been the client itself. Manual testing on applications serving multiple purposes and comprising of various integrated elements is never the most advisable approach. The team believes that the prototype is not bug free, and many more additional tests would be necessary for an actual public release, especially since there was neither the time nor resources enough to conduct full-fledged tests within test groups with specific variable tables etc.

<pre> SELECT [Matchings MM].[Wine Name] From [Matchings MM] NATURAL PREDICTION JOIN (SELECT 23 AS [Age], 'Chicken Breast' AS [Ingredient1], 'Garlic' AS [Ingredient2], 'Honey' AS [Ingredient3], 'Olive oil' AS [Ingredient4], 70 AS [Salty], 50 AS [Sour]) AS t </pre>	<pre> SELECT [Matchings MM].[Wine Name] From [Matchings MM] NATURAL PREDICTION JOIN (SELECT 23 AS [Age], 'Chicken Breast' AS [Ingredient1], 'Garlic' AS [Ingredient2], 'Honey' AS [Ingredient3], 'Olive oil' AS [Ingredient4], 70 AS [Salty], 70 AS [Sour]) AS t </pre>
<div>Wine Name</div> <div>Vermentino di Sardegna Tuvaes</div>	<div>Wine Name</div> <div>Franciacorta Rosé</div>

Figure 57 Singleton Query Test comparison

Feature	Method	Desired outcome	Actual outcome
SERVER			
Primary model	Data setup	Algorithm results agree with tendencies of the data setup.	Success.
Auxiliary model	Data setup	Algorithm results agree with tendencies of the data setup.	Success.
Server GET	Postman and custom testing applet.	GET request with custom headers results in an 'OK' response.	Success.
Server POST	Postman and custom testing applet.	POST request with custom headers results in an 'OK' response.	Success.
Dataflow	Indirect testing by allowing other components data manipulation.	Queries will perform insert and update operations as per their parameters.	Success.
Database Triggers.	Data injection trials via system components.	All query functionality will perform as desired.	Partial failure; the server is incapable of inserting review data unless trigger is disabled.
CLIENT			
Login	Manual login attempt.	Retrieval of team member's facebook data.	Success.
Get matching	Manual retrieval of the desired matching.	Retrieval of the correct wine.	Success.
Insert matching	Manual database access.	Existence of a Matching table row with correct data.	Success.



Get recommendations	Manual prototype trial with prior knowledge of the data.	System returns wines with highest probability to fit the user.	Success.
Add review	Manual database access.	Existence of a Review table tow with correct data.	Success.
Search and filter	Manual prototype trial with prior knowledge of the data.	Output list populated with items matching the testing criteria.	Success.
PayPal transactions via Braintree SDK	Manual prototype trial with prior knowledge of the data.	'OK' response and payment details matching sandbox account credentials and nonce details.	Failure due to lack of physical server connection.
PayPal transactions via PayPal SDK	Manual prototype trial with prior knowledge of the data.	'OK' response and payment details matching sandbox account credentials.	Success.
Additional minor android functions e.g. menu navigation	Manual prototype trial.	Minor features operate in accordance to their purpose.	Success.
Custom server connection	Postman and custom testing applet.	Responses carry JSON objects with requested data.	Failure.

Standardized server connection (running custom code)	Postman and custom testing applet.	Responses carry JSON objects with requested data.	Success.
Delete a wine from the cart	Manual prototype trial with prior knowledge of the data.	Item is deleted, regardless of its position within the chronological order of being added	Failure.

Figure 58 Testing table

6. Results

The development process for *Tastevin* concluded in creation of a functional prototype that fulfilled its proof of concept, thus stating the validity of its original goal. The mining model had proven more than functional with potential to grow organically with time and usage, posing for an interesting future study, provided the product owner deploys a market valid release.

Servers in many cases were patchwork, despite their code being solid, stemming from the lack of funding for an actual, non-experimental release, whereas the mobile phone application had been a resounding success, albeit the necessity to cut corners.

That being said, several of the prototype's aspects had to have been simplified or some even hardcoded. These issues and future ideas for improvement are explored and discussed in the following chapters.

7. Discussion

- Albeit the data-mining functionality of *Microsoft Decision Trees* can be achieved using *Microsoft Clustering*, the desired output is better retrieved as a single item. That being said, the dichotomy between the two is miniscule enough to allow for the system to be recreated using *Clustering*.
- In ideal time conditions the team were wishing to add several additional features into the client application, notably Google Maps integration, showing users leisure locations close to vineyards producing their favorite wines, a profile page, a *Sommelier's Guide* which would assist the user in choosing the right glass, setting the table, how to sample etc., a wine roulette and "I'm feeling lucky" search feature.
- The Facebook login requirement is somewhat limiting for users that do not indulge in social media. Thus, the team would add an in-system register and login functionalities,

as well as Gmail integration.

- Scanning QR codes and/or labels on wines, could allow for faster and more intuitive interaction between the user, the application and products.
- Provided further research, data mining models could be equipped with additional recognition functionalities, such as identifying users' locations, marital status etc.
- The database could be improved upon to more closely resemble the reality of wine cultivation and blending. Data could be provided on how the wine is ageing, separating vines from types and so on, potentially altering the results and mining less obvious associations.
- Statistical marketing information could be access by companies interested in increasing their wine revenue, provided a company client panel were to be developed.
- One of the team members has expressed their concerns towards *Insert Matching* functionality. As customers unfortunately cannot be trusted to have other users best interests in mind. This however, could be harnessed. Best user matchings can be put to leaderboards, shared and contested, providing additional interest and interactions for all users.
- Even if decision tree matchmaking is successful there are some constraints, because the algorithm has to take the ingredients in a certain order, otherwise the system will not be able to use its preset structure.
- The alert dialog in Dashboard View that displays multiple wines could be handled better in a finished product, e.g. by displaying radio buttons next to each item, that allow the user to choose multiple.
- Despite the fact that Android Studio is outfitted with several testing frameworks (e.g. Espresso), much to the team's dismay, due to time constraints, the idea of employing test-driven development had to have been discarded, resulting in having to rely on manual testing.
- What had originally been intended to be the designated payment service provider was Braintree SDK. Unfortunately, as had later transpired connectivity between android OS and (this being an educated guess) Windows OS-based servers, is prone to faulty connections. This had been also the case for the custom Java server ran prior to

Braintree integration.

In this particular case, the responsible team-member had deployed PHP logic on an Apache server ran using XAMPP distribution. The Braintree SDK package was successfully deployed and testing providing viable Sandbox client tokens required to achieve successful PayPal transactions.

Had this approach been attained, the product owner would only need to migrate to an actual business account, replacing the sandbox credentials. After full client-side implementation had been created, the thread would get stuck within Looper class, due to the callback.

The debugging process had yielded the reason for this aberrant behavior had been the lack of a valid client token, which could not be provided as the connection timed out. Both the emulator and the hardware device experienced the same issue. Despite being connected on the same wireless network, a final manual test, conducted by the attempt of retrieving the client token via browser uncovered that Android could not under these circumstances connect to the server, thus making Braintree SDK not a valid tool for the prototypical implementation. Before the tests, the following steps have been taken to implement the Braintree integration;

Firstly, as with previously mentioned SDK integrations, the team member had to download the Braintree PHP library and setup a sandbox account providing necessary information for configuring the server side and connecting to said account; (Paypal, n.d.)

```
<?php
session_start();
require_once("lib/autoload.php");
if(file_exists(__DIR__ . "../.env"))
{
    $dotenv = new Dotenv\Dotenv(__DIR__ . "../");
    $dotenv -> load();
}

Braintree_Configuration::environment('sandbox');
Braintree_Configuration::merchantID('qbcx6fcjxycb9js9');
Braintree_Configuration::publicKey('mccnf6fgygdfxn4w');
Braintree_Configuration::privateKey('26de40d791272cd38fd47907a21e6406');

?>
```

Figure 59

```
<?php
session_start();
require_once("braintree_init.php");
require_once 'lib/Braintree.php';

$nonce = $_POST['nonce'];
$amount = $_POST['amount'];
$result = Braintree_Transaction::sale([
    'amount' => $amount,
    'paymentMethodNonce' => $nonce,
    'options' => [
        'submitForSettlement' => True
    ]
]);
?>
```

Figure 60

Next up, a checkout processing file had to have been implemented.

This is achieved by creating an object called “nonce”, which acts as a request with headers describing the amount of money being paid and type of transaction.

Subsequently the *Main.php* file, which was meant to generate a client token for the transaction, was implemented and placed in the XAMPP/htdocs directory.

```
<?php
session_start();
require_once("braintree_init.php.");
require_once 'lib/Braintree.php';
echo ($clientToken = Braintree_ClientToken::generate());
?>
```

Figure 61

The client-side part of the integration required addition of several elements into the manifest and Gradle build files as depicted on the figures below;

```
<activity
    android:name="com.braintreepayments.api.BraintreeBrowserSwitchActivity"
    android:launchMode="singleTask">
    <intent-filter>
        <action android:name="android.intent.action.VIEW" />

        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.BROWSABLE" />

        <data android:scheme="${applicationId}.braintree" />
    </intent-filter>
</activity>
```

Figure 62

```
implementation 'com.braintreepayments.api:braintree:2.+'
implementation 'com.braintreepayments.api:drop-in:3.+'
```

Figure 63

The in-code part required an additional utility class to handle background processes and executions, asynchronously to the designated activity handling the payments. Divided into three methods, the class had handled requisitioning the client access token from the server in the background (or at least had been planned to, seeing as this is where testing devices failed to connect, spiraling into an infinite loop). Pre- and post-execution methods were required to be implemented by extending *AsyncTask*. Aside from their auto-generated applications, they had only displayed a *ProgressDialog* to notify the user, in case of needing to wait for the connection, as to which part thereof was currently under way.

Within the actual activity, the constructor initialized the payment's button event handler, in which the *submitPayment* method was called, in which the Braintree Activity, mentioned in the manifest was called via *startActivityForResult*, which is an important distinction, because of the remainder of the logic being conducted within *onActivityResult* and nested within *sendPayment*. The former method had handled packaging the nonce with headers, whereas the latter sending it and retrieving the response.

8. Conclusion

In conclusion, *Tastevin's* development process was satisfactory, unfortunately somewhat hurdled by lack of resources. The mining model had proven more than functional with potential to grow organically with time and usage, posing for an interesting future study, provided the product owner deploys a market valid release.

The team had many additional ideas for improvement of their final product and given the opportunity, could fine tune and build upon the content available in the prototype. In case of the product owner delegating the development of a market-ready version to a new team, the following suggestions should be considered:

SSAS deployment on IIS: since SSAS is a Microsoft service, it requires Windows Authentication which is not available in the Windows 10 Home edition.

Further research should be made into smartphone-server connectivity, as this had been the only other insurmountable error.

Test-driven development should be employed in revisiting Android development.

The system's production yielded interesting results especially in the data mining area, where the combination of the models resulted in output more suited to each individual user than either could perform in their own capacity, giving interesting ideas as to directions in which the model can grow.

Perhaps an additional model could open previously not thought of areas? Anyone with basic informatics training realizes the utility design patterns can bring to a system, and the example provided by *Tastevin* showcases that these tools can and should be utilized in tandem with data processing.

The modularity brought to the table by the fact that each model can process same data in various fashions fleshes out possibilities in their improvement and just like in the topic handled by this project, one benefits the most by intelligently combining ingredients to arrive at an outcome best suited to their respective situation. It is that situational awareness and modularity that give validity to pattern recognition usability across various fields, be it mathematics, financing or having a nice supper.

References

- Android Developer, n.d. *Dialogs*. [online] Available at:
<<https://developer.android.com/guide/topics/ui/dialogs#java>>.
- Android Developer, n.d. *Getting Results from an Activity*. [online] Available at:
<<https://developer.android.com/training/basics/intents/result>>.
- Android Developer, n.d. *Manifest Permission*. [online] Available at:
<<https://developer.android.com/reference/android/Manifest.permission>>.
- Android Developer, n.d. *Navigation Drawer*. [online] Available at:
<<https://developer.android.com/training/implementing-navigation/nav-drawer>>.
- Android Developer, n.d. *Volley*. [online] Available at:
<<https://developer.android.com/training/volley/>>.
- Anon n.d. *Software Analysis and Design*. [online] Available at:
<<http://icis.pcz.pl/~dyja/pliki/SE/lecture06.pdf>>.
- apachefriends, n.d. *XAMPP*. Available at: <<https://www.apachefriends.org/index.html>>.
- Berry Bro's & Rudd, n.d. *Food & Wine Matching Guide*. [online] Available at:
<<https://www.bbr.com/wine-knowledge/food-and-wine>>.
- blog.roketinsights, 2017. *MVP Desing Pattern*. [online] Available at:
<<https://blog.rocketinsights.com/mvp-design-pattern/>>.
- BrainTree, n.d. *Payment Method Nonces*. [online] Available at:
<<https://developers.braintreepayments.com/guides/payment-method-nonces>>.
- BrainTree, n.d. *Set Up Your PHP Server*. Available at:
<<https://developers.braintreepayments.com/start/hello-server/php>>.
- BrainTree, n.d. *Setting up SDK*. Available at: <<https://developer.paypal.com/docs/accept-payments/express-checkout/ec-braintree-sdk/client-side/android/v2/>>.
- codingdemos, 2016. *Multiple Choice List Dialog*. [online] Available at:
<<https://www.youtube.com/watch?v=wfADRuyul04>>.
- codingdemos, 2017. *Multichoice*. Available at:
<<https://github.com/codingdemos/MultichoiceTutorial/blob/master/app/src/main/java/com/example/multichoicetutorial/MainActivity.java>>.
- Dewald, B.W.A. Ben, 2008. The role of the sommeliers. *International Journal of Wine Business Research*, 20(2), pp.111–123.
- EDMT Dev, 2017a. *Facebook SDK Login*. [online] Available at:
<<https://www.youtube.com/watch?v=KjBNFWKNMOY>>.
- EDMT Dev, 2017b. *Paypal SDK Intergration*. [online] Available at:

<https://www.youtube.com/watch?v=k5lPy_50f0Y&fbclid=IwAR1pC1GFMGd6P2_zD5erokdsZOeBHcykZyVYn9mFJaeUwBQhw752O8lfS6A>.

Facebook, n.d. *CallbackManager*. [online] Available at: <<https://developers.facebook.com/docs/reference/android/current/interface/CallbackManager/>>.

Facebook, n.d. *Facebook App Setup*. [online] Available at: <<https://developers.facebook.com/docs/app-ads/app-setup/>>.

Feng, A., 2015. *HTTP Server*. Available at: <<https://www.codeproject.com/Tips/1040097/Create-simple-http-server-in-Java>>.

food52, n.d. *5 Tastes*. [online] Available at: <<https://food52.com/blog/12326-the-5-tastes-how-to-cook-with-them>>.

Google, n.d. *Gson*. Available at: <<https://github.com/google/gson>>.

guide.codepath, n.d. *ArrayAdapter*. Available at: <<https://guides.codepath.com/android/Using-an-ArrayAdapter-with-ListView>>.

inducesmile, n.d. *Braintree for Payment*. Available at: <<https://inducesmile.com/android/using-braintree-payment-in-android-to-accept-payment/>>.

Jiawei, Han; Micheline, Kamber; Jian, P., 2012. *Data Mining - Concepts and Techniques*. Third Edit ed. [online] Available at: <<http://myweb.sabanciuniv.edu/rdehkharghani/files/2016/02/The-Morgan-Kaufmann-Series-in-Data-Management-Systems-Jiawei-Han-Micheline-Kamber-Jian-Pei-Data-Mining.-Concepts-and-Techniques-3rd-Edition-Morgan-Kaufmann-2011.pdf>>.

Kanojia, R., 2015. *Base Fragment*. Available at: <<https://www.androidinterview.com/android-fragment-tutorial-example/>>.

Leankit, n.d. *What is Kanban?* [online] Available at: <<https://leankit.com/learn/kanban/what-is-kanban/>>.

Leiva, A., n.d. *What is MVP?* [online] Available at: <<https://antonioleiva.com/mvp-android/>>.

Microsoft, n.d. *Data Mining Tutorial*. [online] Available at: <<https://docs.microsoft.com/en-us/sql/analysis-services/data-mining-tutorials-analysis-services?view=sql-server-2014>>.

Microsoft, n.d. *Decision Tree Technical References*. [online] Available at: <<https://docs.microsoft.com/en-us/sql/analysis-services/data-mining/microsoft-decision-trees-algorithm-technical-reference?view=sql-server-2017>>.

Microsoft, n.d. *Microsoft Association Algorithm*. [online] Available at: <<https://docs.microsoft.com/en-us/sql/analysis-services/data-mining/microsoft-association-algorithm-technical-reference?view=sql-server-2017>>.

Microsoft, n.d. *Microsoft Decision Tree*. [online] Available at: <<https://docs.microsoft.com/en-us/sql/analysis-services/data-mining/microsoft-decision-trees-algorithm?view=sql-server-2017>>.

2017>.

modeliosoft, n.d. *UML Tool: Class and Package Diagrams*. [online] Available at: <<https://www.modeliosoft.com/en/resources/diagram-examples/class-and-package-diagrams.html>>.

Pandey, B., 2017. *MVP Demo Project*. Available at: <<https://medium.com/cr8resume/make-you-hand-dirty-with-mvp-model-view-presenter-eab5b5c16e42>>.

Paypal, n.d. *Braintree Developer*. [online] Available at: <<https://developers.braintreepayments.com/guides/paypal/testing-go-live/php>>.

Paypal, n.d. *Sandbox Account*. [online] Available at: <https://developer.paypal.com/docs/classic/lifecycle/sb_create-accounts/#create-a-sandbox-account>.

Paypal, n.d. *Set up server for Braintree SDK*. [online] Available at: <<https://developer.paypal.com/docs/checkout/how-to/braintree-integration/#2-set-up-your-server-to-call-the-braintree-sdk>>.

Paypal, n.d. *Set up your server*. Available at: <<https://developer.paypal.com/docs/accept-payments/express-checkout/ec-braintree-sdk/server-side/java/>>.

phpknowhow, n.d. *Run php Files*. [online] Available at: <<https://www.phpknowhow.com/basics/running-php-files/>>.

Sinhal, A., 2017. *MVP*. [online] Available at: <<https://medium.com/@ankit.sinhal/mvc-mvp-and-mvvm-design-pattern-6e169567bbad>>.

Stack Overflow, 2010. *Facebook Users Permissions*. [online] Available at: <<https://stackoverflow.com/questions/3611682/facebook-graph-api-how-to-get-users-email>>.

Stack Overflow, 2012a. *Border Linear Layout*. Available at: <<https://stackoverflow.com/questions/9211208/how-to-draw-border-on-just-one-side-of-a-linear-layout>>.

Stack Overflow, 2012b. *How do I add a Maven dependency in Eclipse?* [online] Available at: <<https://stackoverflow.com/questions/9164893/how-do-i-add-a-maven-dependency-in-eclipse>>.

Stack Overflow, 2013. *OnActivityResult*. [online] Available at: <<https://stackoverflow.com/questions/20114485/use-onactivityresult-android>>.

Stack Overflow, 2015. *Facebook Login CallbackManager*. [online] Available at: <<https://stackoverflow.com/questions/29557486/facebook-login-callbackmanager-facebookcallback-not-called>>.

Stack Overflow, 2017. *Error Connection Android App to Server*. [online] Available at: <<https://stackoverflow.com/questions/45905805/my-android-app-cant-connect-to-server>>.

VIA University College, 2018. *ICT Project Report Template*. [online] Available at:

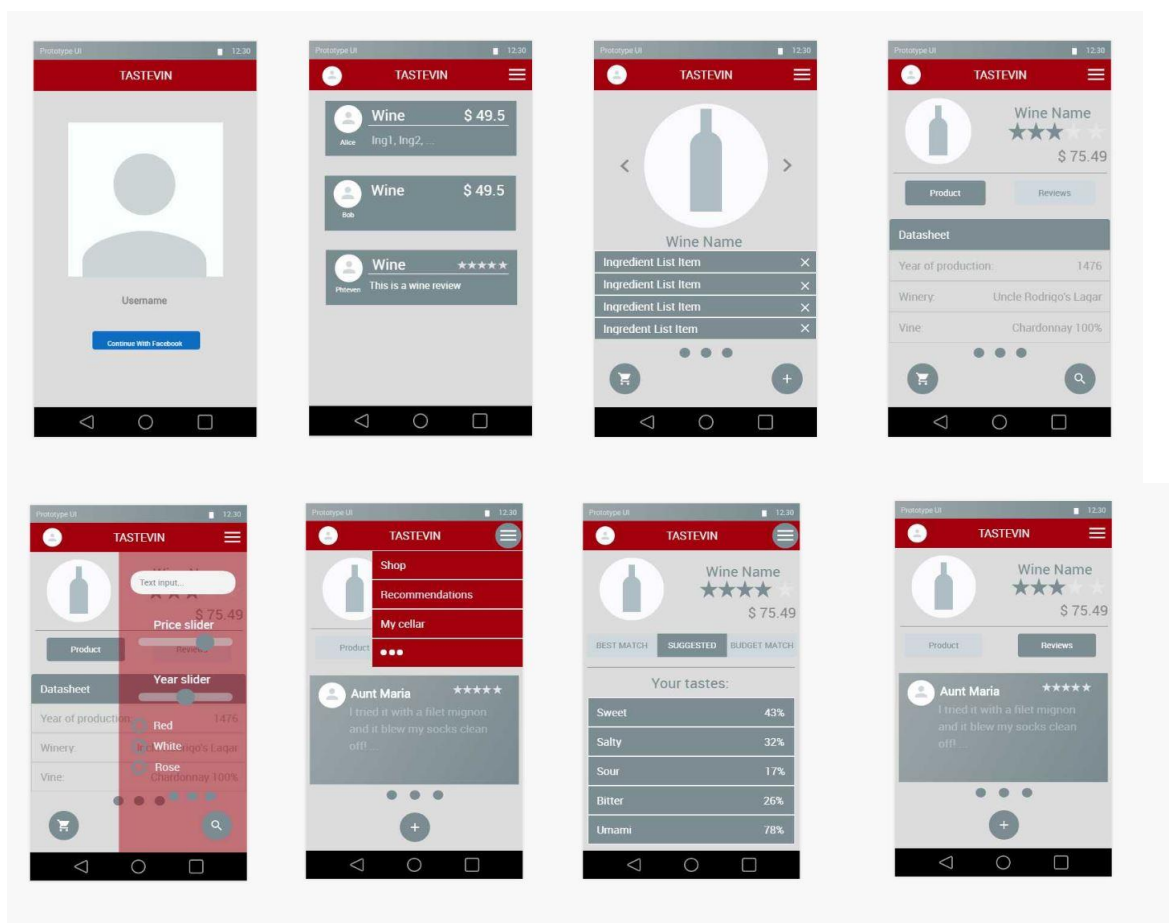


<[https://studienet.via.dk/sites/uddannelse/ict/Horsens/studymaterial/Documents/2018 Semester and Bachelor Project Report Template - VIA ICT Engineering Guidelines.docx](https://studienet.via.dk/sites/uddannelse/ict/Horsens/studymaterial/Documents/2018%20Semester%20and%20Bachelor%20Project%20Report%20Template%20-%20VIA%20ICT%20Engineering%20Guidelines.docx)>.

wineshop, n.d. *Wine Info*. [online] Available at: <<https://www.wineshop.it/en/>>.

Appendices

a) Mockup



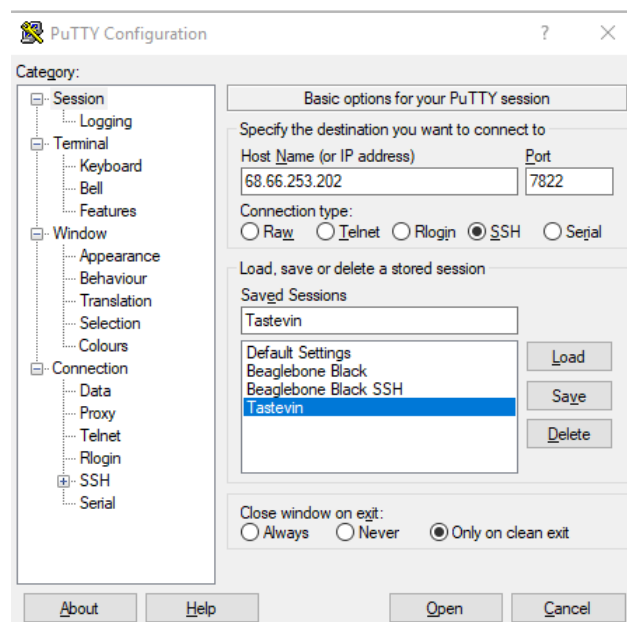
b) User Guide

1. Settings Prototype

For the use of this application there are some settings that needs to be done beforehand.

a. Server

The HTTP server has to be online in order for the app to work. The group has access to a server where an executable java file resides.



It can be accessed through SSH with the IP 68.66.253.202 and the port 7822. Once there, the login should be root and the password are “sunny5566”. After the login enter the command “cd /server” and you will be directed to a folder where the java file resides. Once there enter the command “java -jar Tastevin.jar” to launch the server.

b. Application

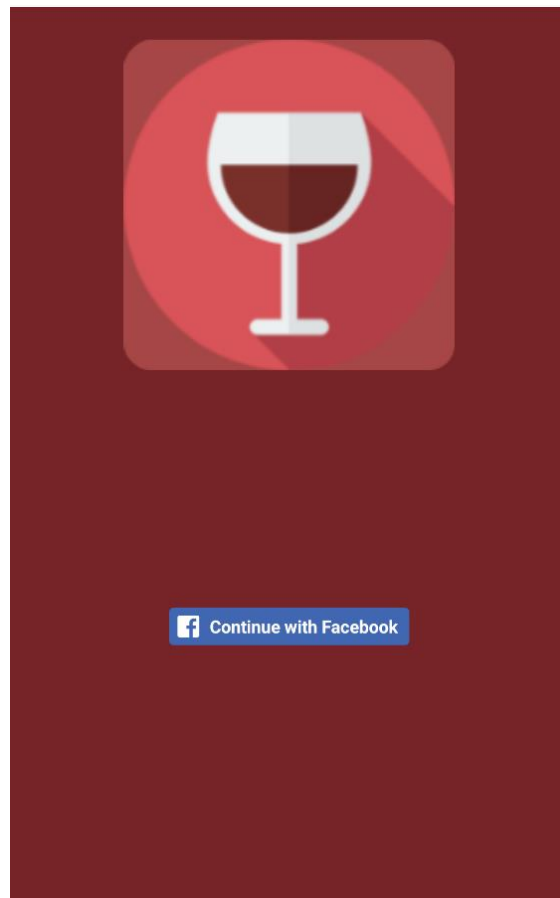
Since the application is uses the Facebook login and it’s still under development, non-authorized users can’t access the login, hence the application. Some tests users were prepared for the tests.

Name	Email	Sweet	Sour	Bitter	Salty	Umami
Bob Rossi	bob_dkplrlw_rossi@tfbnw.net	50	20	20	20	70
Sophia Geller	sophia_brxsrja_geller@tfbnw.net	50	20	50	20	50
Monica Chandler	monica_knoibof_chandler@tfbnw.net	50	50	50	50	50
James Stinson	james_bxrrhhq_stinson@tfbnw.net	70	50	20	20	20

The password to access to their Facebook account is “adminadmin”. The third user has the same tastes as a new registered user.

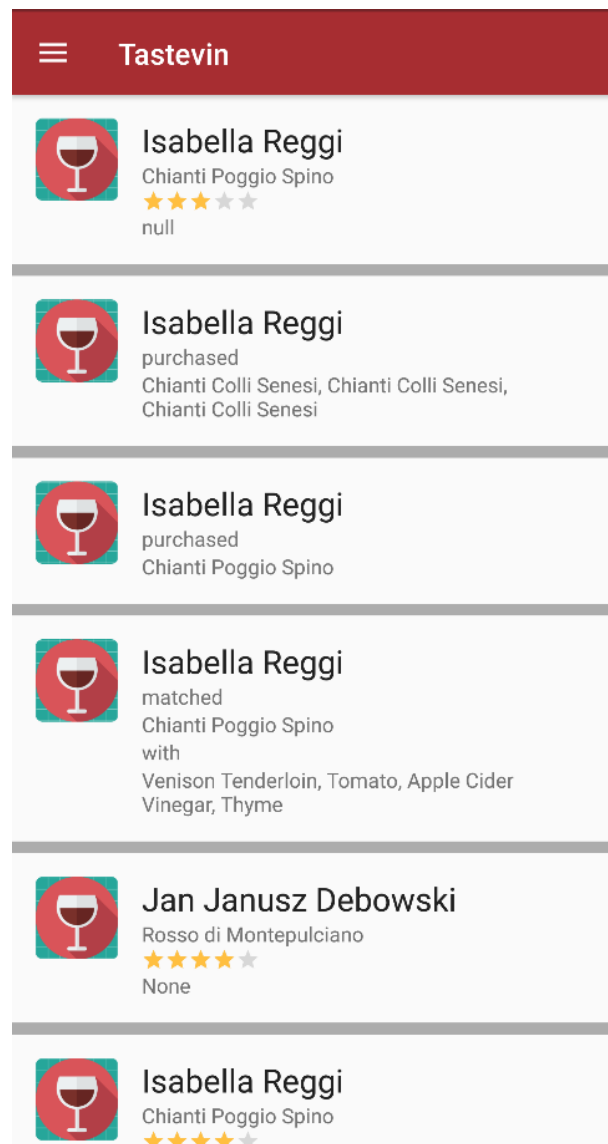
2. Log in

In order to access the Tastevin functionalities, you first have to log in to the system. Upon the first login, the application will automatically register your account.



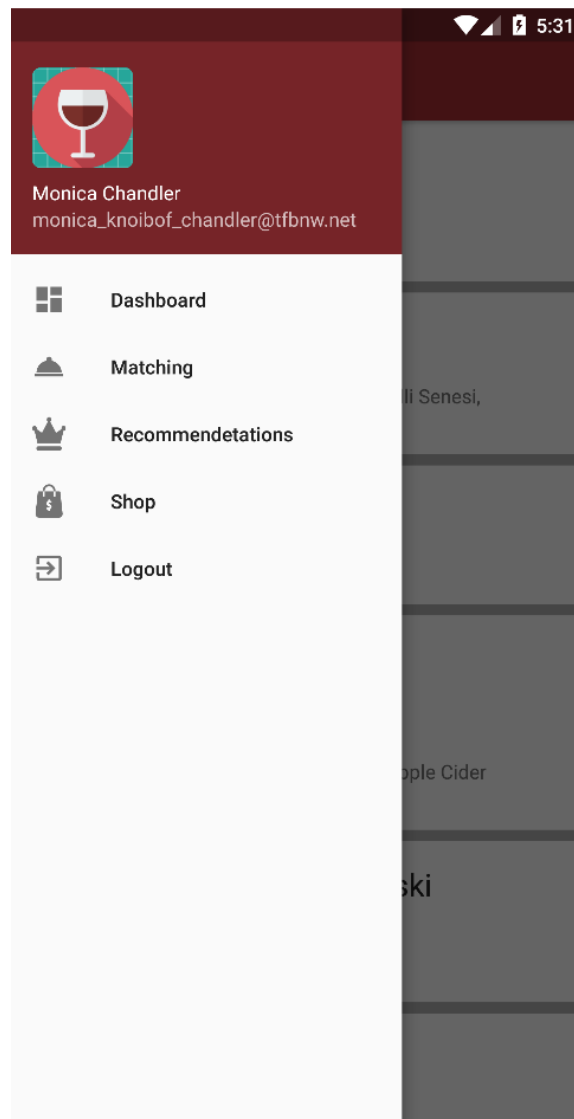
3. Dashboard

The first page displayed is the Dashboard. From here, you can see your friends activities. They come in 3 distinct varieties: reviews, purchases and matchings.



a. Navigation Menu

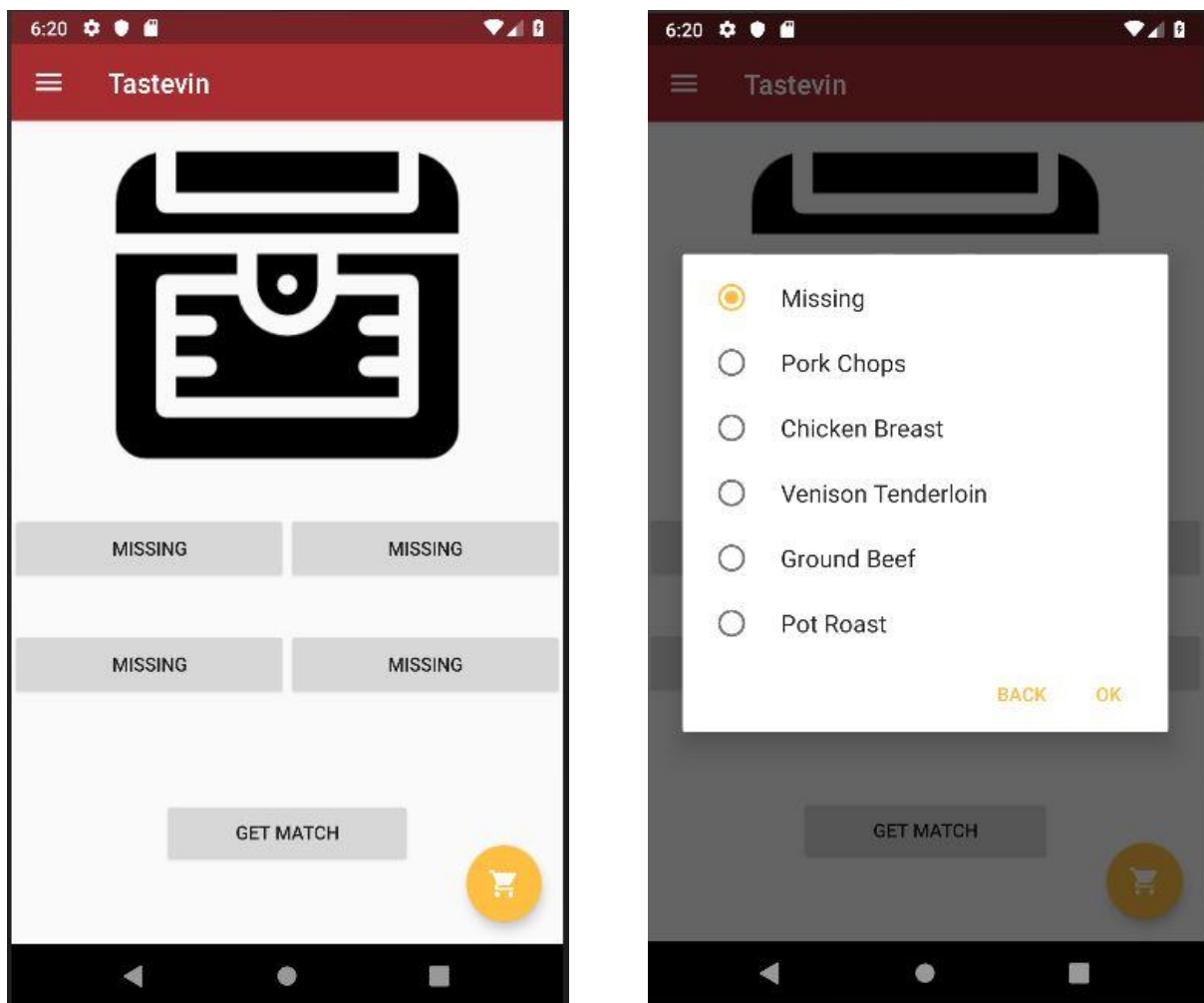
From here, in order to see other features of the app, you need to click the menu button, located in the top left corner.



- *Dashboard* - show the activities of your friends
- *Matching* - here you will be able to match your recipe with a bottle of wine.
- *Recommendations* - display wines *Tastevin* predicts you would like.
- *Shop* - display the available wines in the shop
- *Logout* - exit the application

4. Get a matching

Tastevin's main feature are the matchings. Based on your preferences and history, such as how highly you rate certain wines and which ones you bought, the app will find the best suited wine to your meal.

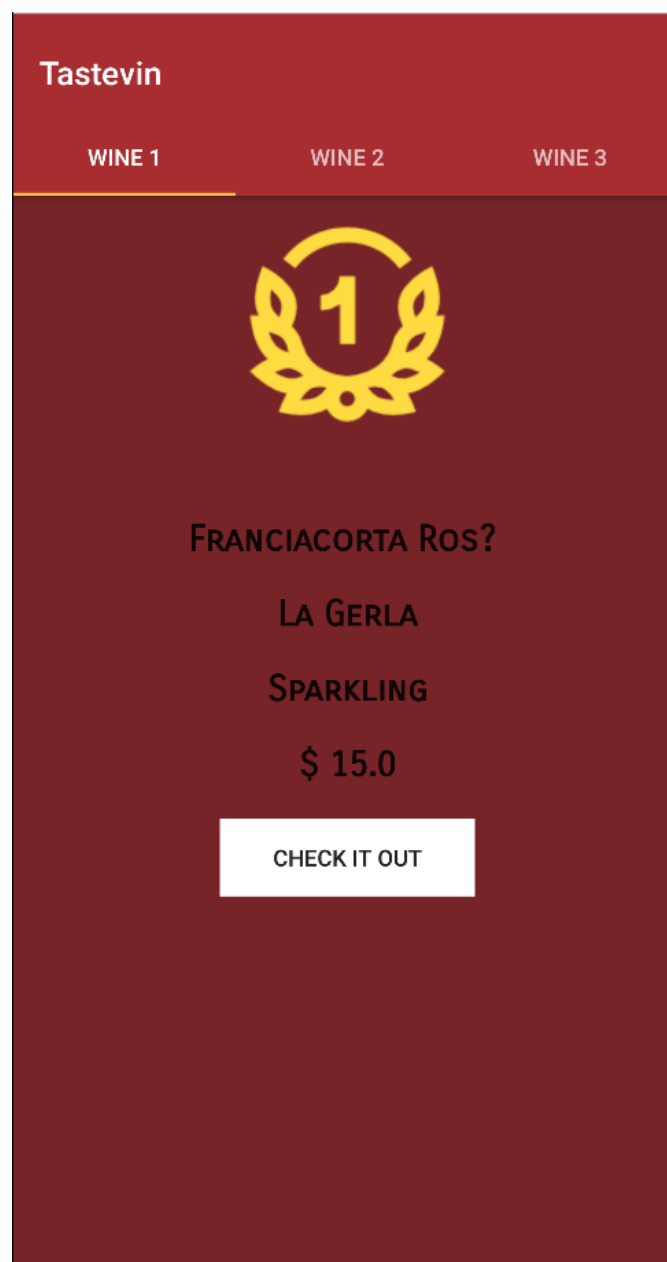


You simply have to choose ingredients you used in your meal, or in case of them not being available on the list, ones resembling them, and press the “GET MATCH” button.

Once a matching is found, you will be directed to a new page where you can check the wines selected.

5. Get a matching

This page shows the 3 selected wines obtained by the matching. They include some information about the wine and a button that will direct you to the page with the full info of the wine.



6. Navigate to Wine Info

Although you can't access this page from the menu, it is very simple to navigate to it. Simply click on any wine you can see in the app, be it on your dashboard, in the shop or from recommendations. Here, you will be able to see all the info available on the currently viewed bottle and add it to your cart.

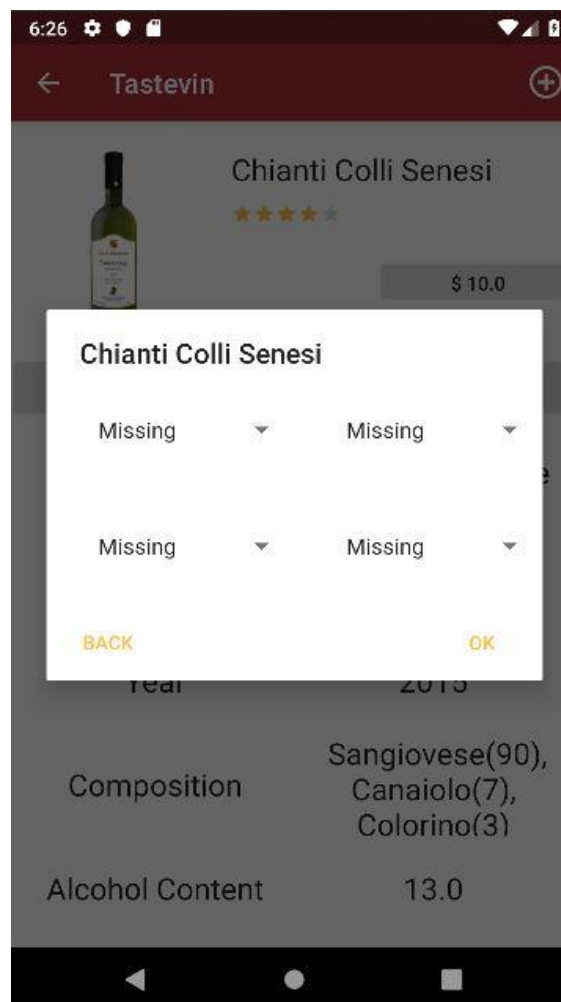


a. Write a review

To add a review, press on the “Review” tab on the right. This will take you to the reviews page, where you can click on the add review button and insert your rating and comment.

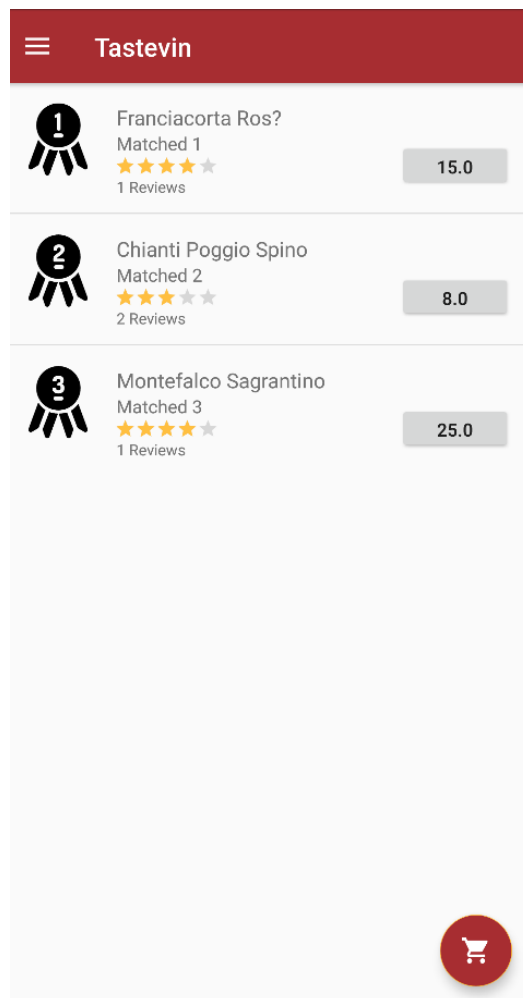
b. Insert a matching

You can also create matchings you know work well! To add a new matching, simply press the “Add” button (+) on the top right corner of the screen and choose ingredients you think go well with the wine you are currently viewing.



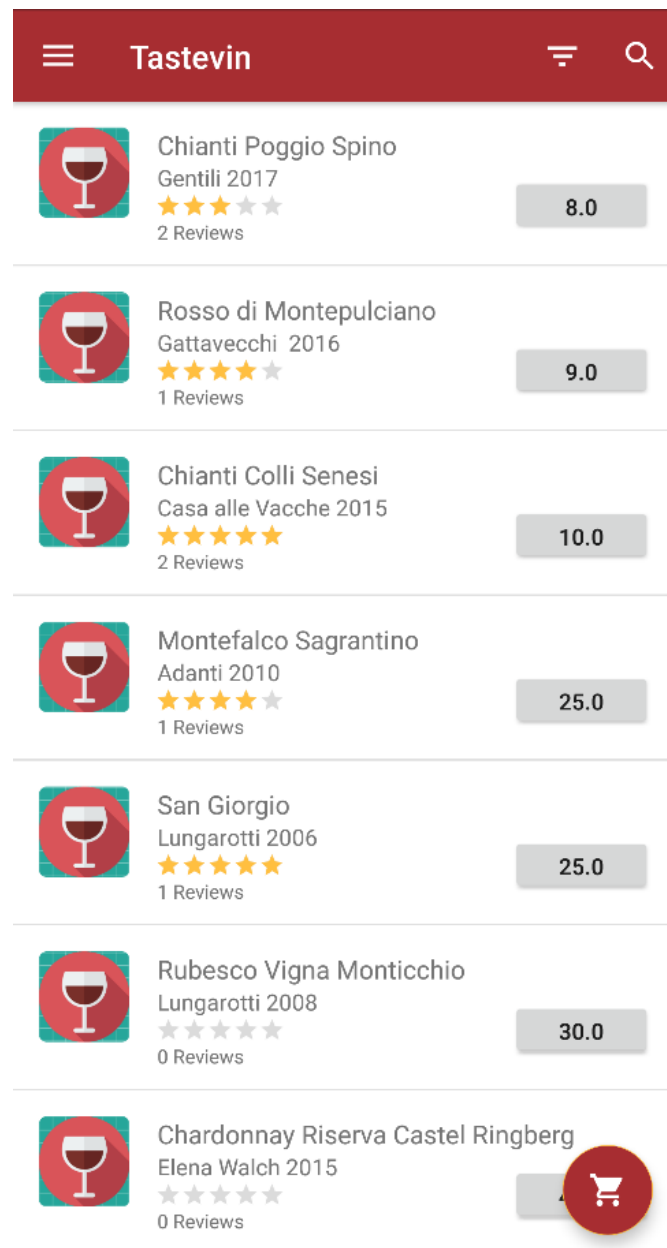
7. Recommendations

Display three wines of our suggestion: Best Match, Suggested (alternative to Best Match) and Budget Match.



8. Shop

As you might suspect, this page allows you to browse wines available for purchase.



a. Search

The app enables you to search for wines available on offer. To search for items, simply press the "Search" lupe button on the right-hand side of the toolbar on top of your screen.

b. Filter

Search results, as well as the shop page, can be filtered based on these criteria;

- Color
- Price
- Year of production
- Rating

6:23

Type

- ☐ Red
- ☐ White
- ☐ Rosé
- ☐ Sparkling
- ☐ Dessert

Price

200 199 1999 2017

☐ 0 - 200 ☐ 2000 - 2018

1 0 2001 1900

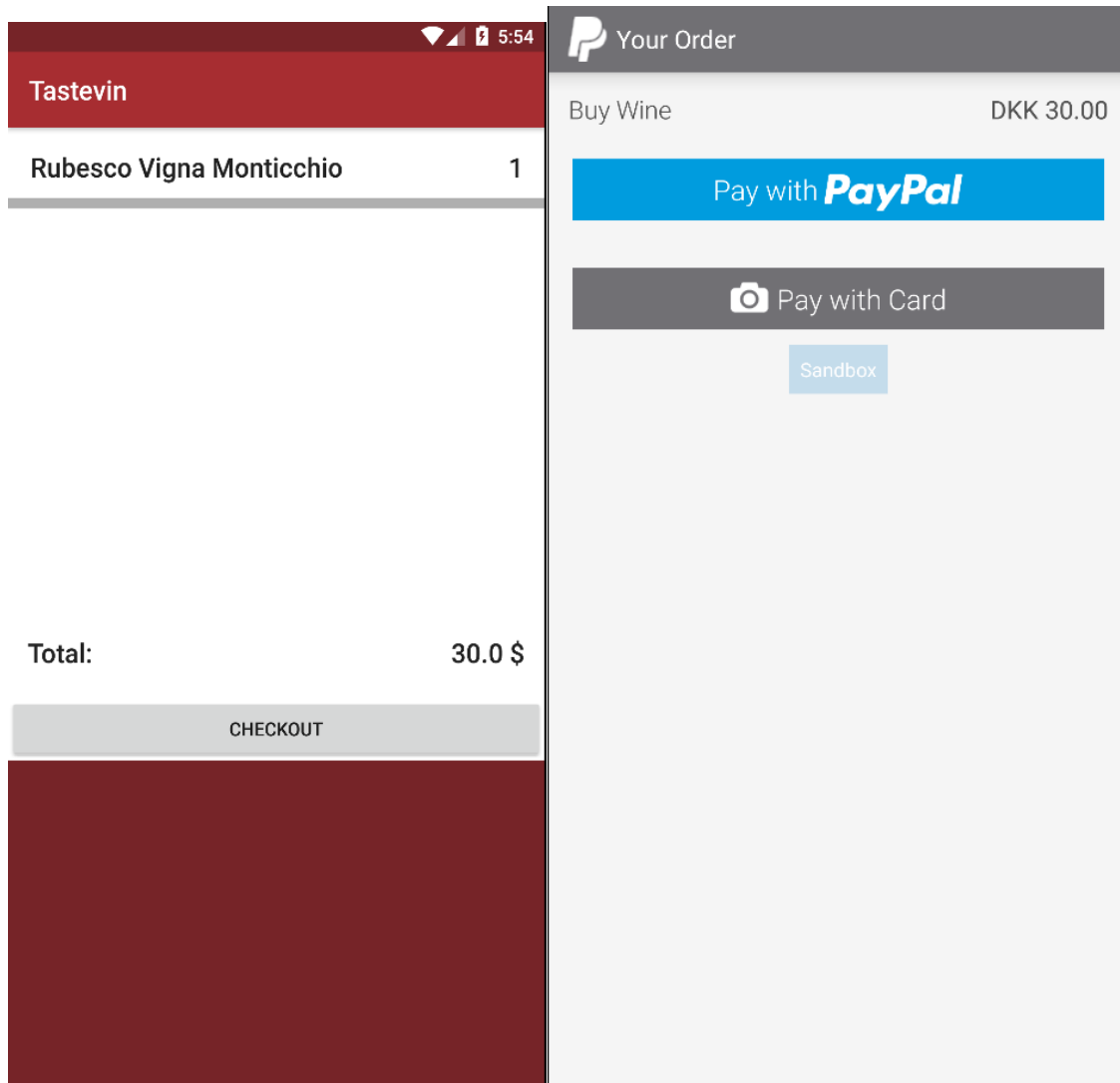
Year

Rating

DISMISS CLEAR ALL OK

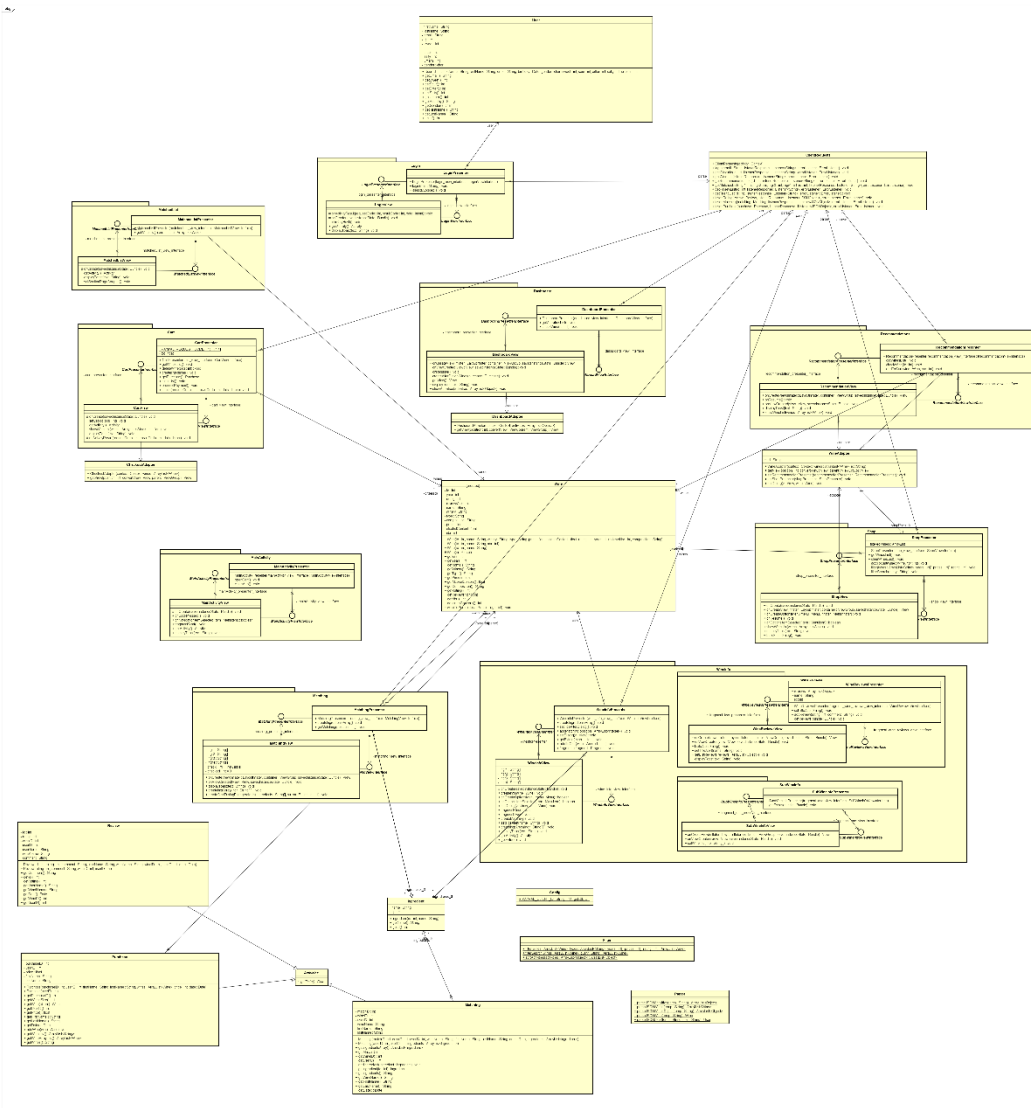
9. Cart

After clicking on the shopping cart button, you will navigate to your cart, from which you can complete the payment using PayPal.



990

d) UML Client Class Diagram



e) Server Code

GET

```
1. package DB;
2.
3.
4. import java.sql.Connection;
5. import java.sql.DriverManager;
```

```

6. import java.sql.ResultSet;
7. import java.sql.Statement;
8. import java.util.ArrayList;
9. import java.util.Arrays;
10.
11. import Util.Ingredient;
12. import Util.Matching;
13. import Util.Purchase;
14. import Util.Review;
15. import Util.User;
16. import Util.Wine;
17.
18. /*
19.  * This file is used for the SELECT of data into the database.
20.  *
21.  * It contains the following methods:
22.  *
23.  *     getUserInfo;
24.  *     getWinesList;
25.  *     getWine;
26.  *     login;
27.  *     getRecommendedWine;
28.  *     getMatchedWine;
29.  *     setValue;
30.  *     getRecommendations
31.  *     getReviewList;
32.  *     getLatestFriendsReviewList;
33.  *     getLatestFriendsPurchasesList;
34.  *     getLatestFriendsMatchingList;
35.  */
36.
37. public class GET {
38.
39.     private static String url = "jdbc:sqlserver://SQL6002.site4now.net;databas
eName=DB_A424FC_IsaRggg;user=DB_A424FC_IsaRggg_admin;password=UvrS9iBUrZ88qZh"
;
40.
41.     /*
42.     * *****
43.     *
44.     *     getUserInfo:
45.     *
46.     *     - Parameters :
47.     *         Integer id                - The id of the user sent by
the client inside the HTTP Get
48.     *
49.     *     - Return type:
50.     *         User user                - The function will return a
User object
51.     *
52.     *     - Body:
53.     *         The function connects to th
e Database and executes the query:
54.     *         SELECT * FROM dbo.[Us
er] WHERE id = id(parameter)
55.     *

```

```

55.      * *****
      * *****
56.      */
57.      public static User getUserInfo(int id)
58.      {
59.          Connection con = null;
60.          Statement stmt = null;
61.          ResultSet rs = null;
62.          User user = null;
63.          try {
64.              Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
65.
66.              con = DriverManager.getConnection(url);
67.              String sql = "SELECT * FROM dbo.[User] WHERE id = "+id;
68.              stmt = con.createStatement();
69.              rs = stmt.executeQuery(sql);
70.              while (rs.next())
71.              {
72.                  user = new User(rs.getInt("id"), rs.getString("firstname"), rs
73.                  .getString("lastname"),
74.                  rs.getString("email"), rs.getDate("birthdate"), rs.get
75.                  String("gender").charAt(0), rs.getInt("sweet"),
76.                  rs.getInt("sour"), rs.getInt("bitter"), rs.getInt("sal
77.                  ty"), rs.getInt("umami"));
78.              }
79.          } catch (Exception e) {
80.              e.printStackTrace();
81.          }
82.          return user;
83.      }
84.      /*
85.      * *****
86.      * *****
87.      *
88.      *      getWineList:
89.      *
90.      *      - Parameters :
91.      *          None
92.      *
93.      *      - Return type:
94.      *          ArrayList<Wine> wines          - The function will return an
95.      *          ArrayList of type Wine
96.      *
97.      *      - Body:
98.      *          The function connects to th
99.      *          e Database and executes the query:
100.      *          SELECT * FROM dbo.Win
101.      *          es
102.      *
103.      *
104.      * *****
105.      * *****
106.      */
107.      public static ArrayList<Wine> getWinesList()
108.      {
109.          Connection con = null;
110.          Statement stmt = null;

```

```

103.         ArrayList<Wine> wines = new ArrayList<>();
104.         ResultSet rs = null;
105.
106.         try {
107.             Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver
108. ");
109.             con = DriverManager.getConnection(url);
110.             String sql = "SELECT * FROM dbo.Wines";
111.             stmt = con.createStatement();
112.             rs = stmt.executeQuery(sql);
113.             while (rs.next())
114.             {
115.                 Wine wine;
116.                 wine = new Wine(rs.getInt("id"), rs.getString("name"),
117. rs.getString("winery"), rs.getString("type"), rs.getFloat("price"),
118. rs.getFloat("alcoholContent"), rs.getInt("rating"), rs.getInt("year"), rs.getInt("reviewsNbr"),
119. rs.getString("composition"));
120.                 wines.add(wine);
121.             }
122.         } catch (Exception e) {
123.             e.printStackTrace();
124.         }
125.         return wines;
126.     }
127.
128.     /*
129.      * *****
130.      *
131.      *      getWine:
132.      *
133.      *      - Parameters :
134.      *          Integer id                - The id of the wine s
135.      *      ent by the client inside the HTTP Get
136.      *
137.      *      - Return type:
138.      *          Wine wine                - The function will re
139.      *      turn a Wine object
140.      *
141.      *      - Body:
142.      *          The function connect
143.      *      s to the Database and executes the query:
144.      *          SELECT * FROM
145.      *      dbo.Wines WHERE id = id(parameter).
146.      *
147.      *      *****
148.      *
149.      *      */
150.     public static Wine getWine(int id)
151.     {
152.         Connection con = null;
153.         Statement stmt = null;
154.         Wine wine = null;
155.         ResultSet rs = null;
156.
157.         try {

```



```

151.         Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver
152.     ");
153.         con = DriverManager.getConnection(url);
154.         String sql = "SELECT * FROM dbo.Wines WHERE id = "+id;
155.         stmt = con.createStatement();
156.         rs = stmt.executeQuery(sql);
157.         while (rs.next())
158.         {
159.             wine = new Wine(rs.getInt("id"), rs.getString("name"),
160.                 rs.getString("winery"), rs.getString("type"), rs.getFloat("price"),
161.                 rs.getFloat("alcoholContent"), rs.getIn
162. t("rating"), rs.getInt("year"), rs.getInt("reviewsNbr"),
163.                 rs.getString("composition"));
164.         }
165.     } catch (Exception e) {
166.         e.printStackTrace();
167.     }
168.     return wine;
169. }
170. /*
171.  * *****
172.  * login:
173.  * - Parameters :
174.  *     String email - The id of the user s
175. ent by the client inside the HTTP Get
176.  * - Return type:
177.  *     User user - The function will re
178. turn a User object
179.  * - Body:
180.  *     The function connect
181. s to the Database and executes the query:
182.  *     SELECT * FROM
183. dbo.[User] WHERE email = email(parameter)
184.  * *****
185.  */
186. public static User login(String mail)
187. {
188.     Connection con = null;
189.     Statement stmt = null;
190.     ResultSet rs = null;
191.     String result = "";
192.     String temp = mail.substring(1, mail.length()-1);
193.     User user = null;
194.     try {
195.         Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver
196.     ");
197.         con = DriverManager.getConnection(url);

```



```

197.         String sql = "SELECT * FROM dbo.[User] WHERE email = '"+tem
    p+"'";
198.         stmt = con.createStatement();
199.         rs = stmt.executeQuery(sql);
200.         while (rs.next())
201.         {
202.             user = new User(rs.getInt("id"), rs.getString("firstNam
e"), rs.getString("lastName"), rs.getString("email"),
203.                 rs.getDate("birthdate"), rs.getString("gender")
                .charAt(0), rs.getInt("sweet"), rs.getInt("sour"),
204.                 rs.getInt("bitter"), rs.getInt("salty"), rs.get
Int("umami"));
205.         }
206.     } catch (Exception e) {
207.         e.printStackTrace();
208.     }
209.     return user;
210. }
211. /*
212.  * *****
213.  *
214.  *     getRecommendedWine:
215.  *
216.  *     - Parameters :
217.  *         Integer id                - The id of the user s
ent by the client inside the HTTP Get
218.  *
219.  *     - Return type:
220.  *         Float[][] probs          - The function will re
turn a Two-dimensional float array
221.  *
222.  *     - Body:                      The function connect
s to the Database and executes the query:
223.  *                                     SELECT Prob1,
Prob2, Prob3, Prob4, Prob5, Prob6, Prob7, Prob8,
224.  *                                     Prob9, Prob10,
Prob11, Prob12, Prob13, Prob14, Prob15 FROM
225.  *                                     dbo.Recommenda
tion WHERE sour = sour(user parameter) AND
226.  *                                     sweet = sweet(
user parameter) AND salty = salty(user parameter)
227.  *                                     AND bitter = b
itter(user parameter) AND umami =
228.  *                                     umami(user par
ameter)
229.  *
230.  * *****
231.  */
232. public static Float[][] getRecommendedWine(int userID)
233. {
234.     Connection con = null;
235.     Statement stmt = null;
236.     ResultSet rs = null;
237.     Float[][] probs = new Float[15][2];
238.     User user = getUserInfo(userID);

```




```

239.
240.         try {
241.             Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver
");
242.
243.             int sour = setValue(user.getSour());
244.             int sweet = setValue(user.getSweet());
245.             int salty = setValue(user.getSalty());
246.             int bitter = setValue(user.getBitter());
247.             int umami = setValue(user.getUmami());
248.             con = DriverManager.getConnection(url);
249.             String sql = "SELECT Prob1, Prob2, Prob3, Prob4, Prob5, Pro
b6, Prob7, Prob8, Prob9, Prob10,"
250.                         + "Prob11, Prob12, Prob13, Prob14, Prob15 FR
OM dbo.Recommendation "
251.                         + "WHERE sour = "+sour+" AND sweet = "+sw
eet+" AND salty = "+salty+" "
252.                         + "AND bitter = "+bitter+" AND umam
i = "+umami+";";
253.             stmt = con.createStatement();
254.             rs = stmt.executeQuery(sql);
255.             while (rs.next())
256.             {
257.                 for(int i = 0; i < 15; i++)
258.                 {
259.                     probs[i][0] = rs.getFloat("Prob"+(i+1));
260.                     probs[i][1] = (float) i+1;
261.                 }
262.             }
263.
264.
265.         } catch (Exception e) {
266.             e.printStackTrace();
267.         }
268.         return probs;
269.     }
270.
271.     /*
272.     * *****
*****
273.     *
274.     *     getMatchedWine:
275.     *
276.     *     - Parameters :
277.     *         Integer ingr1, ingr2           - The id of the user a
nd the chosen ingredients
278.     *         ingr3, ingr4, id             sent by the client i
nside the HTTP Get
279.     *     - Return type:
280.     *         ArrayList<Wine> wines        - The function will re
turn an ArrayList of wines
281.     *
282.     *     - Body:                          The function connect
s to the Database and executes the query:
283.     *                                         SELECT AVG(PRO
B1) AS Prob1, AVG(PROB2) AS Prob2, AVG(PROB3)

```

```

284.      *                               AS Prob3, AVG(
      PROB4) AS Prob4, AVG(PROB5) AS Prob5, AVG(PROB6)
285.      *                               AS Prob6, AVG(
      PROB7) AS Prob7, AVG(PROB8) AS Prob8, AVG(PROB9)
286.      *                               AS Prob9, AVG(
      PROB10) AS Prob10, AVG(PROB11) AS Prob11, AVG(PROB12)
287.      *                               AS Prob12, AVG
      (PROB13) AS Prob13, AVG(PROB14) AS Prob14, AVG(PROB15)
288.      *                               AS Prob15 FRO
      M dbo.MatchProbs WHERE (ingredients)
289.      *
290.      * *****
      *****
291.      */
292.      public static ArrayList<Wine> getMatchedWine(int ingr1, int ingr2,
      int ingr3, int ingr4, int userID)
293.      {
294.          Connection con = null;
295.          Statement stmt = null;
296.          int id = 0;
297.          ResultSet rs = null;
298.          Float[][] probs = new Float[15][2];
299.          ArrayList<Wine> wines = new ArrayList<>();
300.          ArrayList<Integer> ingrID = new ArrayList<>();
301.          ArrayList<String> ingrName = new ArrayList<>();
302.          ingrID.add(ingr1);
303.          ingrID.add(ingr2);
304.          ingrID.add(ingr3);
305.          ingrID.add(ingr4);
306.          User user = getUserInfo(userID);
307.
308.          try {
309.              Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver
      ");
310.
311.              int sour = setValue(user.getSour());
312.              int sweet = setValue(user.getSweet());
313.              int salty = setValue(user.getSalty());
314.              int bitter = setValue(user.getBitter());
315.              int umami = setValue(user.getUmami());
316.              con = DriverManager.getConnection(url);
317.              String sql = "";
318.
319.              for(int k = 0; k < ingrID.size(); k++)
320.              {
321.                  sql = "SELECT name FROM dbo.Ingredient WHERE id = "+ing
      rID.get(k);
322.                  stmt = con.createStatement();
323.                  rs = stmt.executeQuery(sql);
324.                  while (rs.next())
325.                      ingrName.add(rs.getString("name"));
326.              }
327.
328.              sql = "SELECT AVG(PROB1) AS Prob1, AVG(PROB2) AS Prob2, AVG
      (PROB3) AS Prob3, AVG(PROB4) AS Prob4, AVG(PROB5) AS Prob5, "

```

```

329.         +"AVG(PROB6) AS Prob6, AVG(PROB7) AS Prob7, AVG(PRO
    B8) AS Prob8, AVG(PROB9) AS Prob9, AVG(PROB10) AS Prob10, AVG(PROB11) AS Prob1
    1, "
330.         +"AVG(PROB12) AS Prob12, AVG(PROB13) AS Prob13, AVG
    (PROB14) AS Prob14, AVG(PROB15) AS Prob15"
331.         +" FROM dbo.MatchProbs "
332.         + "WHERE ";
333.         int i;
334.
335.         if(ingrID.get(0) != 0)
336.             sql += "Ingredient" + (i+1) + " = '" + ingrName.get(0) + "' ";
337.
338.         for(i = 1; i < ingrID.size()-1; i++)
339.         {
340.             if(ingrID.get(i) != 0)
341.                 sql += "AND Ingredient" + (i+1) + " = '" + ingrName.get(i
    ) + "' ";
342.         }
343.         if(ingrID.get(i) != 0)
344.             sql += "AND Ingredient" + (i+1) + " = '" + ingrName.get(i) + "'
    ";
345.
346.         stmt = con.createStatement();
347.         rs = stmt.executeQuery(sql);
348.
349.         while (rs.next())
350.         {
351.             for(int j = 0; j < 15; j++)
352.             {
353.                 probs[j][0] = rs.getFloat("Prob" + (j+1));
354.                 probs[j][1] = (float) j+1;
355.             }
356.
357.         }
358.         Float[][] recWine = getRecommendedWine(userID);
359.         Float[][] result = new Float[probs.length][2];
360.         for(int k = 0; k < probs.length; k++)
361.         {
362.             result[k][0] = (probs[k][0]) * recWine[k][0];
363.             result[k][1] = probs[k][1];
364.         }
365.         Arrays.sort(result, (a, b) -> Float.compare(b[0], a[0]));
366.         wines.add(getWine(Math.round(result[0][1])));
367.         wines.add(getWine(Math.round(result[1][1])));
368.         wines.add(getWine(Math.round(result[2][1])));
369.
370.     } catch (Exception e) {
371.         e.printStackTrace();
372.     }
373.     return wines;
374. }
375.
376. /*
377.  * *****
378.  *

```

```

379.      *      setValue:
380.      *
381.      *      -   Parameters :
382.      *          Integer i                      - The integer to adapt
383.      *
384.      *      -   Return type:
385.      *          Integer x                      - The function will re
turn an Integer that will be equal
386.      *                                          to 20 or 50 or 70.
387.      *      -   Body:
388.      *
389.      * *****
*****
390.      */
391.
392.      public static int setValue(int i)
393.      {
394.          int x = 0;
395.          if(i < 33 && i > 0)
396.              x = 20;
397.          else if(i < 66 && i >= 33)
398.              x = 50;
399.          else if(i <= 100 && i >= 66)
400.              x = 70;
401.          return x;
402.      }
403.
404.      /*
405.      * *****
*****
406.      *
407.      *      setValue:
408.      *
409.      *      -   Parameters :
410.      *          Integer i                      - The integer to adapt
411.      *
412.      *      -   Return type:
413.      *          Integer x                      - The function will re
turn an Integer that will be equal
414.      *                                          to 20 or 50 or 70.
415.      *      -   Body:
416.      *
417.      * *****
*****
418.      */
419.
420.      public static ArrayList<Wine> getRecommendations(int userID)
421.      {
422.          ArrayList<Wine> wines = new ArrayList<>();
423.          Float[][] recWine = getRecommendedWine(userID);
424.          Arrays.sort(recWine, (a, b) -
> Float.compare(b[0], a[0]));
425.          wines.add(getWine(Math.round(recWine[0][1])));
426.          wines.add(getWine(Math.round(recWine[1][1])));
427.          wines.add(getWine(Math.round(recWine[2][1])));

```

```

428.
429.         return wines;
430.     }
431.
432.     /*
433.     * *****
434.     *
435.     *     getReviewList:
436.     *
437.     *     - Parameters :
438.     *         Integer id                - The id of the wine s
ent by the client inside the HTTP Get
439.     *
440.     *     - Return type:
441.     *         ArrayList<Review> reviews - The function will re
turn an ArrayList of type Review
442.     *
443.     *     - Body:
444.     *         The function connect
s to the Database and executes the query:
445.     *         SELECT * FROM
dbo.Rating_Info WHERE wineID = id(parameter)
446.     * *****
447.     */
448.     public static ArrayList<Review> getReviewList(int id)
449.     {
450.         Connection con = null;
451.         Statement stmt = null;
452.         ArrayList<Review> reviews = new ArrayList<>();
453.         ResultSet rs = null;
454.
455.         try {
456.             Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver
");
457.
458.             con = DriverManager.getConnection(url);
459.             String sql = "SELECT * FROM dbo.Rating_Info WHERE wineID =
"+id;
460.             stmt = con.createStatement();
461.             rs = stmt.executeQuery(sql);
462.             while (rs.next())
463.             {
464.                 Review review = new Review(rs.getInt("id"), rs.getInt("
rating"), rs.getInt("userID"), rs.getString("firstName")+ " "+rs.getString("las
tName"),
465.                 rs.getInt("wineID"), rs.getS
tring("name"), rs.getDate("date"));
466.                 if(rs.getString("text") != null)
467.                     review.setComment(rs.getString("text"));
468.                 else
469.                     review.setComment("None");
470.                 reviews.add(review);
471.             }
472.         } catch (Exception e) {
473.             e.printStackTrace();

```

```

474.         }
475.         return reviews;
476.     }
477.
478.     /*
479.     * *****
480.     *
481.     *     getLatestFriendsReviewList:
482.     *
483.     *     - Parameters :
484.     *         Integer id                - The id of the user s
ent by the client inside the HTTP Get
485.     *
486.     *     - Return type:
487.     *         ArrayList<Review> reviews - The function will re
turn an ArrayList of type Review
488.     *
489.     *     - Body:
s to the Database and executes the query:
490.     *
491.     *         SELECT TOP 5 d
bo.Rating_Info.* FROM dbo.Friends INNER JOIN dbo.Rating_Info ON
492.     *         dbo.Friends.us
erID2 = dbo.Rating_Info.userID WHERE dbo.Friends.userID1 = id(parameter)
493.     *         ORDER BY dbo.R
ating_Info.date DESC
494.     * *****
495.     */
496.     public static ArrayList<Review> getLatestFriendsReviewList(int id)
497.     {
498.         Connection con = null;
499.         Statement stmt = null;
500.         ArrayList<Review> reviews = new ArrayList<>();
501.         ResultSet rs = null;
502.
503.         try {
504.             Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver
");
505.
506.             con = DriverManager.getConnection(url);
507.             String sql = "SELECT TOP 5 dbo.Rating_Info.* FROM dbo.Frien
ds INNER JOIN dbo.Rating_Info ON "
508.                 +"dbo.Friends.userID2 = dbo.Rating_Info.userID WHER
E dbo.Friends.userID1 = "+id
509.                 +" ORDER BY dbo.Rating_Info.date DESC";
510.             stmt = con.createStatement();
511.             rs = stmt.executeQuery(sql);
512.             while (rs.next())
513.             {
514.                 Review review = new Review(rs.getInt("id"), rs.getInt("
rating"), rs.getInt("userID"), rs.getString("firstName")+ " "+rs.getString("las
tName"),
515.                 rs.getInt("wineID"), rs.getS
tring("name"), rs.getDate("date"));

```

```

516.         if(rs.getString("text") != null)
517.             review.setComment(rs.getString("text"));
518.         else
519.             review.setComment("None");
520.         reviews.add(review);
521.     }
522. } catch (Exception e) {
523.     e.printStackTrace();
524. }
525. return reviews;
526. }
527.
528. /*
529.  * *****
530.  *
531.  *     getLatestFriendsPurchasesList:
532.  *
533.  *     - Parameters :
534.  *         Integer id                - The id of the user s
ent by the client inside the HTTP Get
535.  *
536.  *     - Return type:
537.  *         ArrayList<Purchase> purchase - The function will re
turn an ArrayList of type Purchase
538.  *
539.  *     - Body:                        The function connect
s to the Database and executes the query:
540.  *                                     SELECT TOP 5
dbo.Purchases_Info.id, dbo.Purchases_Info.userID, dbo.Purchases_Info.wineID,
541.  *                                     dbo.Purchases_
Info.firstName, dbo.Purchases_Info.lastName, dbo.Purchases_Info.name,
542.  *                                     dbo.Purchases_
Info.date, dbo.Purchases_Info.price FROM dbo.Friends INNER JOIN dbo.Purchases_
Info ON
543.  *                                     dbo.Friends.us
erID2 = dbo.Purchases_Info.userID WHERE dbo.Friends.userID1 = id(parameter)
544.  *                                     ORDER BY dbo.P
urchases_Info.date DESC
545.  *
546.  * *****
547.  */
548.
549. public static ArrayList<Purchase> getLatestFriendsPurchasesList(int
id)
550. {
551.     Connection con = null;
552.     Statement stmt = null;
553.     ArrayList<Purchase> purchases = new ArrayList<>();
554.     ResultSet rs = null;
555.     Purchase purchase = null;
556.
557.     try {
558.         Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver
");
559.

```

```

560.         con = DriverManager.getConnection(url);
561.         String sql = "SELECT TOP 5 dbo.Purchases_Info.id, dbo.Purc
      hases_Info.userID, dbo.Purchases_Info.wineID, "
562.         +"dbo.Purchases_Info.firstName, dbo.Purchases_Info.
      lastName, dbo.Purchases_Info.name, "
563.         +"dbo.Purchases_Info.date, dbo.Purchases_Info.price
      FROM dbo.Friends INNER JOIN dbo.Purchases_Info ON "
564.         +"dbo.Friends.userID2 = dbo.Purchases_Info.userID W
      HERE dbo.Friends.userID1 = "+id
565.         +" ORDER BY dbo.Purchases_Info.date DESC";
566.         stmt = con.createStatement();
567.         rs = stmt.executeQuery(sql);
568.         int i = 0;
569.         while (rs.next())
570.         {
571.             if(i == 0)
572.             {
573.                 i = rs.getInt("id");
574.                 purchase = new Purchase(rs.getInt("id"), rs.getInt(
      "userID"), rs.getString("firstName"),
575.                 rs.getString("lastName"), rs.getDate("date"
      ));
576.                 purchase.addWine(new Wine(rs.getInt("wineID"), rs.g
      etString("name"), rs.getFloat("price")));
577.             }
578.             else if(rs.getInt("id") == i)
579.                 purchase.addWine(new Wine(rs.getInt("wineID"), rs.g
      etString("name"), rs.getFloat("price")));
580.             else if(rs.getInt("id") != i)
581.             {
582.                 purchase.calculatePrice();
583.                 purchases.add(purchase);
584.                 i = rs.getInt("id");
585.                 purchase = new Purchase(rs.getInt("id"), rs.getInt(
      "userID"), rs.getString("firstName"),
586.                 rs.getString("lastName"), rs.getDate("date"
      ));
587.                 purchase.addWine(new Wine(rs.getInt("wineID"), rs.g
      etString("name"), rs.getFloat("price")));
588.             }
589.             purchase.calculatePrice();
590.             purchases.add(purchase);
591.         } catch (Exception e) {
592.             e.printStackTrace();
593.         }
594.         return purchases;
595.     }
596.
597.
598.     /*
599.     * *****
      *****
600.     *
601.     *     getLatestFriendsMatchingList:
602.     *
603.     *     - Parameters :

```



```

604.      *           Integer id           - The id of the user s
ent by the client inside the HTTP Get
605.      *
606.      *   -   Return type:
607.      *           ArrayList<Matching> matching - The function will re
turn an ArrayList of type Matching
608.      *
609.      *   -   Body:           The function connect
s to the Database and executes the query:
610.      *                               SELECT TOP 16
dbo.Matchings.* FROM dbo.Friends INNER JOIN dbo.Matchings ON
611.      *                               dbo.Friends.us
erID2 = dbo.Matchings.userID WHERE dbo.Friends.userID1 = id(parameter)
612.      *                               ORDER BY dbo.M
atchings.date DESC
613.      *
614.      * *****
*****
615.      */
616.
617.      public static ArrayList<Matching> getLatestFriendsMatchingList(int
id)
618.      {
619.          Connection con = null;
620.          Statement stmt = null;
621.          ArrayList<Matching> matchings = new ArrayList<>();
622.          ResultSet rs = null;
623.          Matching matching = null;
624.
625.          try {
626.              Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver
");
627.
628.              con = DriverManager.getConnection(url);
629.              String sql = "SELECT TOP 16 dbo.Matchings.* FROM dbo.Friend
s INNER JOIN dbo.Matchings ON "
630.                  +"dbo.Friends.userID2 = dbo.Matchings.userID WHERE
dbo.Friends.userID1 = "+id+" ORDER BY "
631.                  +"dbo.Matchings.date DESC";
632.              stmt = con.createStatement();
633.              rs = stmt.executeQuery(sql);
634.              int i = 0;
635.
636.              while (rs.next())
637.              {
638.                  if(i == 0)
639.                  {
640.                      i = rs.getInt("matchingID");
641.                      matching = new Matching(rs.getInt("matchingID"), rs
.getInt("wineID"), rs.getInt("userID"), rs.getString("name"),
642.                          rs.getString("firstName"), rs.getString("la
stName"), rs.getDate("date"));
643.                      matching.addIngredient(new Ingredient(rs.getInt("in
gredientID"), rs.getString("ingredientName")));
644.                  }
645.                  else if(rs.getInt("matchingID") == i)

```



```

646.             matching.addIngredient(new Ingredient(rs.getInt("in
        ingredientID"), rs.getString("ingredientName")));
647.             else if(rs.getInt("matchingID") != i)
648.             {
649.                 matchings.add(matching);
650.                 i = rs.getInt("matchingID");
651.                 matching = new Matching(rs.getInt("matchingID"), rs
        .getInt("wineID"), rs.getInt("userID"), rs.getString("name"),
652.                 rs.getString("firstName"), rs.getString("la
        stName"), rs.getDate("date"));
653.                 matching.addIngredient(new Ingredient(rs.getInt("in
        gredientID"), rs.getString("ingredientName")));
654.             }
655.
656.             }
657.             matchings.add(matching);
658.
659.         } catch (Exception e) {
660.             e.printStackTrace();
661.         }
662.         return matchings;
663.     }
664. }

```

INSERT

```

1. package DB;
2.
3.
4. import java.sql.Connection;
5. import java.sql.DriverManager;
6. import java.sql.ResultSet;
7. import java.sql.Statement;
8. import java.util.ArrayList;
9.
10. import Util.Ingredient;
11. import Util.Matching;
12. import Util.Purchase;
13. import Util.Review;
14.
15. /*
16.  * This file is used for the INSERT of data into the database.
17.  *
18.  * It contains the following methods:
19.  *
20.  *     insertRating;
21.  *     insertPurchases;
22.  *     insertMatching;
23.  *
24.  */
25.
26. public class INSERT {
27.

```

```

28.     private static String url = "jdbc:sqlserver://SQL6002.site4now.net;databas
eName=DB_A424FC_IsaRggg;user=DB_A424FC_IsaRggg_admin;password=UvrS9iBUrZ88qZh"
;
29.
30.     /*
31.     * *****
32.     *
33.     *     insertRating:
34.     *
35.     *     - Parameters :
36.     *         Review review      - The object review received from the
HTTP Post of the client
37.     *
38.     *     - Return type:
39.     *         String result      - The function will return a String to
notify the client if it
40.     *                               was successful or not.
41.     *
42.     *     - Body:
43.     *         The function connects to the Databas
e and executes the query:
44.     *                               INSERT INTO dbo.Reviews VALUES
(text, rating, userID, wineID, date).
45.     *                               To see if it was successful, the row
s changed are counted.
46.     * *****
47.     */
48.     public static String insertRating(Review review)
49.     {
50.         Connection con = null;
51.         Statement stmt = null;
52.         int rows = 0;
53.         String result = "";
54.
55.         try {
56.             Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
57.
58.             con = DriverManager.getConnection(url);
59.             String sql = "INSERT INTO dbo.Review VALUES (" + review.getComment()
+ "," + review.getRating() + " , "
60.                 + review.getUserID() + " , " + review.getWineID() + " , GETDATE())"
;
61.             stmt = con.createStatement();
62.             rows = stmt.executeUpdate(sql);
63.             if(rows > 0)
64.                 result = "Success";
65.         } catch (Exception e) {
66.             result = "Failure";
67.         }
68.         return result;
69.     }
70.
71.     /*
72.     * *****

```

```

73.      *
74.      *      insertPurchases:
75.      *      -   Parameters :
76.      *          Purchase purchase - The object review received from the
HTTP post of the Client
77.      *
78.      *      -   Return type:
79.      *          String result      - The function will return a String to
notify the client if it
80.      *                               was successful or not.
81.      *
82.      *      -   Body:                The function connects to the Databas
e and executes the query:
83.      *                               INSERT INTO dbo.Purchases VALUES
(userID, date).
84.      *                               After this , it asks for the the ID
that was given to this
85.      *                               new row and executes a new query to
add the elements of the purchase:
86.      *                               INSERT INTO dbo.Purchases_Mappin
g VALUES (wineID, purchaseID, wineNbr).
87.      *                               To see if it was successful, the row
s changed are counted.
88.      *
89.      * *****
*****
90.      */
91.      public static String insertPurchases(Purchase purchase, ArrayList<Integer>
wines)
92.      {
93.          Connection con = null;
94.          Statement stmt = null;
95.          ResultSet rs = null;
96.          int id = 0;
97.          int rows = 0;
98.          String result = "";
99.
100.         try {
101.             Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver
");
102.
103.             con = DriverManager.getConnection(url);
104.             String sql = "INSERT INTO dbo.Purchases VALUES("+purchase.g
etUserID()+", GETDATE());";
105.             stmt = con.createStatement();
106.             rows = stmt.executeUpdate(sql);
107.             if(rows == 0)
108.                 return "Failure";
109.             rows = 0;
110.             sql = "SELECT TOP 1 id FROM dbo.Purchases WHERE userID = "+
purchase.getUserID()+" ORDER BY date DESC";
111.             stmt = con.createStatement();
112.             rs = stmt.executeQuery(sql);
113.             while (rs.next())
114.                 id = rs.getInt("id");
115.             sql = "";
116.             for(int i = 0; i < wines.size(); i++)

```

```

117.         sql += "INSERT INTO dbo.Purchases_Mapping VALUES(+" + win
es.get(i) + ", " + id + ", " + (i+1) + ");\n";
118.         rows = stmt.executeUpdate(sql);
119.         if(rows > 0)
120.             result = "Success";
121.     } catch (Exception e) {
122.         return "Failure";
123.     }
124.     return result;
125. }
126.
127.     /*
128.     * *****
129.     *
130.     *     insertMatching:
131.     *
132.     *     - Parameters :
133.     *         Matching matching - The object matching received
from the HTTP Post of the client
134.     *
135.     *     - Return type:
136.     *         String result - The function will return a St
ring to notify the client if it
137.     *                             was successful or not.
138.     *
139.     *     - Body:
140.     *         The function connects to the
Database and executes the query:
INSERT INTO dbo.Matchin
g VALUES (wineID, userID, date).
141.     *
142.     *         After this , it asks for the
the ID that was given to this
143.     *
144.     *         new row and executes a new qu
ery to add the elements of the purchase:
INSERT INTO dbo.Mapping V
ALUES (ingredientID, matchingID, ingredientNbr).
145.     *
146.     *         To see if it was successful,
the rows changed are counted.
147.     *
148.     * *****
149.     *
150.     */
151.     public static String insertMatching(Matching matching, ArrayList<In
gredient> ingr)
152.     {
153.         Connection con = null;
154.         Statement stmt = null;
155.         ResultSet rs = null;
156.         int id = 0;
157.         int rows = 0;
158.         String result = "";
159.
160.         try {
161.             Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver
");
162.             con = DriverManager.getConnection(url);

```

```

161.         String sql = "INSERT INTO dbo.Matching VALUES (" + matching.g
            etWineID() + ", " + matching.getUserID() + ", GETDATE());";
162.         stmt = con.createStatement();
163.         rows = stmt.executeUpdate(sql);
164.         if (rows == 0)
165.             return "Failure";
166.         rows = 0;
167.         sql = "SELECT TOP 1 id FROM dbo.Matching WHERE userID = " + m
            atching.getUserID() + " ORDER BY date DESC";
168.         stmt = con.createStatement();
169.         rs = stmt.executeQuery(sql);
170.         while (rs.next())
171.             id = rs.getInt("id");
172.         sql = "";
173.         for (int i = 0; i < ingr.size(); i++)
174.             sql += "INSERT INTO dbo.Matching_Mapping VALUES (" + ingr
                .get(i).getId() + ", " + id + ", " + (i + 1) + ");\n";
175.         rows = stmt.executeUpdate(sql);
176.         if (rows > 0)
177.             result = "Success";
178.         } catch (Exception e) {
179.             return "Failure";
180.         }
181.         return result;
182.     }
183.
184. }

```

Ingredient

```

1. package Util;
2.
3. /*
4.  *
5.  * Class for the Ingredient
6.  *
7.  */
8.
9. public class Ingredient {
10.
11.     private String name;
12.     private int id;
13.
14.     /*
15.     * Constructor
16.     */
17.
18.     public Ingredient(int id, String name) {
19.         this.name = name;
20.         this.id = id;
21.     }
22.     /*
23.     * Getters
24.     */
25.     public String getName() {

```

```

26.         return name;
27.     }
28.
29.     public int getId() {
30.         return id;
31.     }
32.
33. }

```

Matching

```

1. package Util;
2.
3.
4. import java.util.ArrayList;
5. import java.util.Date;
6. /*
7.  *
8.  * Class for the Matching
9.  *
10. */
11. public class Matching{
12.
13.     private int matchID, wineID, userID;
14.     private ArrayList<Ingredient> ingredients;
15.     private Ingredient[] ingredientsArray;
16.     private String wineName, firstName, lastName;
17.     private Date date;
18. /*
19.  * Constructor
20.  */
21.     public Matching(int matchID, int wineID, int userID, String wineName,
22.         String firstName, String lastName, Date date) {
23.         this.matchID = matchID;
24.         this.wineID = wineID;
25.         this.userID = userID;
26.         this.ingredients = new ArrayList<>();
27.         this.wineName = wineName;
28.         this.firstName = firstName;
29.         this.lastName = lastName;
30.         this.date = date;
31.     }
32. /*
33.  * Constructor with less info
34.  */
35.     public Matching(int wineID, int userID, Date date) {
36.         this.wineID = wineID;
37.         this.userID = userID;
38.         this.ingredients = new ArrayList<>();
39.         this.date = date;
40.     }
41. /*
42.  * Constructor with less info and ingredients
43.  */
44.     public Matching(int wineID, int userID, Ingredient[] ingredientsArray)

```



```

45.     {
46.         this.wineID = wineID;
47.         this.userID = userID;
48.         this.ingredientsArray = ingredientsArray;
49.     }
50. /*
51.  * Getters
52.  */
53.     public ArrayList<Ingredient> getIngredientsArray()
54.     {
55.         return ingredients;
56.     }
57.
58.     public int getMatchID() {
59.         return matchID;
60.     }
61.     public int getWineID() {
62.         return wineID;
63.     }
64.     public int getUserID() {
65.         return userID;
66.     }
67.
68.     public Ingredient getIngredient(int id)
69.     {
70.         return ingredients.get(id);
71.     }
72.
73.     public String getWineName() {
74.         return wineName;
75.     }
76.     public String getFirstName() {
77.         return firstName;
78.     }
79.     public String getLastName() {
80.         return lastName;
81.     }
82.     public Date getDate()
83.     {
84.         return date;
85.     }
86. /*
87.  * Setters
88.  */
89.     public void addIngredient(Ingredient ingredient)
90.     {
91.         ingredients.add(ingredient);
92.     }
93.
94. }

```

Purchase

```

1. package Util;
2.

```




```

3.
4. import java.util.ArrayList;
5. import java.util.Date;
6. /*
7.  *
8.  * Class for the Purchases
9.  *
10. */
11. public class Purchase{
12.
13.     private int purchaseID, userID;
14.     private float price;
15.     private ArrayList<Wine> wines;
16.     private String firstName, lastName;
17.     private Date date;
18. /*
19.  * Constructor
20.  */
21.     public Purchase(int purchaseID, int userID, String firstName, String lastN
ame, Date date)
22.     {
23.         this.purchaseID = purchaseID;
24.         this.userID = userID;
25.         wines = new ArrayList<>();
26.         this.firstName = firstName;
27.         this.lastName = lastName;
28.         this.date = date;
29.         price = 0;
30.     }
31. /*
32.  * Getters
33.  */
34.     public int getPurchaseID()
35.     {
36.         return purchaseID;
37.     }
38.
39.     public ArrayList<Wine> getWines()
40.     {
41.         return wines;
42.     }
43.
44.     public int getWinesSize()
45.     {
46.         return wines.size();
47.     }
48.     public Wine getWine(int id)
49.     {
50.         return wines.get(id);
51.     }
52.
53.     public int getUserID() {
54.         return userID;
55.     }
56.
57.     public float getPrice()
58.     {

```



```

59.         return price;
60.     }
61.
62.     public String getFirstName() {
63.         return firstName;
64.     }
65.
66.     public String getLastName() {
67.         return lastName;
68.     }
69.
70.     public Date getDate()
71.     {
72.         return date;
73.     }
74. /*
75.  * Setters
76.  */
77.     public void addWine(Wine wine)
78.     {
79.         wines.add(wine);
80.     }
81. /*
82.  * Calculate Price
83.  */
84.     public void calculatePrice()
85.     {
86.         for(int i = 0; i < wines.size(); i++)
87.             price += wines.get(i).getPrice();
88.     }
89. }

```

Review

```

1. package Util;
2.
3.
4. import java.sql.Date;
5. /*
6.  * Class for the Review
7.  */
8. public class Review{
9.
10.     private int id, rating, userID, wineID;
11.     private String comment, userName, wineName;
12.     private Date date;
13. /*
14.  * Constructor
15.  */
16.     public Review(int id, int rating, int userID, String userName, int wineID,
17.         String wineName, Date date)
18.     {
19.         this.id = id;

```



```

19.         this.rating = rating;
20.         this.userName = userName;
21.         this.userID = userID;
22.         this.wineName = wineName;
23.         this.wineID = wineID;
24.         this.date = date;
25.     }
26. /*
27.  * Getters
28.  */
29.     public String getComment()
30.     {
31.         return comment;
32.     }
33.
34.     public int getId()
35.     {
36.         return id;
37.     }
38.
39.     public int getRating()
40.     {
41.         return rating;
42.     }
43.
44.     public int getUserID()
45.     {
46.         return userID;
47.     }
48.
49.     public int getWineID()
50.     {
51.         return wineID;
52.     }
53.
54.     public Date getDate()
55.     {
56.         return date;
57.     }
58.
59.     public String.getUserName()
60.     {
61.         return userName;
62.     }
63.
64.     public String getWineName()
65.     {
66.         return wineName;
67.     }
68. /*
69.  * Setters
70.  */
71.     public void setComment(String comment)
72.     {
73.         this.comment = comment;
74.     }
75.

```



```
76.
77. }
```

User

```
1. package Util;
2.
3.
4. import java.util.Date;
5. /*
6.  * Class for the User
7.  */
8. public class User {
9.
10.     private String firstName, lastName, email;
11.     private int id, sweet, sour, bitter, salty, umami;
12.     private Date birthday;
13.     private char gender;
14. /*
15.  * Constructor
16.  */
17.     public User(int id, String firstName, String lastName, String email, Date
        birthday,
18.                 char gender, int sweet, int sour, int bitter, int salty, int umami
19.     )
20.     {
21.         this.id = id;
22.         this.firstName = firstName;
23.         this.lastName = lastName;
24.         this.email = email;
25.         this.birthday = birthday;
26.         this.gender = gender;
27.         this.sweet = sweet;
28.         this.sour = sour;
29.         this.bitter = bitter;
30.         this.salty = salty;
31.         this.umami = umami;
32.     }
33. /*
34.  * Getters
35.  */
36.     public String getEmail()
37.     {
38.         return email;
39.     }
40.     public int getSweet()
41.     {
42.         return sweet;
43.     }
44.     public int getSour()
45.     {
46.         return sour;
47.     }
48. }
```



```

49.
50.     public int getBitter()
51.     {
52.         return bitter;
53.     }
54.
55.     public int getSalty()
56.     {
57.         return salty;
58.     }
59.
60.     public int getUmami()
61.     {
62.         return umami;
63.     }
64.
65.     public String getBirthday()
66.     {
67.         return birthday.toString();
68.     }
69.
70.     public char getGender()
71.     {
72.         return gender;
73.     }
74.
75.     public String getFirstName()
76.     {
77.
78.         return firstName;
79.     }
80.
81.     public String getLastName()
82.     {
83.         return lastName;
84.     }
85.
86.     public int getId()
87.     {
88.         return id;
89.     }
90. }

```

Wine

```

1. package Util;
2.
3. /*
4.  * Class for the Wine
5.  */
6. public class Wine {
7.
8.     private int id, year, rating, reviewsNbr;

```

```

9.     private String name, winery, type, composition;
10.    private float price, alcoholContent;
11.    /*
12.     * Constructor
13.     */
14.    public Wine(int id, String name, String winery, String type, float price,
15.               float alcoholContent, int rating, int year, int reviewsNbr, String
16.               composition)
17.    {
18.        this.id = id;
19.        this.name = name;
20.        this.winery = winery;
21.        this.type = type;
22.        this.price = price;
23.        this.alcoholContent = alcoholContent;
24.        this.composition = composition;
25.        this.rating = rating;
26.        this.year = year;
27.        this.reviewsNbr = reviewsNbr;
28.    }
29.    /*
30.     * Constructor with less info
31.     */
32.    public Wine(int id, String name, float price)
33.    {
34.        this.id = id;
35.        this.name = name;
36.        this.price = price;
37.    }
38.    /*
39.     * Getters
40.     */
41.    public int getId()
42.    {
43.        return id;
44.    }
45.    public int getYear()
46.    {
47.        return year;
48.    }
49.    public String getName()
50.    {
51.        return name;
52.    }
53.    public String getWinery()
54.    {
55.        return winery;
56.    }
57.    public String getType()
58.    {
59.        return type;
60.    }
61.    }
62.

```

```

64.
65.     public float getPrice()
66.     {
67.         return price;
68.     }
69.
70.     public float getAlcoholContent()
71.     {
72.         return alcoholContent;
73.     }
74.
75.     public String getComposition()
76.     {
77.         return composition;
78.     }
79.
80.     public int getRating()
81.     {
82.         return rating;
83.     }
84.
85.     public int getReviewsNbr() {
86.         return reviewsNbr;
87.     }
88. }

```

HTTP

```

1.  package server;
2.
3.
4.  import java.io.BufferedReader;
5.  import java.io.IOException;
6.  import java.io.InputStreamReader;
7.  import java.io.OutputStream;
8.  import java.util.List;
9.  import java.util.Map;
10.
11. import org.json.JSONException;
12.
13. import com.sun.net.httpserver.HttpExchange;
14. import com.sun.net.httpserver.HttpHandler;
15. /*
16.  * Class for the handling of the HTTP Requests
17.  */
18. public class HTTP implements HttpHandler {
19.
20.     ServerModel model = new ServerModel();
21.
22.     public void handle(HttpExchange con) throws IOException
23.     {
24.         Map<String,List<String>> header = con.getRequestHeaders();
25.         String method = con.getRequestMethod();
26.         InputStreamReader input = new InputStreamReader(con.getRequestBody(),
"utf-8");

```



```

27.         BufferedReader buffer = new BufferedReader(input);
28.         String body = buffer.readLine();
29.
30.         String response;
31.         try {
32.             response = model.checkAction(method, header, body);
33.             if(con.getRequestMethod().equals("POST"))
34.                 con.getResponseHeaders().set("Content-
Type", "application/json");
35.             con.sendResponseHeaders(200, response.length());
36.             OutputStream output = con.getResponseBody();
37.             output.write(response.toString().getBytes());
38.             output.close();
39.         } catch (JSONException e) {
40.             e.printStackTrace();
41.         }
42.
43.
44.     }
45. }

```

ServerModel

```

1. package server;
2.
3.
4. import java.util.ArrayList;
5. import java.util.Arrays;
6. import java.util.List;
7. import java.util.Map;
8.
9. import org.json.JSONArray;
10. import org.json.JSONException;
11. import org.json.JSONObject;
12.
13. import com.google.gson.Gson;
14. import com.google.gson.JsonElement;
15. import com.google.gson.JsonParser;
16.
17. import DB.GET;
18. import DB.INSERT;
19. import Util.User;
20. import Util.Wine;
21. import Util.Ingredient;
22. import Util.Matching;
23. import Util.Purchase;
24. import Util.Review;
25. /*
26.  * Model of the server
27.  */
28. public class ServerModel {
29. /*
30.  * checkMethod - GET or POST

```




```
31. */
32. public String checkAction(String method, Map<String,List<String>> header,
    String body) throws JSONException
33. {
34.     if(method.equals("POST"))
35.         return checkActionPOST(header, body);
36.     else if(method.equals("GET"))
37.         return checkActionGET(header, body);
38.     else
39.         return "ERROR METHOD";
40. }
41. /*
42. * Check which GET requests is, elaborate and give response
43. */
44. public String checkActionGET(Map<String,List<String>> header, String body)
    throws JSONException
45. {
46.     String response = "";
47.     String i;
48.     Wine wine;
49.     JSONObject object;
50.     User user;
51.     ArrayList<Wine> wines;
52.     ArrayList<Review> reviews;
53.     ArrayList<Purchase> purchases;
54.     ArrayList<Matching> matchings;
55.
56.     switch (header.get("REQUEST").get(0).toString())
57.     {
58.
59.     case "WINEINFO":
60.         i = header.get("NBR").get(0).toString();
61.         wine = GET.wine(Integer.parseInt(i.toString()));
62.         object = new JSONObject(wine);
63.         response += object.toString();
64.         break;
65.
66.     case "RECOMMENDATIONLIST":
67.         i = header.get("NBR").get(0).toString();
68.         wines = GET.getRecommendations(Integer.parseInt(i.toString()));
69.         object = new JSONObject();
70.         object.append("wines", wines);
71.         response += object.toString();
72.         break;
73.
74.     case "MATCHEDLIST":
75.         int i0 = Integer.parseInt(header.get("NBR1").get(0).toString());
76.         int i1 = Integer.parseInt(header.get("NBR2").get(0).toString());
77.         int i2 = Integer.parseInt(header.get("NBR3").get(0).toString());
78.         int i3 = Integer.parseInt(header.get("NBR4").get(0).toString());
79.         int i4 = Integer.parseInt(header.get("NBR5").get(0).toString());
80.         wines = GET.getMatchedWine(i0, i1, i2, i3, i4);
81.         object = new JSONObject();
82.         object.append("wines", wines);
83.         response += object.toString();
84.         break;
85.     }
```

```

86.         case "WINELIST":
87.             wines = GET.getWinesList();
88.             object = new JSONObject();
89.             object.append("wines", wines);
90.             response += object.toString();
91.             break;
92.
93.         case "REVIEWLIST":
94.             i = header.get("NBR").get(0).toString();
95.             reviews = GET.getReviewList(Integer.parseInt(i.toString()));
96.             object = new JSONObject();
97.             object.append("reviews", reviews);
98.             response += object.toString();
99.             break;
100.
101.         case "LISTFRIENDS":
102.             i = header.get("NBR").get(0).toString();
103.             reviews = GET.getLatestFriendsReviewList(Integer.parseInt(i
104. toString()));
105.             purchases = GET.getLatestFriendsPurchasesList(Integer.parse
106. Int(i.toString()));
107.             matchings = GET.getLatestFriendsMatchingList(Integer.parseI
108. nt(i.toString()));
109.             object = new JSONObject();
110.             object.append("reviews", reviews);
111.             object.append("purchases", purchases);
112.             object.append("matching", matchings);
113.             response += object.toString();
114.             break;
115.
116.         case "LOGIN":
117.             i = header.get("EMAIL").toString();
118.             user = GET.login(i);
119.             object = new JSONObject(user);
120.             response += object.toString();
121.             break;
122.
123.         default:
124.             response = "REQUEST NOT RECONGNIZED";
125.             break;
126.     }
127.     return response;
128. }
129. /*
130.  * Check which POST requests is, elaborate and give response
131.  */
132. public String checkActionPOST(Map<String,List<String>> header, Stri
133. ng body) throws JSONException
134. {
135.     String response = "";
136.     Object i;
137.     Gson gson = new Gson();
138.     JsonParser parser = new JsonParser();
139.     JsonElement object;
140.     Review review = null;
141.     String result = "";
142.     JSONObject jsonObject, jsonObject2;

```

```

139.         JSONArray array1;
140.
141.         switch (header.get("REQUEST").get(0).toString())
142.         {
143.
144.             case "INSERTRATING":
145.                 i = body;
146.                 object = parser.parse((String) i);
147.                 review = gson.fromJson(object, Review.class);
148.                 jsonObject = new JSONObject();
149.                 result = INSERT.insertRating(review);
150.                 response += jsonObject.append("result", result);
151.                 break;
152.
153.             case "INSERTPURCHASING":
154.                 i = body;
155.                 jsonObject2 = new JSONObject(body);
156.                 array1 = jsonObject2.getJSONArray("winesArray");
157.
158.                 ArrayList<Integer> wines = new ArrayList<>();
159.                 for(int j = 0; j < array1.length(); j++)
160.                     wines.add(array1.getJSONObject(j).getInt("id"));
161.                 Purchase purchase = gson.fromJson(body, Purchase.class);
162.                 jsonObject = new JSONObject();
163.                 result = INSERT.insertPurchases(purchase, wines);
164.                 response += jsonObject.append("result", result);
165.                 break;
166.
167.             case "INSERTMATCHING":
168.                 i = body;
169.                 jsonObject2 = new JSONObject(body);
170.                 array1 = jsonObject2.getJSONArray("ingredientsArray");
171.                 ArrayList<Ingredient> ingr = new ArrayList<>();
172.                 for(int j = 0; j < array1.length(); j++)
173.                 {
174.                     JSONObject jsonObject3 = array1.getJSONObject(j);
175.                     Ingredient ingredient = gson.fromJson(jsonObject3.toString(), Ingredient.class);
176.                     ingr.add(ingredient);
177.                 }
178.                 Matching matching = gson.fromJson(body, Matching.class);
179.                 jsonObject = new JSONObject();
180.                 result = INSERT.insertMatching(matching, ingr);
181.                 response += jsonObject.append("result", result);
182.                 break;
183.
184.             default:
185.                 response = "REQUEST NOT RECONGNIZED";
186.                 break;
187.         }
188.         return response;
189.     }
190.
191.
192. }
```

TastevinMain

```
1. package server;
2.
3.
4. import java.io.IOException;
5. import java.net.InetSocketAddress;
6.
7. import com.sun.net.httpserver.HttpServer;
8. /*
9.  * Main for the launch of the server
10. */
11. public class TastevinMain {
12.
13.     public static void main(String[] args) throws IOException
14.     {
15.         HttpServer server = HttpServer.create(new InetSocketAddress("68.66.253
16.         .202", 8080), 0);
17.         System.out.println("Server started at " + 8080);
18.         server.createContext("/", new HTTP());
19.         server.setExecutor(null);
20.         server.start();
21.     }
22. }
```

f) Android Code

CartPresenter

```
1. package com.tastevin.android.tastevin.Cart_Presenter;
2.
3. import android.app.Activity;
4. import android.content.Context;
5. import android.content.Intent;
6. import android.content.SharedPreferences;
7. import android.support.annotation.Nullable;
8.
9. import com.android.volley.Response;
```



```

10. import com.android.volley.VolleyError;
11. import com.paypal.android.sdk.payments.PayPalConfiguration;
12. import com.paypal.android.sdk.payments.PayPalPayment;
13. import com.paypal.android.sdk.payments.PayPalService;
14. import com.paypal.android.sdk.payments.PaymentActivity;
15. import com.paypal.android.sdk.payments.PaymentConfirmation;
16. import com.tastevin.android.tastevin.Cart_View.CartViewInterface;
17. import com.tastevin.android.tastevin.Model.Config;
18. import com.tastevin.android.tastevin.Model.ClientRequests;
19. import com.tastevin.android.tastevin.Model.Purchase;
20. import com.tastevin.android.tastevin.Model.Wine;
21.
22. import org.json.JSONException;
23. import org.json.JSONObject;
24.
25. import java.math.BigDecimal;
26. import java.util.ArrayList;
27.
28. import static android.app.Activity.RESULT_OK;
29.
30.     /*
31.      * Presenter of the Cart Activity.
32.      */
33.
34. public class CartPresenter implements CartPresenterInterface
35. {
36.     private CartViewInterface cart_view_interface;
37.     private ClientRequests parse;
38.     private SharedPreferences sharedPref;
39.     private ArrayList<Wine> wines;
40.     private static final int PAYPAL_REQUEST_CODE = 7171;
41.     private static PayPalConfiguration config = new PayPalConfiguration()
42.         .environment(PayPalConfiguration.ENVIRONMENT_SANDBOX)
43.         .clientId(Config.PAYPAL_CLIENT_ID);
44.     private float tot;
45.
46.     /*
47.      * Constructor():
48.      * To be able to make changes to the view,
49.      * it takes as argument an object of type
50.      * CartViewInterface.
51.      * It also initializes the connection to the server
52.      * and the Shared Preferences.
53.      */
54.
55.     public CartPresenter(CartViewInterface cart_view_interface)
56.     {
57.         this.cart_view_interface = cart_view_interface;
58.         parse = new ClientRequests(cart_view_interface.getActivity());
59.         sharedPref = cart_view_interface.getActivity().getSharedPreferences(
60.             "com.tastevin.android.tastevin.PREFERENCE_FILE_KEY", Context.M
61.             ODE_PRIVATE);
62.     }
63.
64.     /*
65.      * getWinesList():
66.      * This method gets all the wines that the

```

```

66.         user putted in the cart and it displays
67.         them in the list with the total.
68.     */
69.
70.     public void getWinesList()
71.     {
72.         int size = sharedPref.getInt("winesNbr", 0);
73.         wines = new ArrayList<>();
74.         for(int i = 1; i < size+1; i++)
75.             wines.add(new Wine(sharedPref.getInt("wineID"+i, 0), sharedPref.ge
76. tString("wineName"+i, "Null"),
77.         sharedPref.getInt("wineNbr"+i, 0)));
78.         tot = sharedPref.getFloat("winesTOT", 0);
79.         cart_view_interface.showWineList(wines, tot);
80.     }
81.     /*
82.     deleteWine():
83.         This method deletes a wine from the Shared
84.         Preferences and from the ArrayList and then
85.         it refreshes the list and the total.
86.     */
87.
88.     public void deleteWine(int pos)
89.     {
90.         int nbr, index;
91.         float price;
92.
93.         SharedPreferences.Editor editor = sharedPref.edit();
94.         price = sharedPref.getFloat("winePrice"+(pos+1), 0);
95.         nbr = sharedPref.getInt("wineNbr"+(pos+1), 0);
96.         tot = sharedPref.getFloat("winesTOT", 0);
97.         tot -= price*nbr;
98.         editor.remove("wineName"+(pos+1));
99.         editor.remove("wineID"+(pos+1));
100.        editor.remove("wineNbr"+(pos+1));
101.        editor.remove("winePrice"+(pos+1));
102.        index = sharedPref.getInt("winesNbr", 0);
103.        editor.putInt("winesNbr", (index-1));
104.        editor.putFloat("winesTOT", tot);
105.        editor.commit();
106.        wines.remove(pos);
107.        cart_view_interface.showWineList(wines, tot);
108.    }
109.
110.    /*
111.    createPurchase():
112.        This method sends a request to create a
113.        purchase to the server. It creates the
114.        purchase and then it sends it to the
115.        server. If the connection has problems, the presenter will
116.    use
117.        the view method "displayToast" to create a Toast
118.        that alerts the user of the network error.
119.        If the connection was return Success, the request was
120.        successful, otherwise it will display an error.
121.    */

```

```

121.
122.     public void createPurchase()
123.     {
124.         Purchase purchase = getPurchase();
125.         Response.Listener<JSONObject> listenerResponse = new Response.L
126.         istener<JSONObject>() {
127.             public void onResponse(JSONObject response)
128.             {
129.                 if(response.toString().contains("Success")) {
130.                     cart_view_interface.displayToast("Purchase Done");
131.
132.                     clearList();
133.                 }
134.                 else
135.                     cart_view_interface.displayToast("Network Error");
136.             }
137.         };
138.         Response.ErrorListener errorListener = new Response.ErrorListen
139.         er() {
140.             @Override
141.             public void onErrorResponse(VolleyError error) {
142.                 cart_view_interface.displayToast("Network Error");
143.             }
144.         };
145.         parse.insertPurchase(purchase, listenerResponse, errorListener)
146.         ;
147.     }
148.
149.     /*
150.     getPurchase():
151.     This method creates the object of the purchase
152.     needed from the createPurchase().
153.     */
154.
155.     public Purchase getPurchase()
156.     {
157.         int id = sharedPref.getInt("userID", 0);
158.         Purchase purchase = new Purchase(id);
159.         for(int i = 0; i < wines.size(); i++)
160.         {
161.             for(int k = 0; k < wines.get(i).getNbr(); k++)
162.                 purchase.addWine(wines.get(i));
163.         }
164.         return purchase;
165.     }
166.
167.     /*
168.     clearList():
169.     This method, after the purchase is done,
170.     clears the entire list and the wines from
171.     the Shared Preferences.
172.     */
173.
174.     public void clearList()
175.     {

```

```

173.         SharedPreferences.Editor editor = sharedPref.edit();
174.         int size = sharedPref.getInt("winesNbr", 0);
175.         wines = new ArrayList<>();
176.         for(int i = 1; i < size+1; i++)
177.         {
178.             editor.remove("wineName"+i);
179.             editor.remove("wineID"+i);
180.             editor.remove("winePrice"+i);
181.             editor.remove("wineNbr"+i);
182.             editor.remove("winesNbr");
183.             editor.remove("winesTOT");
184.         }
185.         editor.commit();
186.     }
187.
188.     /*
189.     processPayment():
190.     This method starts the process payment for the
191.     paypal.
192.     */
193.
194.     public void processPayment()
195.     {
196.         PayPalPayment payPalPayment = new PayPalPayment(new BigDecimal(
197.             String.valueOf(tot)), "DKK",
198.             "Buy Wine", PayPalPayment.PAYMENT_INTENT_SALE);
199.         Intent intent = new Intent(cart_view_interface.getActivity(), P
200.             aymentActivity.class);
201.         intent.putExtra(PayPalService.EXTRA_PAYPAL_CONFIGURATION, confi
202.             g);
203.         intent.putExtra(PaymentActivity.EXTRA_PAYMENT, payPalPayment);
204.         cart_view_interface.getActivity().startActivityForResult(intent
205.             , PAYPAL_REQUEST_CODE);
206.     }
207.
208.     public void startService()
209.     {
210.         Intent intent = new Intent(cart_view_interface.getActivity(), P
211.             ayPalService.class);
212.         intent.putExtra(PayPalService.EXTRA_PAYPAL_CONFIGURATION, confi
213.             g);
214.         cart_view_interface.getActivity().startService(intent);
215.     }
216.
217.     /*
218.     processPayment():
219.     This method checks the results of the
220.     payment of the Paypal. If the payment
221.     was successful it will be registered in
222.     the database. Otherwise it will display
223.     and error.
224.     */
225.
226.     public void check(int requestCode, int resultCode, @Nullable Intent
227.         data)
228.     {

```



```

222.         if(requestCode == PAYPAL_REQUEST_CODE){
223.             if(resultCode == RESULT_OK){
224.                 PaymentConfirmation confirmation = data.getParcelableEx
tra(PaymentActivity.EXTRA_RESULT_CONFIRMATION);
225.                 if(confirmation != null){
226.                     try{
227.                         String paymentDetails = confirmation.toJSONObject()
.toString(4);
228.                         cart_view_interface.displayToast(paymentDetails
);
229.                         createPurchase();
230.                     }
231.                     catch(JSONException e){
232.                         e.printStackTrace();
233.                     }
234.                 }
235.                 else if(resultCode == Activity.RESULT_CANCELED)
cart_view_interface.displayToast("Cancel");
236.             }
237.         }
238.     }
239.     else if(resultCode == PaymentActivity.RESULT_EXTRAS_INVALID){
240.         cart_view_interface.displayToast("Invalid");
241.     }
242. }
243. }

```

Cart Presenter Interface

```

1. package com.tastevin.android.tastevin.Cart_Presenter;
2.
3. import android.content.Intent;
4. import android.support.annotation.Nullable;
5.
6. public interface CartPresenterInterface
7. {
8.     void getWinesList();
9.     void clearList();
10.    void deleteWine(int pos);
11.    void createPurchase();
12.    void processPayment();
13.    void check(int requestCode, int resultCode, @Nullable Intent data);
14.    void startService();
15. }

```

Cart View

```

1. package com.tastevin.android.tastevin.Cart_View;
2.
3. import android.app.Activity;
4. import android.content.DialogInterface;
5. import android.content.Intent;

```

```

6. import android.os.Bundle;
7. import android.support.annotation.Nullable;
8. import android.support.v7.app.AlertDialog;
9. import android.support.v7.app.AppCompatActivity;
10. import android.view.View;
11. import android.widget.AdapterView;
12. import android.widget.Button;
13. import android.widget.ListView;
14. import android.widget.TextView;
15. import android.widget.Toast;
16.
17. import com.paypal.android.sdk.payments.PayPalService;
18. import com.tastevin.android.tastevin.Cart_Presenter.CartPresenter;
19. import com.tastevin.android.tastevin.Cart_Presenter.CartPresenterInterface;
20. import com.tastevin.android.tastevin.Model.CheckoutAdapter;
21. import com.tastevin.android.tastevin.Model.Wine;
22. import com.tastevin.android.tastevin.R;
23.
24. import java.util.ArrayList;
25.
26.     /*
27.         View of the Cart Activity.
28.     */
29.
30. public class CartView extends AppCompatActivity implements CartViewInterface{
31.
32.     private ListView listView;
33.     private AlertDialog.Builder builder;
34.     private TextView textView;
35.     private CartPresenterInterface cart_presenter_interface;
36.
37.     /*
38.         onCreate():
39.         It creates the layout and initialize the presenter.
40.     */
41.
42.     protected void onDestroy()
43.     {
44.         stopService(new Intent(this, PayPalService.class));
45.         super.onDestroy();
46.     }
47.
48.     protected void onCreate(Bundle savedInstanceState)
49.     {
50.         super.onCreate(savedInstanceState);
51.         setContentView(R.layout.activity_cart);
52.         cart_presenter_interface = new CartPresenter(this);
53.         cart_presenter_interface.startService();
54.         listView = findViewById(R.id.list);
55.         textView = findViewById(R.id.textView4);
56.         cart_presenter_interface.getWinesList();
57.         listView.setOnItemClickListener(new AdapterView.OnItemClickListener()
58.         {
59.             @Override
60.             public void onItemClick(AdapterView<?> parent, View view, final in
61. t position, long id) {

```



```

60.         setDialog(position);
61.     }
62. });
63.     Button button = findViewById(R.id.button3);
64.     button.setOnClickListener(new View.OnClickListener() {
65.         @Override
66.         public void onClick(View v) {
67.             cart_presenter_interface.processPayment();
68.         }
69.     });
70. }
71.
72. /*
73.     setDialog():
74.         This method creates an AlertDialog
75.         when the user wants to delete a wine.
76. */
77.
78. public void setDialog(final int pos)
79. {
80.     builder = new AlertDialog.Builder(this);
81.     builder.setMessage("Do you want to delete this item?");
82.     builder.setPositiveButton("Yes", new DialogInterface.OnClickListener()
83. {
84.         @Override
85.         public void onClick(DialogInterface dialog, int which) {
86.             cart_presenter_interface.deleteWine(pos);
87.         }
88.     }).setNegativeButton("No", new DialogInterface.OnClickListener() {
89.         @Override
90.         public void onClick(DialogInterface dialog, int which) {
91.             dialog.dismiss();
92.         }
93.     });
94.     builder.create();
95.     builder.show();
96. }
97. /*
98.     getActivity():
99.         Returns the Cart activity.
100. */
101.
102. public Activity getActivity()
103. {
104.     return this;
105. }
106.
107. /*
108.     showWineList():
109.         This method displays the list of wine received
110.         present in the cart.
111. */
112.
113. public void showWineList(ArrayList<Wine> wines, float tot)
114. {

```

```

115.         CheckoutAdapter checkoutAdapter = new CheckoutAdapter(this, win
    es);
116.         listView.setAdapter(checkoutAdapter);
117.         textView.setText(tot+" $");
118.     }
119.
120.     /*
121.         displayToast():
122.         This method displays a toast with a message.
123.     */
124.
125.     public void displayToast(String text)
126.     {
127.         Toast.makeText(getActivity(), text, Toast.LENGTH_LONG).show();
128.     }
129.
130.     /*
131.         onActivityResult:
132.         Check the results of the paypal activity.
133.     */
134.
135.     protected void onActivityResult(int requestCode, int resultCode, @N
        ullable Intent data)
136.     {
137.         cart_presenter_interface.check(requestCode, resultCode, data);
138.     }
139. }

```

Cart View Interface

```

1. package com.tastevin.android.tastevin.Cart_View;
2.
3. import android.app.Activity;
4.
5. import com.tastevin.android.tastevin.Model.Wine;
6.
7. import java.util.ArrayList;
8.
9. public interface CartViewInterface
10. {
11.     Activity getActivity();
12.     void showWineList(ArrayList<Wine> wines, float tot);
13.     void displayToast(String text);
14. }

```

Dashboard Presenter

```

1. package com.tastevin.android.tastevin.Dashboard_Presenter;
2.

```

```

3. import android.content.Context;
4. import android.content.Intent;
5. import android.content.SharedPreferences;
6.
7. import com.android.volley.Response;
8. import com.android.volley.VolleyError;
9. import com.tastevin.android.tastevin.Dashboard_View.DashboardViewInterface;
10. import com.tastevin.android.tastevin.Model.ClientRequests;
11. import com.tastevin.android.tastevin.Model.Filter;
12. import com.tastevin.android.tastevin.Model.Parser;
13. import com.tastevin.android.tastevin.WineInfo_View.WineInfoView;
14.
15. import org.json.JSONException;
16.
17. import java.text.ParseException;
18. import java.util.ArrayList;
19.
20.     /*
21.         Presenter of the Dashboard Fragment.
22.     */
23.
24. public class DashboardPresenter implements DashboardPresenterInterface
25. {
26.     private DashboardViewInterface dashboard_view_interface;
27.     private ClientRequests client;
28.     private SharedPreferences sharedPref;
29.
30.     /*
31.         Constructor():
32.         To be able to make changes to the view,
33.         it takes as argument an object of type
34.         DashboardViewInterface.
35.         It also initializes the connection to the server
36.         and the Shared Preferences.
37.     */
38.
39.     public DashboardPresenter(DashboardViewInterface dashboard_view_interface)
40.     {
41.         this.dashboard_view_interface = dashboard_view_interface;
42.         client = new ClientRequests(dashboard_view_interface.getActivity());
43.         sharedPref = dashboard_view_interface.getActivity().getSharedPreference
44. es(
45.         "com.tastevin.android.tastevin.PREFERENCE_FILE_KEY", Context.M
46.         ODE_PRIVATE);
47.     }
48.     /*
49.         getActivities():
50.         It asks to the server a JSON Object that contains
51.         the latest activities of the friends of the user.
52.         It contains purchases, matchings and reviews.
53.         If the connection has problems, the presenter will use
54.         the view method "displayToast" to create a Toast
55.         that alerts the user of the network error.
56.         If the connection was successful, the JSON Object is

```

```

56.         converted to ArrayList using the method "parseJSONActivities", the
57.         n
58.         the list is sorted by date using the Filter class and then
59.         it's passed to the view in order to be displayed.
60.     */
61.     public void getActivitiesList()
62.     {
63.         Response.Listener<String> listenerResponse = new Response.Listener<String>() {
64.
65.             @Override
66.             public void onResponse(String response)
67.             {
68.                 try {
69.                     ArrayList<Object> items = Filter.sortActivities(Parser.
70.                         parseJSONActivities(response));
71.                     dashboard_view_interface.showActivities(items);
72.                 } catch (JSONException e) {
73.                     dashboard_view_interface.displayToast("Network Error");
74.                 } catch (ParseException e) {
75.                     dashboard_view_interface.displayToast("Network Error");
76.                 }
77.             }
78.         };
79.         Response.ErrorListener errorListener = new Response.ErrorListener() {
80.
81.             public void onErrorResponse(VolleyError error) {
82.                 dashboard_view_interface.displayToast("Network Error");
83.             }
84.         };
85.         client.getFriendList(sharedPref.getInt("userID", 0), listenerResponse,
86.             errorListener);
87.     }
88.     /*
89.     checkWine():
90.         The checkWine is used by the Dashboard View when
91.         the user wants to check out a wine listed in the
92.         activities. It takes the id of the wine selected
93.         and it gets passed to the activity "WineInfo" that
94.         will display the full info of the wine.
95.     */
96.
97.     public void checkWine(int id)
98.     {
99.         Intent intent = new Intent(dashboard_view_interface.getActivity(), Win
100.             eInfoView.class);
101.         intent.putExtra("id", id);
102.         dashboard_view_interface.getActivity().startActivity(intent);
103.     }

```

Dashboard Presenter Interface

```

1. package com.tastevin.android.tastevin.Dashboard_Presenter;
2.
3. public interface DashboardPresenterInterface
4. {
5.     void getActivitiesList();
6.     void checkWine(int id);
7. }
```

Dashboard View

```

1. package com.tastevin.android.tastevin.Dashboard_View;
2.
3. import android.app.AlertDialog;
4. import android.content.DialogInterface;
5. import android.os.Bundle;
6. import android.support.annotation.NonNull;
7. import android.support.annotation.Nullable;
8. import android.support.v4.app.Fragment;
9. import android.view.LayoutInflater;
10. import android.view.View;
11. import android.view.ViewGroup;
12. import android.widget.AdapterView;
13. import android.widget.AdapterView;
14. import android.widget.AdapterView;
15. import android.widget.AdapterView;
16. import android.widget.AdapterView;
17. import android.widget.AdapterView;
18. import android.widget.AdapterView;
19. import android.widget.AdapterView;
20. import android.widget.AdapterView;
21. import android.widget.AdapterView;
22. import android.widget.AdapterView;
23.
24. import java.util.ArrayList;
25.
26. /*
27.     View of the Dashboard Fragment.
28. */
29.
30. public class DashboardView extends Fragment implements DashboardViewInterface
31. {
32.     private ListView listView;
33.     private DashboardAdapter adapter;
34.     private AlertDialog.Builder builder;
35.     private DashboardPresenterInterface dashboard_presenter_interface;
36.
37.     /*
38.     onCreateView():
39.     It creates the layout and initialize the presenter.
40.     */
41.
```

```

42.     public View onCreateView(@NonNull LayoutInflater inflater, @Nullable ViewGroup
      roup container, @Nullable Bundle savedInstanceState)
43.     {
44.         View layout = inflater.inflate(R.layout.fragment_dashboard, null);
45.         listView = layout.findViewById(R.id.list_dashboard);
46.         dashboard_presenter_interface = new DashboardPresenter(this);
47.         return layout;
48.     }
49.
50.     /*
51.         onViewCreated():
52.         It asks the presenter to get the list of the wine and
53.         set the onItemClickListener on the ListView. If the
54.         object clicked is of type Review or Matching, it will get open the
55.         wineInfo activity of that wine. If the object clicked is
56.         of type Purchase, it will open an AlertDialog where the user can c
      hoose
57.         the wine to check.
58.     */
59.
60.     public void onViewCreated(@NonNull View view, @Nullable Bundle savedInstan
      ceState)
61.     {
62.         super.onViewCreated(view, savedInstanceState);
63.         dashboard_presenter_interface.getActivitiesList();
64.         listView.setOnItemClickListener(new AdapterView.OnItemClickListener()
65.         {
66.             @Override
67.             public void onItemClick(AdapterView<?> parent, View view, int posi
      tion, long id) {
68.                 if(parent.getItemAtPosition(position) instanceof Review)
69.                     dashboard_presenter_interface.checkWine(((Review) parent.g
      etItemAtPosition(position)).getWineID());
70.                 else if(parent.getItemAtPosition(position) instanceof Purchase
71.                 )
72.                 {
73.                     final Purchase item = (Purchase) parent.getItemAtPosition(
      position);
74.                     createAlterDialogWines(item);
75.                 }
76.                 else if(parent.getItemAtPosition(position) instanceof Matching
77.                 )
78.                     dashboard_presenter_interface.checkWine(((Matching) parent
      .getItemAtPosition(position)).getWineID());
79.                 }
80.             });
81.         }
82.     }
83.     /*
84.         onResume():
85.         On resume of the activity, the list is refreshed.
86.     */
87.     public void onResume()
88.     {
89.         super.onResume();

```



```

88.     dashboard_presenter_interface.getActivitiesList();
89. }
90.
91.  /*
92.     createAlterDialogWines():
93.         This method creates the AlertDialog for the purchase clicked.
94.         The user can then choose one and at the ok click, the WineInfo
95.         activity will open.
96.  */
97.
98.  public void createAlterDialogWines(final Purchase purchase)
99.  {
100.      builder = new AlertDialog.Builder(getContext());
101.      final int[] checked = {0};
102.      builder.setTitle("Do you want to check a wine of this purchase?
103. ");
104.      ArrayList<String> temp = new ArrayList<>();
105.      for(int i = 0; i < purchase.getWineList().size(); i++)
106.      {
107.          if(!(temp.contains(purchase.getWineList().get(i))))
108.              temp.add(purchase.getWineList().get(i));
109.      }
110.      String[] items = new String[temp.size()];
111.      temp.toArray(items);
112.      builder.setSingleChoiceItems(items, checked[0], new DialogInter
113. face.OnClickListener() {
114.     @Override
115.     public void onClick(DialogInterface dialog, int which) {
116.         checked[0] = which;
117.     }
118. });
119. builder.setPositiveButton("Ok", new DialogInterface.OnClickListener
120. ener() {
121.     @Override
122.     public void onClick(DialogInterface dialog, int which) {
123.         dashboard_presenter_interface.checkWine(purchase.getWin
124. eArrayList().get(checked[0]).getId());
125.     }
126. });
127. builder.setNeutralButton("Back", new DialogInterface.OnClickListener
128. tener() {
129.     @Override
130.     public void onClick(DialogInterface dialog, int which) {
131.         dialog.dismiss();
132.     }
133. });
134. AlertDialog alertDialog = builder.create();
135. alertDialog.show();
136. }
137.
138.  /*
139.     getView():
140.         This method returns the view of the Fragment.
141.  */
142.
143.  public View getView()
144.  {

```



```

140.         return super.getView();
141.     }
142.
143.     /*
144.         displayToast():
145.         This method displays a toast with a message.
146.     */
147.
148.     public void displayToast(String text)
149.     {
150.         Toast.makeText(getActivity(), text, Toast.LENGTH_LONG).show();
151.     }
152.
153.     /*
154.         showActivities():
155.         This method displays the list of wine received from
156.         the server.
157.     */
158.
159.     public void showActivities(ArrayList<Object> activities)
160.     {
161.         adapter = new DashboardAdapter(getActivity(), activities);
162.         listView.setAdapter(adapter);
163.     }
164. }

```

Dashboard View Interface

```

1. package com.tastevin.android.tastevin.Dashboard_View;
2.
3. import android.app.Activity;
4.
5. import java.util.ArrayList;
6.
7. public interface DashboardViewInterface {
8.
9.     Activity getActivity();
10.     void displayToast(String text);
11.     void showActivities(ArrayList<Object> activities);
12. }

```

Login Presenter

```

1. package com.tastevin.android.tastevin.Login_Presenter;
2.
3. import android.content.Context;
4. import android.content.Intent;

```



```

5. import android.content.SharedPreferences;
6. import android.support.annotation.Nullable;
7.
8. import com.android.volley.Response;
9. import com.android.volley.VolleyError;
10. import com.tastevin.android.tastevin.Login_View.LoginViewInterface;
11. import com.tastevin.android.tastevin.MainActivity_View.MainActivityView;
12. import com.tastevin.android.tastevin.Model.ClientRequests;
13. import com.tastevin.android.tastevin.Model.Parser;
14. import com.tastevin.android.tastevin.Model.User;
15.
16. import org.json.JSONException;
17.
18. import java.text.ParseException;
19.
20.     /*
21.         Presenter of the Login Activity.
22.     */
23.
24. public class LoginPresenter implements LoginPresenterInterface
25. {
26.     private SharedPreferences sharedPref;
27.     private LoginViewInterface login_view_interface;
28.     private ClientRequests parse;
29.     private User user;
30.
31.     /*
32.         Constructor():
33.         To be able to make changes to the view,
34.         it takes as argument an object of type
35.         LoginViewInterface.
36.         It also initializes the connection to the server
37.         and the Shared Preferences.
38.     */
39.
40.     public LoginPresenter(LoginViewInterface login_view_interface)
41.     {
42.         this.login_view_interface = login_view_interface;
43.         parse = new ClientRequests(login_view_interface.getActivity());
44.         sharedPref = login_view_interface.getActivity().getSharedPreferences(
45.             "com.tastevin.android.tastevin.PREFERENCE_FILE_KEY", Context.M
ODE_PRIVATE);
46.     }
47.
48.     /*
49.         login():
50.         This method asks the server all the information
51.         about the user with the mail inserted.
52.         If the connection has problems, the presenter will use
53.         the view method "displayToast" to create a Toast
54.         that alerts the user of the network error.
55.         If the connection was successful, the JSON Object is
56.         converted to User object and then all the info are stored
57.         in the Shared Preferences.
58.     */
59.

```

```

60.     public void login(final String mail)
61.     {
62.         Response.Listener<String> listenerResponse = new Response.Listener<String>() {
63.
64.             @Override
65.             public void onResponse(String response)
66.             {
67.                 try {
68.                     user = Parser.parseJSONUser(response.toString(), mail);
69.                     SharedPreferences.Editor editor = sharedPref.edit();
70.                     editor.clear();
71.                     editor.putInt("userID", user.getId());
72.                     editor.putString("userName", user.getFirstName() + " " + user
73.                         .getLastName());
74.                     editor.putString("userBirthday", user.getBirthday());
75.                     editor.putString("userEmail", user.getEmail());
76.                     editor.putInt("userSweet", user.getSweet());
77.                     editor.putInt("userSour", user.getSour());
78.                     editor.putInt("userSalty", user.getSalty());
79.                     editor.putInt("userBitter", user.getBitter());
80.                     editor.putInt("userUmami", user.getUmami());
81.                     editor.commit();
82.                     Intent intent = new Intent(login_view_interface.getActivit
83.                         y(),
84.                         MainActivityView.class);
85.                     intent.putExtra("name", user.getFirstName() + " " + user.getL
86.                         astName());
87.                     intent.putExtra("email", mail);
88.                     login_view_interface.getActivity().finish();
89.                     login_view_interface.getActivity().startActivity(intent);
90.                 } catch (JSONException e) {
91.                     login_view_interface.displayToast("Failed Login");
92.                 } catch (ParseException e) {
93.                     login_view_interface.displayToast("Failed Login");
94.                 }
95.             }
96.         };
97.         Response.ErrorListener errorListener = new Response.ErrorListener() {
98.
99.             @Override
100.            public void onErrorResponse(VolleyError error) {
101.                login_view_interface.displayToast("Failed Login");
102.            }
103.        };
104.        parse.login(mail, listenerResponse, errorListener);
105.
106.        /*
107.        alreadyLogged():
108.        This method checks if the user is
109.        already logged-in. If is true then it will
110.        open immediately the MainActivity.

```

```

110.         */
111.
112.         public void alreadyLogged()
113.         {
114.             String name = sharedPref.getString("userName", "None");
115.             String email = sharedPref.getString("userEmail", "None");
116.             Intent intent = new Intent(login_view_interface.getActivity(),
117.                                     MainActivityView.class);
118.             intent.putExtra("name", name);
119.             intent.putExtra("email", email);
120.             login_view_interface.getActivity().finish();
121.             login_view_interface.getActivity().startActivity(intent);
122.         }
123.     }

```

Login Presenter Interface

```

1. package com.tastevin.android.tastevin.Login_Presenter;
2.
3. public interface LoginPresenterInterface
4. {
5.     void login(String mail);
6.     void alreadyLogged();
7. }

```

Login View

```

1. package com.tastevin.android.tastevin.Login_View;
2.
3. import android.app.Activity;
4. import android.content.Intent;
5. import android.content.pm.PackageInfo;
6. import android.content.pm.PackageManager;
7. import android.content.pm.Signature;
8. import android.os.Bundle;
9. import android.support.annotation.Nullable;
10. import android.support.v7.app.AppCompatActivity;
11. import android.util.Base64;
12. import android.util.Log;
13. import android.widget.Toast;
14.
15. import com.facebook.AccessToken;
16. import com.facebook.CallbackManager;
17. import com.facebook.FacebookCallback;
18. import com.facebook.FacebookException;
19. import com.facebook.FacebookSdk;
20. import com.facebook.GraphRequest;
21. import com.facebook.GraphResponse;
22. import com.facebook.login.LoginManager;
23. import com.facebook.login.LoginResult;
24. import com.facebook.login.widget.LoginButton;
25. import com.tastevin.android.tastevin.Login_Presenter.LoginPresenter;

```

```

26. import com.tastevin.android.tastevin.Login_Presenter.LoginPresenterInterface;
27. import com.tastevin.android.tastevin.R;
28.
29. import org.json.JSONObject;
30.
31. import java.security.MessageDigest;
32. import java.security.NoSuchAlgorithmException;
33. import java.util.Arrays;
34.
35.     /*
36.         View of the Login Activity.
37.     */
38.
39.
40. public class LoginView extends AppCompatActivity implements LoginViewInterface
41. {
42.     private CallbackManager callbackManager;
43.     private LoginPresenterInterface login_presenter_interface;
44.
45.     /*
46.         onActivityResult():
47.         This method checks the results from the
48.         facebook login.
49.     */
50.
51.     protected void onActivityResult(int requestCode, int resultCode, @Nullable
Intent data)
52.     {
53.         super.onActivityResult(requestCode, resultCode, data);
54.         callbackManager.onActivityResult(requestCode, resultCode, data);
55.     }
56.
57.     /*
58.         onCreate():
59.         It creates the layout and initialize the presenter.
60.     */
61.
62.     protected void onCreate(Bundle savedInstanceState)
63.     {
64.
65.         super.onCreate(savedInstanceState);
66.         setContentView(R.layout.activity_login);
67.         login_presenter_interface = new LoginPresenter(this);
68.         callbackManager = CallbackManager.Factory.create();
69.         LoginButton loginButton = findViewById(R.id.loginButton);
70.         loginButton.setReadPermissions(Arrays.asList("public_profile"));
71.         if(AccessToken.getCurrentAccessToken() != null)
72.             login_presenter_interface.alreadyLogged();
73.         loginButton.registerCallback(callbackManager, new FacebookCallback<Log
inResult>() {
74.
75.             public void onSuccess(LoginResult loginResult) {
76.
77.                 final AccessToken accessToken = AccessToken.getCurrentAccessTo
ken();

```

```

78.         AccessToken.refreshCurrentAccessTokenAsync();
79.         GraphRequest request = GraphRequest.newMeRequest(accessToken,
80.             new GraphRequest.GraphJSONObjectCallback() {
81.                 @Override
82.                 public void onCompleted(JSONObject object, GraphResponse r
83.                     esponse) {
84.                     Log.d("response",response.toString());
85.                     login_presenter_interface.login(object.optString("email"
86.                         l"));
87.                 }
88.             });
89.         Bundle parameters = new Bundle();
90.         parameters.putString("fields","name, email");
91.         request.setParameters(parameters);
92.         request.executeAsync();
93.     }
94.
95.     public void onCancel()
96.     {
97.         FacebookSdk.sdkInitialize(getApplicationContext());
98.         LoginManager.getInstance().logout();
99.         finishAffinity();
100.    }
101.
102.    public void onError(FacebookException error)
103.    {
104.        FacebookSdk.sdkInitialize(getApplicationContext());
105.        LoginManager.getInstance().logout();
106.        finishAffinity();
107.    }
108.    });
109.
110.    //printKeyHash(); //to be kept out unless the app needs to be r
111.    einstalled on fb
112.    }
113.    /*
114.    printKeyHash():
115.        This method has Hash key is required
116.        for first-time FB app setup.
117.    */
118.
119.    private void printKeyHash() {
120.        try {
121.            PackageInfo info = getPackageManager().getPackageInfo("com.
122.                gui.tastevin",
123.                    PackageManager.GET_SIGNATURES);
124.            for(Signature signature:info.signatures){
125.                MessageDigest md = MessageDigest.getInstance("SHA");
126.                md.update(signature.toByteArray());
127.                Log.d("KeyHash", Base64.encodeToString(md.digest(), Bas
128.                    e64.DEFAULT));
129.            }
130.        } catch (PackageManager.NameNotFoundException e) {

```

```

129.         e.printStackTrace();
130.     } catch (NoSuchAlgorithmException e) {
131.         e.printStackTrace();
132.     }
133. }
134.
135.     /*
136.     getActivity():
137.         This method returns the Login Activity.
138.     */
139.
140.     public Activity getActivity() {
141.         return this;
142.     }
143.
144.     /*
145.     displayToast():
146.         This method displays a toast with a message.
147.     */
148.
149.     public void displayToast(String text) {
150.         Toast.makeText(this, text, Toast.LENGTH_LONG).show();
151.     }
152. }

```

Login View Interface

```

1. package com.tastevin.android.tastevin.Login_View;
2.
3. import android.app.Activity;
4.
5. public interface LoginViewInterface
6. {
7.     Activity getActivity();
8.     void displayToast(String text);
9. }

```

MainActivity Presenter

```

1. package com.tastevin.android.tastevin.MainActivity_Presenter;
2.
3. import android.content.Intent;
4.
5. import com.facebook.FacebookSdk;
6. import com.facebook.login.LoginManager;
7. import com.tastevin.android.tastevin.Cart_View.CartView;
8. import com.tastevin.android.tastevin.MainActivity_View.MainActivityViewInterface;
9.
10. import static com.facebook.FacebookSdk.getApplicationContext;
11.
12.     /*
13.     Presenter of the MainActivity.
14.     */

```



```

15.
16. public class MainActivityPresenter implements MainActivityPresenterInterface
17. {
18.     private MainActivityViewInterface mainActivity_view_interface;
19.
20.     /*
21.         Constructor():
22.         To be able to make changes to the view,
23.         it takes as argument an object of type
24.         MainActivityViewInterface.
25.     */
26.
27.     public MainActivityPresenter(MainActivityViewInterface mainActivity_view_i
28.         nterface)
29.     {
30.         this.mainActivity_view_interface = mainActivity_view_interface;
31.     }
32.     /*
33.         openCart():
34.         This method is called when the user presses
35.         the cart button. The activity
36.     */
37.
38.     public void openCart()
39.     {
40.         Intent intent = new Intent(mainActivity_view_interface.getActivity(),
41.             CartView.class);
42.         mainActivity_view_interface.getActivity().startActivity(intent);
43.     }
44.     /*
45.         closeApp():
46.         If the user chooses the log out
47.         option in the navigation drawer, this
48.         method is called and the app is closed.
49.     */
50.
51.     public void closeApp()
52.     {
53.         FacebookSdk.sdkInitialize(getApplicationContext());
54.         LoginManager.getInstance().logout();
55.         mainActivity_view_interface.getActivity().finishAffinity();
56.     }
57. }

```

MainActivity Presenter Interface

```

1. package com.tastevin.android.tastevin.MainActivity_Presenter;
2.
3. public interface MainActivityPresenterInterface
4. {
5.     void openCart();
6.     void closeApp();
7. }

```

MainActivity View

```

1. package com.tastevin.android.tastevin.MainActivity_View;
2.
3. import android.annotation.SuppressLint;
4. import android.app.Activity;
5. import android.content.Context;
6. import android.content.Intent;
7. import android.content.SharedPreferences;
8. import android.os.Bundle;
9. import android.support.design.widget.FloatingActionButton;
10. import android.support.design.widget.NavigationView;
11. import android.support.v4.app.Fragment;
12. import android.support.v4.app.FragmentManager;
13. import android.support.v4.app.FragmentTransaction;
14. import android.support.v4.view.GravityCompat;
15. import android.support.v4.widget.DrawerLayout;
16. import android.support.v7.app.ActionBarDrawerToggle;
17. import android.support.v7.app.AppCompatActivity;
18. import android.support.v7.widget.Toolbar;
19. import android.view.MenuItem;
20. import android.view.View;
21. import android.widget.TextView;
22. import android.widget.Toast;
23.
24. import com.facebook.FacebookSdk;
25. import com.facebook.login.LoginManager;
26. import com.tastevin.android.tastevin.Cart_View.CartView;
27. import com.tastevin.android.tastevin.Dashboard_View.DashboardView;
28. import com.tastevin.android.tastevin.MainActivity_Presenter.MainActivityPresen
    ter;
29. import com.tastevin.android.tastevin.MainActivity_Presenter.MainActivityPresen
    terInterface;
30. import com.tastevin.android.tastevin.Matching_View.MatchingView;
31. import com.tastevin.android.tastevin.R;
32. import com.tastevin.android.tastevin.Recommendation_View.RecommendationView;
33. import com.tastevin.android.tastevin.Shop_View.ShopView;
34.
35.     /*
36.      View of the MainActivity.
37.     */
38.
39. public class MainActivityView extends AppCompatActivity
40.     implements NavigationView.OnNavigationItemSelectedListener, MainActivi
    tyViewInterface{
41.
42.     private Fragment base_fragment = null;
43.     private FloatingActionButton floatingActionButton;
44.     private MainActivityPresenterInterface mainActivity_presenter_interface;
45.
46.     /*
47.      onCreate():
48.      It creates the layout and initialize the presenter.
49.     */
50.
51.     @SuppressWarnings("RestrictedApi")
52.     @Override

```

```

53.     protected void onCreate(Bundle savedInstanceState) {
54.         super.onCreate(savedInstanceState);
55.         setContentView(R.layout.activity_main);
56.         mainActivity_presenter_interface = new MainActivityPresenter(this);
57.
58.         Toolbar toolbar = findViewById(R.id.toolbar);
59.         setSupportActionBar(toolbar);
60.
61.         DrawerLayout drawer = findViewById(R.id.drawer_layout);
62.         ActionBarDrawerToggle toggle = new ActionBarDrawerToggle(
63.             this, drawer, toolbar, R.string.navigation_drawer_open, R.string.navigation_drawer_close);
64.         drawer.addDrawerListener(toggle);
65.         toggle.syncState();
66.
67.         NavigationView navigationView = findViewById(R.id.nav_view);
68.         navigationView.setNavigationItemSelectedListener(this);
69.
70.         View hView = navigationView.getHeaderView(0);
71.         TextView nav_user = hView.findViewById(R.id.userNameTextView);
72.         nav_user.setText(getIntent().getStringExtra("name"));
73.         TextView nav_user2 = hView.findViewById(R.id.userMailTextView);
74.         nav_user2.setText(getIntent().getStringExtra("email"));
75.
76.         FloatingActionButton = findViewById(R.id.fab);
77.         base_fragment = new DashboardView();
78.         FloatingActionButton.setVisibility(View.GONE);
79.         FloatingActionButton.setOnClickListener(new View.OnClickListener() {
80.             @Override
81.             public void onClick(View v)
82.             {
83.                 mainActivity_presenter_interface.openCart();
84.             }
85.         });
86.         fragmentStart();
87.     }
88.
89.     public void onBackPressed() {
90.         if (getFragmentManager().getBackStackEntryCount() > 0 ){
91.             getFragmentManager().popBackStack();
92.         } else {
93.             super.onBackPressed();
94.         }
95.     }
96.
97.     /*
98.     onNavigationItemSelected():
99.         This method handles the navigation drawer and based
100.        on the icon selected, the base fragment will change.
101.    */
102.
103.    @SuppressWarnings("RestrictedApi")
104.    public boolean onNavigationItemSelected(MenuItem item) {
105.
106.        int id = item.getItemId();
107.        if (id == R.id.nav_dashboard)
108.        {

```

```

109.         base_fragment = new DashboardView();
110.         floatingActionButton.setVisibility(View.GONE);
111.         fragmentStart();
112.     }
113.     else if (id == R.id.nav_matching)
114.     {
115.         base_fragment = new MatchingView();
116.         floatingActionButton.setVisibility(View.VISIBLE);
117.         fragmentStart();
118.     }
119.     else if(id == R.id.nav_recommendation)
120.     {
121.         base_fragment = new RecommendationView();
122.         floatingActionButton.setVisibility(View.VISIBLE);
123.         fragmentStart();
124.     }
125.     else if (id == R.id.nav_shop)
126.     {
127.         base_fragment = new ShopView();
128.         floatingActionButton.setVisibility(View.VISIBLE);
129.         fragmentStart();
130.     }
131.     else if (id == R.id.nav_logout)
132.     {
133.         mainActivity_presenter_interface.closeApp();
134.     }
135.
136.     DrawerLayout drawer = findViewById(R.id.drawer_layout);
137.     drawer.closeDrawer(GravityCompat.START);
138.     return true;
139. }
140.
141. /*
142.     fragmentStart():
143.         This method is used to change the base fragment.
144. */
145.
146. public void fragmentStart()
147. {
148.     if(base_fragment != null)
149.     {
150.         FragmentManager fragmentManager = getSupportFragmentManager
151.         ();
152.         FragmentTransaction ft = fragmentManager.beginTransaction()
153.         ;
154.         ft.replace(R.id.screen_area, base_fragment);
155.         ft.commit();
156.     }
157. }
158. /*
159.     getActivity():
160.         This method returns the Login Activity.
161. */
162. public Activity getActivity()
163. {

```

```

164.         return this;
165.     }
166.
167.     /*
168.         displayToast():
169.             This method displays a toast with a message.
170.     */
171.
172.     public void displayToast(String text) {
173.         Toast.makeText(this, text, Toast.LENGTH_LONG).show();
174.     }
175. }

```

MainActivity View Interface

```

1. package com.tastevin.android.tastevin.MainActivity_View;
2.
3. import android.app.Activity;
4.
5. public interface MainActivityViewInterface
6. {
7.     Activity getActivity();
8.     void displayToast(String text);
9. }

```

MatchedList Presenter

```

1. package com.tastevin.android.tastevin.MatchedList_Presenter;
2.
3. import android.content.Intent;
4.
5. import com.tastevin.android.tastevin.MatchedList_View.MatchedListViewInterface
6. ;
7. import com.tastevin.android.tastevin.Model.Wine;
8. import java.util.ArrayList;
9.
10.    /*
11.        Presenter of the MatchedList Activity.
12.    */
13.
14.    public class MatchedListPresenter implements MatchedListPresenterInterface
15.    {
16.        private MatchedListViewInterface matchedList_view_interface;
17.        private ArrayList<Wine> wines;
18.
19.        /*
20.            Constructor():
21.                To be able to make changes to the view,
22.                it takes as argument an object of type
23.                MatchedListViewInterface.
24.        */
25.

```

```

26.     public MatchedListPresenter(MatchedListViewInterface matchedList_view_inte
      rface)
27.     {
28.         this.matchedList_view_interface = matchedList_view_interface;
29.     }
30.
31.     /*
32.         getActivities():
33.             It retrieves from the intent the 3 wines
34.             to display.
35.     */
36.
37.     public ArrayList<Wine> getWineList(Intent intent)
38.     {
39.         wines = new ArrayList<>();
40.         wines.add((Wine) intent.getParcelableExtra("wine1"));
41.         wines.add((Wine) intent.getParcelableExtra("wine2"));
42.         wines.add((Wine) intent.getParcelableExtra("wine3"));
43.         return wines;
44.     }
45. }

```

MatchedList Presenter Interface

```

1.  package com.tastevin.android.tastevin.MatchedList_Presenter;
2.
3.  import android.content.Intent;
4.
5.  import com.tastevin.android.tastevin.Model.Wine;
6.
7.  import java.util.ArrayList;
8.
9.  public interface MatchedListPresenterInterface
10. {
11.     ArrayList<Wine> getWineList(Intent intent);
12. }

```

MatchedList View

```

1.  package com.tastevin.android.tastevin.MatchedList_View;
2.
3.  import android.app.Activity;
4.  import android.content.Intent;
5.  import android.os.Bundle;
6.  import android.support.design.widget.TabLayout;
7.  import android.support.v4.app.Fragment;
8.  import android.support.v4.app.FragmentManager;
9.  import android.support.v4.app.FragmentPagerAdapter;
10. import android.support.v4.view.ViewPager;
11. import android.support.v7.app.AppCompatActivity;
12. import android.support.v7.widget.Toolbar;
13. import android.view.LayoutInflater;
14. import android.view.View;
15. import android.view.ViewGroup;

```



```

16. import android.widget.Button;
17. import android.widget.ImageView;
18. import android.widget.TextView;
19. import android.widget.Toast;
20.
21. import com.tastevin.android.tastevin.MatchedList_Presenter.MatchedListPresente
    r;
22. import com.tastevin.android.tastevin.MatchedList_Presenter.MatchedListPresente
    rInterface;
23. import com.tastevin.android.tastevin.MatchedList_View.MatchedListViewInterface
    ;
24. import com.tastevin.android.tastevin.Model.Wine;
25. import com.tastevin.android.tastevin.R;
26. import com.tastevin.android.tastevin.WineInfo_View.WineInfoView;
27.
28. import java.util.ArrayList;
29.
30.     /*
31.         View of the MatchedList Activity.
32.     */
33.
34. public class MatchedListView extends AppCompatActivity implements MatchedListV
    iewInterface{
35.
36.     private SectionsPagerAdapter mSectionsPagerAdapter;
37.     private MatchedListPresenterInterface matchedList_presenter_interface;
38.     private ViewPager mViewPager;
39.
40.     /*
41.         onCreate():
42.         It creates the layout and initialize the presenter.
43.     */
44.
45.     protected void onCreate(Bundle savedInstanceState) {
46.         super.onCreate(savedInstanceState);
47.         setContentView(R.layout.activity_matched_list);
48.         matchedList_presenter_interface = new MatchedListPresenter(this);
49.
50.         Toolbar toolbar = findViewById(R.id.toolbar);
51.         setSupportActionBar(toolbar);
52.
53.         setSectionPagerAdapter();
54.
55.         mViewPager = findViewById(R.id.container);
56.         mViewPager.setAdapter(mSectionsPagerAdapter);
57.
58.         TabLayout tabLayout = findViewById(R.id.tabs);
59.
60.         mViewPager.addOnPageChangeListener(new TabLayout.TabLayoutOnPageChange
            Listener(tabLayout));
61.         tabLayout.addOnTabSelectedListener(new TabLayout.ViewPagerOnTabSelecte
            dListener(mViewPager));
62.
63.     }
64.
65.     /*
66.         getActivity():

```

```

67.         This method returns the Login Activity.
68.     */
69.
70.     public Activity getActivity()
71.     {
72.         return this;
73.     }
74.
75.     /*
76.         displayToast():
77.         This method displays a toast with a message.
78.     */
79.
80.     public void displayToast(String text)
81.     {
82.         Toast.makeText(getActivity(), text, Toast.LENGTH_LONG).show();
83.     }
84.
85.     /*
86.         setSectionPagerAdapter():
87.         This method sets the sections adapter for the fragments.
88.     */
89.
90.     public void setSectionPagerAdapter()
91.     {
92.         mSectionsPagerAdapter = new SectionsPagerAdapter(getSupportFragmentManager(),
93. matchedList_presenter_interface.getWineList(getIntent()));
94.     }
95.
96.     /*
97.         PlaceholderFragment
98.         This class handles the fragments inside the activity.
99.     */
100.
101.     public static class PlaceholderFragment extends Fragment {
102.
103.         private static final String ARG_SECTION_NUMBER = "section_number";
104.
105.         public PlaceholderFragment() {
106.
107.         }
108.
109.         public static PlaceholderFragment newInstance(int sectionNumber
110. , Wine wine) {
111.             PlaceholderFragment fragment = new PlaceholderFragment();
112.             Bundle args = new Bundle();
113.             args.putInt(ARG_SECTION_NUMBER, sectionNumber);
114.             args.putParcelable("wine", wine);
115.             fragment.setArguments(args);
116.             return fragment;
117.         }
118.
119.         @Override
120.         public View onCreateView(final LayoutInflater inflater, ViewGroup
121. container,
122.
123. Bundle savedInstanceState) {

```



```

120.        View rootView = inflater.inflate(R.layout.fragment_matched_
    list, container, false);
121.        ImageView imageView = rootView.findViewById(R.id.icon);
122.        TextView textView = rootView.findViewById(R.id.wineNameText
    );
123.        TextView textView2 = rootView.findViewById(R.id.wineNameWin
    ery);
124.        TextView textView3 = rootView.findViewById(R.id.wineTypeTex
    t);
125.        TextView textView4 = rootView.findViewById(R.id.winePriceTe
    xt);
126.        Wine wine = null;
127.        switch (getArguments().getInt(ARG_SECTION_NUMBER))
128.        {
129.            case 1:
130.                imageView.setImageResource(R.drawable.first);
131.                wine = getArguments().getParcelable("wine");
132.                textView.setText(wine.getName());
133.                if(wine.getYear() != 0)
134.                    textView2.setText(wine.getWinery() + " "+wine.ge
    tYear());
135.            else
136.                textView2.setText(wine.getWinery());
137.                textView3.setText(wine.getType());
138.                textView4.setText("$ " +String.valueOf(wine.getPric
    e()));
139.            break;
140.            case 2:
141.                imageView.setImageResource(R.drawable.second);
142.                wine = getArguments().getParcelable("wine");
143.                textView.setText(wine.getName());
144.                if(wine.getYear() != 0)
145.                    textView2.setText(wine.getWinery() + " "+wine.ge
    tYear());
146.            else
147.                textView2.setText(wine.getWinery());
148.                textView3.setText(wine.getType());
149.                textView4.setText("$ " +String.valueOf(wine.getPri
    ce()));
150.            break;
151.            case 3:
152.                imageView.setImageResource(R.drawable.third);
153.                wine = getArguments().getParcelable("wine");
154.                textView.setText(wine.getName());
155.                if(wine.getYear() != 0)
156.                    textView2.setText(wine.getWinery() + " "+wine.ge
    tYear());
157.            else
158.                textView2.setText(wine.getWinery());
159.                textView3.setText(wine.getType());
160.                textView4.setText("$ " +String.valueOf(wine.getPri
    ce()));
161.            break;
162.        }
163.        Button button = rootView.findViewById(R.id.buttonCheck);
164.        final Wine finalWine = wine;
165.        button.setOnClickListener(new View.OnClickListener() {

```

```

166.         @Override
167.         public void onClick(View v) {
168.             Intent intent = new Intent(getContext(), WineInfoVi
169. ew.class);
170.             intent.putExtra("id", finalWine.getId());
171.             getActivity().startActivity(intent);
172.         }
173.     });
174.     return rootView;
175. }
176.
177.     public class SectionsPagerAdapter extends FragmentPagerAdapter {
178.
179.         ArrayList<Wine> wines;
180.
181.         public SectionsPagerAdapter(FragmentManager fm, ArrayList<Wine>
182. wines)
183.         {
184.             super(fm);
185.             this.wines = wines;
186.         }
187.
188.         @Override
189.         public Fragment getItem(int position) {
190.             return PlaceholderFragment.newInstance(position + 1, wines.
191. get(position));
192.         }
193.
194.         @Override
195.         public int getCount() {
196.             return 3;
197.         }
198.     }

```

MatchedList View Interface

```

1. package com.tastevin.android.tastevin.MatchedList_View;
2.
3. import android.app.Activity;
4.
5. public interface MatchedListViewInterface
6. {
7.     Activity getActivity();
8.     void displayToast(String text);
9. }

```

Matching Presenter

```

1. package com.tastevin.android.tastevin.Matching_Presenter;

```



```

2.
3. import android.content.Context;
4. import android.content.Intent;
5. import android.content.SharedPreferences;
6.
7.
8. import com.android.volley.Response;
9. import com.android.volley.VolleyError;
10. import com.tastevin.android.tastevin.MatchedList_View.MatchedListView;
11. import com.tastevin.android.tastevin.Matching_View.MatchingViewInterface;
12. import com.tastevin.android.tastevin.Model.ClientRequests;
13. import com.tastevin.android.tastevin.Model.Ingredient;
14. import com.tastevin.android.tastevin.Model.Wine;
15. import com.tastevin.android.tastevin.Model.Parser;
16.
17. import org.json.JSONException;
18.
19. import java.util.ArrayList;
20.
21. /*
22.     Presenter of the Matching Fragment.
23. */
24.
25. public class MatchingPresenter implements MatchingPresenterInterface
26. {
27.     private Ingredient[] ingredients_1 = {new Ingredient(33, "Pork Chops"),
28.         new Ingredient(21, "Chicken Breast"),
29.         new Ingredient(41, "Venison Tenderloin"),
30.         new Ingredient(39, "Pot Roast")};
31.     private Ingredient[] ingredients_2 = {new Ingredient(9, "Tomato"),
32.         new Ingredient(10, "Potato"),
33.         new Ingredient(8, "Garlic"),
34.         new Ingredient(7, "Onion")};
35.     private Ingredient[] ingredients_3 = {new Ingredient(4, "Parmesan"),
36.         new Ingredient(13, "Bell Pepper"),
37.         new Ingredient(18, "Honey"),
38.         new Ingredient(42, "Apple Vinegar Cider"),
39.         new Ingredient(43, "Barbecue Sauce")};
40.     private Ingredient[] ingredients_4 = {new Ingredient(29, "Olive oil"),
41.         new Ingredient(34, "Soy Sauce"),
42.         new Ingredient(20, "Thyme"),
43.         new Ingredient(38, "Mushrooms Cream")};
44.     private ArrayList<Wine> wines;
45.     private MatchingViewInterface matching_view_interface;
46.     private ClientRequests parse;
47.     private SharedPreferences sharedPref;
48.
49.     /*
50.         Constructor():
51.         To be able to make changes to the view,
52.         it takes as argument an object of type
53.         MatchingViewInterface.
54.         It also initialize the connection to the server
55.         and the Shared Preferences.
56.     */
57.
58.     public MatchingPresenter(MatchingViewInterface matching_view_interface)

```

```

59.     {
60.         this.matching_view_interface = matching_view_interface;
61.         parse = new ClientRequests(matching_view_interface.getActivity());
62.         sharedPref = matching_view_interface.getActivity().getSharedPreferences
s(
63.             "com.tastevin.android.tastevin.PREFERENCE_FILE_KEY", Context.M
ODE_PRIVATE);
64.     }
65.
66.     /*
67.         createIngredientsArray():
68.         This method is called by the view in order to create
69.         the multidimensional String array that contains the names of
70.         the ingredients.
71.     */
72.
73.     public void createIngredientsArray()
74.     {
75.         String[][] temp = new String[4][ingredients_1.length+1];
76.         temp[0] = new String[]{"Missing", ingredients_1[0].getName(), ingredie
nts_1[1].getName(),
77.             ingredients_1[2].getName(), ingredients_1[3].g
etName()};
78.         temp[1] = new String[]{"Missing", ingredients_2[0].getName(), ingredie
nts_2[1].getName(),
79.             ingredients_2[2].getName(), ingredients_2[3].g
etName()};
80.         temp[2] = new String[]{"Missing", ingredients_3[0].getName(), ingredie
nts_3[1].getName(),
81.             ingredients_3[2].getName(), ingredients_3[3].g
etName(),
82.             ingredients_3[4].getName()};
83.         temp[3] = new String[]{"Missing", ingredients_4[0].getName(), ingredie
nts_4[1].getName(),
84.             ingredients_4[2].getName(), ingredients_4[3].g
etName()};
85.         matching_view_interface.createIngrArray(temp);
86.     }
87.
88.     /*
89.         getMatching():
90.         This method is called by the user in order to retrieve a
91.         matching with the chosen ingredients. First, it takes the
92.         id of the ingredients that were selected. If no ingredient
93.         was chosen, it calls the view method "displayToast" to
94.         show a Toast to the user to tell to him that he/she has
95.         to pick an ingredient. Then is send a request to the
96.         server, asking for a JSON object that contains 3
97.         wines. If the connection has problems, the presenter will use
98.         the view method "displayToast" to create a Toast
99.         that alerts the user of the network error.
100.        If the connection was successful, the JSON Object is
101.        converted to ArrayList using the method "parseJSONWines", t
hen
102.        an intent to the activity "MatchedListView" is called and t
he 3
103.        wines are putted as extra of the intent.

```

```

104.      */
105.
106.      public void getMatching(int[] position)
107.      {
108.          Integer[] ingr = new Integer[position.length];
109.          int wasSelected = 0;
110.
111.          for(int i = 0; i < position.length; i++)
112.          {
113.              if(position[i] != 0)
114.              {
115.                  switch (i)
116.                  {
117.                      case 0:
118.                          ingr[i] = ingredients_1[position[i]-
119.                          1].getId();
120.                          wasSelected++;
121.                          break;
122.                      case 1:
123.                          ingr[i] = ingredients_2[position[i]-
124.                          1].getId();
125.                          wasSelected++;
126.                          break;
127.                      case 2:
128.                          ingr[i] = ingredients_3[position[i]-
129.                          1].getId();
130.                          wasSelected++;
131.                          break;
132.                      case 3:
133.                          ingr[i] = ingredients_4[position[i]-
134.                          1].getId();
135.                          wasSelected++;
136.                          break;
137.                  }
138.              }
139.              else
140.              {
141.                  ingr[i] = 0;
142.              }
143.          }
144.
145.          if(wasSelected == 0)
146.              matching_view_interface.displayToast("Select at least one i
147.              ngredient");
148.          else
149.          {
150.              Response.Listener<String> listenerResponse = new Response.L
151.              istener<String>() {
152.                  public void onResponse(String response)
153.                  {
154.                      try {
155.                          wines = Parser.parseJSONWines(response);
156.                          Intent intent = new Intent(matching_view_interf
157.                          ace.getActivity(), MatchedListView.class);
158.                          intent.putExtra("wine1", wines.get(0));
159.                          intent.putExtra("wine2", wines.get(1));
160.                          intent.putExtra("wine3", wines.get(2));
161.                          matching_view_interface.getActivity().startActi
162.                          vity(intent);

```

```

153.                } catch (JSONException e) {
154.                    matching_view_interface.displayToast("Network E
    rror");
155.                }
156.            }
157.        };
158.        Response.ErrorListener errorListener = new Response.ErrorLi
    stener() {
159.            @Override
160.            public void onErrorResponse(VolleyError error) {
161.                matching_view_interface.displayToast("Network Error
    ");
162.            }
163.        };
164.        int id = sharedPref.getInt("userID", 0);
165.        parse.getMatchedList(ingr[0], ingr[1], ingr[2], ingr[3], id
    , listenerResponse, errorListener);
166.    }
167. }
168. }

```

Matching Presenter Interface

```

1. package com.tastevin.android.tastevin.Matching_Presenter;
2.
3. public interface MatchingPresenterInterface
4. {
5.     void createIngredientsArray();
6.     void getMatching(int[] position);
7. }

```

Matching View

```

1. package com.tastevin.android.tastevin.Matching_View;
2.
3. import android.app.AlertDialog;
4. import android.content.DialogInterface;
5. import android.os.Bundle;
6. import android.support.annotation.NonNull;
7. import android.support.annotation.Nullable;
8. import android.support.v4.app.Fragment;
9. import android.view.LayoutInflater;
10. import android.view.View;
11. import android.view.ViewGroup;
12. import android.widget.Button;
13. import android.widget.Toast;
14.
15. import com.tastevin.android.tastevin.Matching_Presenter.MatchingPresenter;
16. import com.tastevin.android.tastevin.Matching_Presenter.MatchingPresenterInter
    face;
17. import com.tastevin.android.tastevin.R;
18.
19. /*
20.     View of the Matching Fragment.

```

```

21.    */
22.
23. public class MatchingView extends Fragment implements MatchingViewInterface{
24.
25.     private AlertDialog.Builder builder;
26.     private MatchingPresenterInterface matching_presenter_interface;
27.     private String[] ingr1, ingr2, ingr3, ingr4;
28.     private int[] check = new int[4];
29.     private int checked = 0;
30.
31.     /*
32.         onCreateView():
33.         It creates the layout and initializes the presenter.
34.     */
35.
36.     public View onCreateView(@NonNull LayoutInflater inflater, @Nullable ViewGroup
roup container, @Nullable Bundle savedInstanceState)
37.     {
38.         matching_presenter_interface = new MatchingPresenter(this);
39.         return inflater.inflate(R.layout.fragment_get_matching, null);
40.     }
41.
42.     /*
43.         onViewCreated():
44.         It sets all the click listener of the 4 buttons
45.         of the ingredients and the button to get the matching.
46.         The first 4 buttons will open an AlertDialog with a list
47.         of ingredients and the last one will open the MatchedList
48.         activity with the 3 wines matched info.
49.     */
50.
51.     public void onViewCreated(@NonNull final View view, @Nullable Bundle saved
InstanceState)
52.     {
53.         super.onViewCreated(view, savedInstanceState);
54.         matching_presenter_interface.createIngredientsArray();
55.
56.         final Button button1 = view.findViewById(R.id.button_ingredient_1);
57.         button1.setOnClickListener(new View.OnClickListener() {
58.             @Override
59.             public void onClick(View v) {
60.                 createAlertDialogForIngredients(ingr1, button1, 1);
61.                 builder.show();
62.             }
63.         });
64.         final Button button2 = view.findViewById(R.id.button_ingredient_2);
65.         button2.setOnClickListener(new View.OnClickListener() {
66.             @Override
67.             public void onClick(View v) {
68.                 createAlertDialogForIngredients(ingr2, button2, 2);
69.                 builder.show();
70.             }
71.         });
72.         final Button button3 = view.findViewById(R.id.button_ingredient_3);
73.         button3.setOnClickListener(new View.OnClickListener() {
74.             @Override
75.             public void onClick(View v) {

```

```

76.         createAlertDialogForIngredients(ingr3, button3, 3);
77.         builder.show();
78.     }
79. });
80. final Button button4 = view.findViewById(R.id.button_ingredient_4);
81. button4.setOnClickListener(new View.OnClickListener() {
82.     @Override
83.     public void onClick(View v) {
84.         createAlertDialogForIngredients(ingr4, button4, 4);
85.         builder.show();
86.     }
87. });
88. Button button5 = view.findViewById(R.id.getMatching);
89.
90. button5.setOnClickListener(new View.OnClickListener() {
91.     @Override
92.     public void onClick(View v) {
93.         matching_presenter_interface.getMatching(check);
94.     }
95. });
96. }
97.
98. /*
99.     displayToast():
100.         This method displays a toast with a message.
101. */
102.
103. public void displayToast(String text)
104. {
105.     Toast.makeText(getActivity(), text, Toast.LENGTH_LONG).show();
106. }
107.
108. /*
109.     createIngrArray():
110.         This method takes a multidimensional array from the present
111.         er
112.         and then it will divide the array in the 4 arrays for
113.         the ingredients.
114. */
115.
116. public void createIngrArray(String[][] ingr)
117. {
118.     ingr1 = ingr[0];
119.     ingr2 = ingr[1];
120.     ingr3 = ingr[2];
121.     ingr4 = ingr[3];
122. }
123.
124. /*
125.     createAlertDialogForIngredients():
126.         This method creates the alter dialog for the choice
127.         of the ingredients.
128. */
129. public void createAlertDialogForIngredients(final String[] ingredie
nts, final Button button, final int id)

```



```

130.         {
131.             builder = new AlertDialog.Builder(getContext());
132.             builder.setSingleChoiceItems(ingredients, checked, new DialogInterface.OnClickListener() {
133.                 @Override
134.                 public void onClick(DialogInterface dialog, int which) {
135.                     checked = which;
136.                 }
137.             }).setPositiveButton("Ok", new DialogInterface.OnClickListener(
138.             ) {
139.                 @Override
140.                 public void onClick(DialogInterface dialog, int which) {
141.                     switch (id)
142.                     {
143.                         case 1:
144.                             check[0] = checked;
145.                             break;
146.                         case 2:
147.                             check[1] = checked;
148.                             break;
149.                         case 3:
150.                             check[2] = checked;
151.                             break;
152.                         case 4:
153.                             check[3] = checked;
154.                             break;
155.                     }
156.                     button.setText(ingredients[checked]);
157.                     checked = 0;
158.                 }
159.             }).setNegativeButton("Back", new DialogInterface.OnClickListener() {
160.                 @Override
161.                 public void onClick(DialogInterface dialog, int which) {
162.                     checked = 0;
163.                     button.setText(ingredients[checked]);
164.                 }
165.             });
166.             builder.create();
167.         }

```

Matching View Interface

```

1. package com.tastevin.android.tastevin.Matching_View;
2.
3. import android.app.Activity;
4.
5. public interface MatchingViewInterface
6. {
7.     Activity getActivity();
8.     void displayToast(String text);
9.     void createIngrArray(String[][] ingr);

```

```
10. }
```

Recommendation Presenter

```
1. package com.tastevin.android.tastevin.Recommendation_Presenter;
2.
3. import android.content.Context;
4. import android.content.Intent;
5. import android.content.SharedPreferences;
6.
7. import com.android.volley.Response;
8. import com.android.volley.VolleyError;
9. import com.tastevin.android.tastevin.Model.ClientRequests;
10. import com.tastevin.android.tastevin.Model.Wine;
11. import com.tastevin.android.tastevin.Model.Parser;
12. import com.tastevin.android.tastevin.Recommendation_View.RecommendationViewInt
    erface;
13. import com.tastevin.android.tastevin.WineInfo_View.WineInfoView;
14.
15. import org.json.JSONException;
16.
17. import java.util.ArrayList;
18.
19. /*
20.     Presenter of the Recommendation Fragment.
21. */
22.
23. public class RecommendationPresenter implements RecommendationPresenterInterfa
    ce
24. {
25.     private RecommendationViewInterface recommendation_view_interface;
26.     private ArrayList<Wine> wines;
27.     private ClientRequests parse;
28.     private SharedPreferences sharedPref;
29.
30.     /*
31.         Constructor():
32.         To be able to make changes to the view,
33.         it takes as argument an object of type
34.         RecommendationViewInterface.
35.         It also initialize the connection to the server
36.         and the Shared Preferences.
37.     */
38.
39.     public RecommendationPresenter(RecommendationViewInterface recommendation_
        view_interface)
40.     {
41.         this.recommendation_view_interface = recommendation_view_interface;
42.         parse = new ClientRequests(recommendation_view_interface.getActivity())
        );
43.         sharedPref = recommendation_view_interface.getActivity().getSharedPref
        erences(
44.             "com.tastevin.android.tastevin.PREFERENCE_FILE_KEY", Context.M
        ODE_PRIVATE);
45.     }
```

```

46.
47.  /*
48.      getWinesList():
49.          This method sends a request to the server in
50.          order to get the recommended wines. It asks for 5
51.          wines: the best sold, the best rated and the
52.          best 3 wines based on the user tastes.
53.          If the connection has problems, the presenter will use
54.          the view method "displayToast" to create a Toast
55.          that alerts the user of the network error.
56.          If the connection was successful, the JSON Object is
57.          converted to ArrayList using the method "parseJSONWines", then
58.          it's passed to the view in order to be displayed.
59.  */
60.
61.  public void getWinesList()
62.  {
63.      Response.Listener<String> listenerResponse = new Response.Listener<String>() {
64.
65.          @Override
66.          public void onResponse(String response)
67.          {
68.              try {
69.                  wines = Parser.parseJSONWines(response);
70.                  recommendation_view_interface.showWineList(wines);
71.              } catch (JSONException e) {
72.                  recommendation_view_interface.displayToast(" Error");
73.              }
74.          }
75.      };
76.      Response.ErrorListener errorListener = new Response.ErrorListener() {
77.
78.          @Override
79.          public void onErrorResponse(VolleyError error) {
80.              recommendation_view_interface.displayToast("Network ");
81.          }
82.      };
83.      int id = sharedPref.getInt("userID", 0);
84.      parse.getRecommendationList(id, listenerResponse, errorListener);
85.  }
86.  /*
87.      checkWine():
88.          The checkWine is used by the Recommendation View when
89.          the user wants to check out a wine listed in the
90.          activities. It takes the id of the wine selected
91.          and it gets passed to the activity "WineInfo" that
92.          will display the full info of the wine.
93.  */
94.
95.  public void checkWine(int id)
96.  {
97.      Intent intent = new Intent(recommendation_view_interface.getActivity(), WineInfoView.class);
98.      intent.putExtra("id", id);
99.      recommendation_view_interface.getActivity().startActivity(intent);

```

```

100.     }
101.
102.     /*
103.         addToCart():
104.         This method stores the wines info and the
105.         number of bottles the user would like to
106.         purchase. The info are stored into Shared Preferences
107.         and they will then be used in the cart activity.
108.     */
109.
110.     public void addToCart(Wine wine, int nbr)
111.     {
112.         SharedPreferences.Editor editor = sharedPref.edit();
113.         int index = sharedPref.getInt("winesNbr", 0);
114.         boolean wasPresent = false;
115.         for(int i = 0; i < index; i++)
116.         {
117.             if(sharedPref.getString("wineName"+(i+1), "Null").equals(wi
118. ne.getName()))
119.             {
120.                 wasPresent = true;
121.                 int number = sharedPref.getInt("wineNbr"+(i+1), 0);
122.                 editor.putInt("wineNbr"+(i+1), number+nbr);
123.                 float tot = sharedPref.getFloat("winesTOT", 0);
124.                 editor.putFloat("winesTOT", tot+(wine.getPrice()*nbr));
125.
126.                 editor.commit();
127.                 break;
128.             }
129.         }
130.         if(wasPresent == false)
131.         {
132.             editor.putInt("winesNbr", (index+1));
133.             editor.putInt("wineID"+(index+1), wine.getId());
134.             editor.putInt("wineNbr"+(index+1), nbr);
135.             editor.putString("wineName"+(index+1), wine.getName());
136.             editor.putFloat("winePrice"+(index+1), wine.getPrice());
137.             float tot = sharedPref.getFloat("winesTOT", 0);
138.             tot += wine.getPrice() * nbr;
139.             editor.putFloat("winesTOT", tot);
140.             editor.commit();
141.         }
142.     }

```

Recommendation Presenter Interface

```

1. package com.tastevin.android.tastevin.Recommendation_Presenter;
2.
3. public interface RecommendationPresenterInterface
4. {
5.     void getWinesList();
6.     void checkWine(int id);
7. }

```

Recommendation View

```

1. package com.tastevin.android.tastevin.Recommendation_View;
2.
3. import android.os.Bundle;
4. import android.support.annotation.NonNull;
5. import android.support.annotation.Nullable;
6. import android.support.v4.app.Fragment;
7. import android.view.LayoutInflater;
8. import android.view.View;
9. import android.view.ViewGroup;
10. import android.widget.ListView;
11. import android.widget.Toast;
12.
13.
14. import com.tastevin.android.tastevin.Model.WineAdapter;
15. import com.tastevin.android.tastevin.Model.Wine;
16. import com.tastevin.android.tastevin.R;
17. import com.tastevin.android.tastevin.Recommendation_Presenter.RecommendationPr
    esenter;
18. import com.tastevin.android.tastevin.Recommendation_Presenter.RecommendationPr
    esenterInterface;
19.
20. import java.util.ArrayList;
21.
22.     /*
23.      * View of the Recommendation Fragment.
24.      */
25.
26. public class RecommendationView extends Fragment implements RecommendationView
    Interface{
27.
28.     private ListView listview;
29.     private RecommendationPresenterInterface recommendation_presenter_interfac
    e;
30.     private WineAdapter adapter;
31.
32.     /*
33.      * onCreateView():
34.      * It created the layout and initialize the presenter
35.      * and the ListView.
36.      */
37.
38.     public View onCreateView(@NonNull LayoutInflater inflater, @Nullable ViewG
    roup container, @Nullable Bundle savedInstanceState)
39.     {
40.         View layout = inflater.inflate(R.layout.fragment_recommendation, conta
    iner, false);
41.         recommendation_presenter_interface = new RecommendationPresenter(this)
    ;
42.         listview = layout.findViewById(R.id.recommendation_list);
43.         return layout;
44.     }
45.
46.     /*
47.      * onViewCreated():
48.      * It asks the presenter to get the list of the wines.

```

```

49.    */
50.
51.    public void onResume()
52.    {
53.        super.onResume();
54.        recommendation_presenter_interface.getWinesList();
55.    }
56.
57.    public void onCreateView(@NonNull View view, @Nullable Bundle savedInstanceState)
58.    {
59.        super.onCreate(savedInstanceState);
60.        recommendation_presenter_interface.getWinesList();
61.    }
62.
63.    /**
64.     * displayToast():
65.     * This method displays a toast with a message.
66.     */
67.
68.    public void displayToast(String text)
69.    {
70.        Toast.makeText(getContext(), text, Toast.LENGTH_LONG).show();
71.    }
72.
73.    /**
74.     * showWineList():
75.     * This method displays the list of wines obtained
76.     * from the presenter.
77.     */
78.
79.    public void showWineList(ArrayList<Wine> wines)
80.    {
81.        adapter = new WineAdapter(getContext(), wines, "Recommendation");
82.        listview.setAdapter(adapter);
83.        adapter.setRecommendationPresenter((RecommendationPresenter) recommendation_presenter_interface);
84.    }
85. }

```

Recommendation View Interface

```

1.  package com.tastevin.android.tastevin.Recommendation_View;
2.
3.  import android.app.Activity;
4.
5.
6.  import com.tastevin.android.tastevin.Model.Wine;
7.
8.  import java.util.ArrayList;
9.
10. public interface RecommendationViewInterface
11. {
12.     Activity getActivity();
13.     void showWineList(ArrayList<Wine> wines);

```

```
14.     void displayToast(String text);
15. }
```

Shop Presenter

```
1.  package com.tastevin.android.tastevin.Shop_Presenter;
2.
3.  import android.content.Context;
4.  import android.content.SharedPreferences;
5.
6.  import com.android.volley.Response;
7.  import com.android.volley.VolleyError;
8.  import com.tastevin.android.tastevin.Model.ClientRequests;
9.  import com.tastevin.android.tastevin.Model.Filter;
10. import com.tastevin.android.tastevin.Model.Wine;
11. import com.tastevin.android.tastevin.Model.Parser;
12. import com.tastevin.android.tastevin.Shop_View.ShopViewInterface;
13.
14. import org.json.JSONException;
15.
16. import java.util.ArrayList;
17.
18.     /*
19.         Presenter of the Shop Fragment.
20.     */
21.
22. public class ShopPresenter implements ShopPresenterInterface
23. {
24.     private ShopViewInterface shop_view_interface;
25.     private ArrayList<Wine> wines, filteredWines;
26.     private ClientRequests parse;
27.     private SharedPreferences sharedPref;
28.
29.     /*
30.         Constructor of the Presenter:
31.         To be able to make changes to the view,
32.         it takes as argument an object of type
33.         ShopViewInterface.
34.         It also initialize the connection to the server
35.         and the Shared Preferences.
36.     */
37.
38.     public ShopPresenter(ShopViewInterface shop_view_interface)
39.     {
40.         this.shop_view_interface = shop_view_interface;
41.         parse = new ClientRequests(shop_view_interface.getActivity());
42.         sharedPref = shop_view_interface.getActivity().getSharedPreferences(
43.             "com.tastevin.android.tastevin.PREFERENCE_FILE_KEY", Context.M
44.             ODE_PRIVATE);
45.     }
46.
47.     /*
48.         getWinesList():
49.         This method sends a request to the server in
50.         order to get all the wines.
```



```

50.         If the connection has problems, the presenter will use
51.         the view method "displayToast" to create a Toast
52.         that alerts the user of the network error.
53.         If the connection was successful, the JSON Object is
54.         converted to ArrayList using the method "parseJSONWinesObject",
55.         then it's passed to the view in order to be displayed.
56.     */
57.
58.     public void getWinesList()
59.     {
60.         Response.Listener<String> listenerResponse = new Response.Listener<Str
ing>() {
61.
62.             @Override
63.             public void onResponse(String response)
64.             {
65.                 try {
66.                     wines = Parser.parseJSONWines(response);
67.                     shop_view_interface.showWineList(wines);
68.                 } catch (JSONException e) {
69.                     shop_view_interface.displayToast("Network Error");
70.                 }
71.             }
72.         };
73.         Response.ErrorListener errorListener = new Response.ErrorListener() {
74.
75.             @Override
76.             public void onErrorResponse(VolleyError error) {
77.                 shop_view_interface.displayToast("Network Error");
78.             }
79.         };
80.         parse.getWineList(listenerResponse, errorListener);
81.     }
82.     /*
83.     clearWinesList():
84.         This method is called when the view has to
85.         restore the original list of the shop.
86.     */
87.
88.     public void clearWinesList()
89.     {
90.         shop_view_interface.showWineList(wines);
91.     }
92.
93.     /*
94.     addToCart():
95.         This method stores the wines info and the
96.         number of bottles the user would like to
97.         purchase. The info are stored into Shared Preferences
98.         and they will then be used in the cart activity.
99.     */
100.
101.     public void addToCart(Wine wine, int nbr)
102.     {
103.         SharedPreferences.Editor editor = sharedPref.edit();
104.         int index = sharedPref.getInt("winesNbr", 0);

```




```

105.         boolean wasPresent = false;
106.         for(int i = 0; i < index; i++)
107.         {
108.             if(sharedPref.getString("wineName"+(i+1), "Null").equals(wi
ne.getName()))
109.             {
110.                 wasPresent = true;
111.                 int number = sharedPref.getInt("wineNbr"+(i+1), 0);
112.                 editor.putInt("wineNbr"+(i+1), number+nbr);
113.                 float tot = sharedPref.getFloat("winesTOT", 0);
114.                 editor.putFloat("winesTOT", tot+(wine.getPrice()*nbr));
115.                 editor.commit();
116.                 break;
117.             }
118.         }
119.         if(wasPresent == false)
120.         {
121.             editor.putInt("winesNbr", (index+1));
122.             editor.putInt("wineID"+(index+1), wine.getId());
123.             editor.putInt("wineNbr"+(index+1), nbr);
124.             editor.putString("wineName"+(index+1), wine.getName());
125.             editor.putFloat("winePrice"+(index+1), wine.getPrice());
126.             float tot = sharedPref.getFloat("winesTOT", 0);
127.             tot += wine.getPrice() * nbr;
128.             editor.putFloat("winesTOT", tot);
129.             editor.commit();
130.         }
131.     }
132.
133.     /*
134.     filter():
135.         This method takes the filter input from the user
136.         and used the method "filter" from the class Filter
137.         in order to return a filtered list of wines.
138.     */
139.
140.     public void filter(ArrayList<String> types, int[] years, int[] pric
es, int rating)
141.     {
142.         filteredWines = Filter.filter(wines, types, years, prices, rati
ng);
143.         shop_view_interface.showWineList(filteredWines);
144.     }
145.
146.     /*
147.     filterSearch():
148.         This method takes the search input from the user
149.         and used the method "filterSearch" from the class
150.         Filter in order to return a filtered list of wines.
151.     */
152.
153.     public void filterSearch(String query)
154.     {
155.         filteredWines = Filter.filterSearch(wines, query);
156.         shop_view_interface.showWineList(filteredWines);
157.     }

```

```
158.     }
```

Shop Presenter Interface

```
1. package com.tastevin.android.tastevin.Shop_Presenter;
2.
3. import com.tastevin.android.tastevin.Model.Wine;
4.
5. import java.util.ArrayList;
6.
7. public interface ShopPresenterInterface
8. {
9.     void getWinesList();
10.    void addToCart(Wine wine, int nbr);
11.    void filter(ArrayList<String> types, int[] years, int[] prices, int rating
12.    );
13.    void clearWinesList();
14.    void filterSearch(String query);
15. }
```

Shop View

```
1. package com.tastevin.android.tastevin.Shop_View;
2.
3. import android.app.AlertDialog;
4. import android.app.SearchManager;
5. import android.content.Context;
6. import android.content.DialogInterface;
7. import android.os.Bundle;
8. import android.support.annotation.NonNull;
9. import android.support.annotation.Nullable;
10. import android.support.v4.app.Fragment;
11. import android.view.LayoutInflater;
12. import android.view.Menu;
13. import android.view.MenuInflater;
14. import android.view.MenuItem;
15. import android.view.View;
16. import android.view.ViewGroup;
17. import android.widget.CheckBox;
18. import android.widget.ListView;
19. import android.widget.NumberPicker;
20. import android.widget.RatingBar;
21. import android.widget.SearchView;
22. import android.widget.Toast;
23.
24. import com.tastevin.android.tastevin.Model.WineAdapter;
25. import com.tastevin.android.tastevin.Model.Wine;
26. import com.tastevin.android.tastevin.R;
27. import com.tastevin.android.tastevin.Shop_Presenter.ShopPresenter;
28. import com.tastevin.android.tastevin.Shop_Presenter.ShopPresenterInterface;
29.
30. import java.util.ArrayList;
31.
32. /*
```

```

33.     View of the Shop Fragment.
34.     */
35.
36. public class ShopView extends Fragment implements ShopViewInterface
37. {
38.     private ShopPresenterInterface shop_presenter_interface;
39.     private ListView listView;
40.     private AlertDialog.Builder builder;
41.     private AlertDialog alertDialog;
42.     private WineAdapter adapter;
43.
44.     /*
45.         onCreate():
46.         It creates the activity.
47.     */
48.
49.     public void onCreate(Bundle savedInstanceState)
50.     {
51.         super.onCreate(savedInstanceState);
52.         setHasOptionsMenu(true);
53.     }
54.
55.     /*
56.         onCreateView():
57.         It creates the layout and initialize the presenter
58.         and the ListView.
59.     */
60.
61.     public View onCreateView(@NonNull LayoutInflater inflater, @Nullable ViewGroup
        roup container, @Nullable Bundle savedInstanceState) {
62.         View view = inflater.inflate(R.layout.fragment_shop, container, false)
63.         ;
64.         listView = view.findViewById(R.id.list);
65.         shop_presenter_interface = new ShopPresenter(this);
66.         shop_presenter_interface.getWinesList();
67.         return view;
68.     }
69.
70.     /*
71.         onCreateOptionsMenu():
72.         It sets the menu items and the search filter for the list
73.         of wines.
74.     */
75.
76.     public void onCreateOptionsMenu(Menu menu, MenuInflater inflater) {
77.         inflater.inflate(R.menu.search_menu, menu);
78.         final SearchManager searchManager = (SearchManager) getActivity().getS
79.         ystemService(Context.SEARCH_SERVICE);
80.         final SearchView searchView = (SearchView) menu.findItem(R.id.action_s
81.         earch).getActionView();
82.         if (null != searchView)
83.             searchView.setSearchableInfo(searchManager.getSearchableInfo(getAc
84.             tivity().getComponentName()));
85.
86.         SearchView.OnQueryTextListener queryTextListener = new SearchView.OnQu
87.         eryTextListener() {
88.             public boolean onQueryTextChange(String query) {

```



```
84.         shop_presenter_interface.filterSearch(query);
85.         return false;
86.     }
87.
88.     public boolean onQueryTextSubmit(String query) {
89.         return true;
90.     }
91. };
92. searchView.setOnQueryTextListener(queryTextListener);
93. super.onCreateOptionsMenu(menu, inflater);
94. }
95.
96. /*
97.     onOptionsItemSelected():
98.     It sets the onClick for the menu items. If the
99.     filter item is clicked, it will open an AlertDialog
100.    with the filtering options.
101. */
102.
103. public void onResume()
104. {
105.     super.onResume();
106.     shop_presenter_interface.getWinesList();
107. }
108.
109. public boolean onOptionsItemSelected(MenuItem item)
110. {
111.     switch (item.getItemId()) {
112.         case R.id.action_filter:
113.             buildFilterDialog();
114.             alertDialog = builder.create();
115.             alertDialog.show();
116.             return true;
117.         default:
118.             return super.onOptionsItemSelected(item);
119.     }
120. }
121.
122. /*
123.     showWineList():
124.     This method displays the list of wines obtained
125.     from the presenter.
126. */
127.
128. public void showWineList(ArrayList<Wine> wines)
129. {
130.     adapter = new WineAdapter(getActivity(), wines, "Shop");
131.     listView.setAdapter(adapter);
132.     adapter.setShopPresenter((ShopPresenter) shop_presenter_interfa
ce);
133. }
134.
135. /*
136.     displayToast():
137.     This method displays a toast with a message.
138. */
139.
```

```

140.         public void displayToast(String text)
141.         {
142.             Toast.makeText(getContext(), text, Toast.LENGTH_LONG).show();
143.         }
144.
145.         /*
146.             buildFilterDialog():
147.             This method builds the layout for the AlertDialog
148.             for the filter. It sets the options for the types
149.             (checkbox), price(number prickers), year(number
150.             prickers) and rating(rating bar).
151.         */
152.
153.         public void buildFilterDialog()
154.         {
155.             builder = new AlertDialog.Builder(getActivity());
156.             LayoutInflater inflater = getActivity().getLayoutInflater();
157.             final View theView = inflater.inflate(R.layout.fragment_filter,
158.                 null);
159.             final CheckBox checkBox_Red = theView.findViewById(R.id.checkBox_Type_1);
160.             final CheckBox checkBox_White = theView.findViewById(R.id.checkBox_Type_2);
161.             final CheckBox checkBox_Rosé = theView.findViewById(R.id.checkBox_Type_3);
162.             final CheckBox checkBox_Sparkling = theView.findViewById(R.id.checkBox_Type_4);
163.             final CheckBox checkBox_Dessert = theView.findViewById(R.id.checkBox_Type_5);
164.             final CheckBox checkBox_Year = theView.findViewById(R.id.checkBox_User_Year);
165.             final CheckBox checkBox_Price = theView.findViewById(R.id.checkBox_User_Price);
166.
167.             final NumberPicker min_Price = theView.findViewById(R.id.filter_Min_Price);
168.             final NumberPicker max_Price = theView.findViewById(R.id.filter_Max_Price);
169.             final NumberPicker min_Year = theView.findViewById(R.id.filter_Year_Min);
170.             final NumberPicker max_Year = theView.findViewById(R.id.filter_Year_Max);
171.
172.             min_Price.setMinValue(0);
173.             min_Price.setMaxValue(200);
174.             min_Price.setValue(0);
175.             max_Price.setMinValue(0);
176.             max_Price.setMaxValue(200);
177.             max_Price.setValue(200);
178.             min_Year.setMinValue(1900);
179.             min_Year.setMaxValue(2018);
180.             min_Year.setValue(2000);
181.             max_Year.setMinValue(1900);
182.             max_Year.setMaxValue(2018);
183.             max_Year.setValue(2018);
184.

```

```

185.         final RatingBar ratingBar = theView.findViewById(R.id.ratingBar
    _filter);
186.         builder.setView(theView);
187.         builder.setCancelable(false);
188.         builder.setPositiveButton("Ok", new DialogInterface.OnClickListener
ener() {
189.             @Override
190.             public void onClick(DialogInterface dialogInterface, int wh
ich) {
191.                 ArrayList<String> types = new ArrayList<>();
192.                 int[] years = new int[3];
193.                 int[] prices = new int[3];
194.                 prices[0] = 0;
195.                 years[0] = 0;
196.                 int rating = 0;
197.
198.                 if(checkBox_Red.isChecked())
199.                     types.add(checkBox_Red.getText().toString());
200.                 if(checkBox_White.isChecked())
201.                     types.add(checkBox_White.getText().toString());
202.                 if(checkBox_Rosé.isChecked())
203.                     types.add(checkBox_Rosé.getText().toString());
204.                 if(checkBox_Sparkling.isChecked())
205.                     types.add(checkBox_Sparkling.getText().toString());
206.
207.                 if(checkBox_Dessert.isChecked())
208.                     types.add(checkBox_Dessert.getText().toString());
209.
210.                 if(checkBox_Price.isChecked())
211.                 {
212.                     prices[0] = 1;
213.                     prices[1] = max_Price.getValue();
214.                     prices[2] = min_Price.getValue();
215.                 }
216.                 if(checkBox_Year.isChecked())
217.                 {
218.                     years[0] = 1;
219.                     years[1] = max_Year.getValue();
220.                     years[2] = min_Year.getValue();
221.                 }
222.                 if(ratingBar.getRating() != 0)
223.                     rating = (int) ratingBar.getRating();
224.                 shop_presenter_interface.filter(types, years, prices, r
ating);
225.             }
226.         });
227.         builder.setNegativeButton("Dismiss", new DialogInterface.OnClickListener
Listener() {
228.             @Override
229.             public void onClick(DialogInterface dialogInterface, int i)
{
230.                 dialogInterface.dismiss();
231.             }
232.         });
233.         builder.setNegativeButton("Clear all", new DialogInterface.OnCl
ickListener() {

```

```

234.             @Override
235.             public void onClick(DialogInterface dialogInterface, int wh
    ich) {
236.                 shop_presenter_interface.clearWinesList();
237.             }
238.         });
239.     }
240. }

```

Shop View Interface

```

1. package com.tastevin.android.tastevin.Shop_View;
2.
3. import android.app.Activity;
4.
5. import com.tastevin.android.tastevin.Model.Wine;
6.
7. import java.util.ArrayList;
8.
9. public interface ShopViewInterface
10. {
11.     Activity getActivity();
12.     void showWineList(ArrayList<Wine> wines);
13.     void displayToast(String text);
14.     void buildFilterDialog();
15. }

```

WineInfo Presenter

```

1. package com.tastevin.android.tastevin.WineInfo_Presenter;
2.
3. import android.content.Context;
4. import android.content.Intent;
5. import android.content.SharedPreferences;
6. import android.os.Bundle;
7. import android.support.v4.app.Fragment;
8. import android.view.View;
9.
10. import com.android.volley.Response;
11. import com.android.volley.VolleyError;
12. import com.tastevin.android.tastevin.Model.ClientRequests;
13. import com.tastevin.android.tastevin.Model.Ingredient;
14. import com.tastevin.android.tastevin.Model.Matching;
15. import com.tastevin.android.tastevin.Model.Wine;
16. import com.tastevin.android.tastevin.Model.Parser;
17. import com.tastevin.android.tastevin.WineInfo_View.SubWineInfo_View.SubWineInf
    oView;
18. import com.tastevin.android.tastevin.WineInfo_View.WineInfoViewInterface;
19.
20. import org.json.JSONException;
21. import org.json.JSONObject;
22.
23. import java.util.ArrayList;
24.

```

```

25.  /*
26.      Presenter of the WineInfo Activity.
27.  */
28.
29. public class WineInfoPresenter implements WineInfoPresenterInterface
30. {
31.     private Ingredient[] ingredients_1 = {new Ingredient(33, "Pork Chops"),
32.         new Ingredient(21, "Chicken Breast"),
33.         new Ingredient(41, "Venison Tenderloin"),
34.         new Ingredient(39, "Pot Roast")};
35.     private Ingredient[] ingredients_2 = {new Ingredient(9, "Tomato"),
36.         new Ingredient(10, "Potato"),
37.         new Ingredient(8, "Garlic"),
38.         new Ingredient(7, "Onion")};
39.     private Ingredient[] ingredients_3 = {new Ingredient(4, "Parmesan"),
40.         new Ingredient(13, "Bell Pepper"),
41.         new Ingredient(18, "Honey"),
42.         new Ingredient(42, "Apple Vinegar Cider"),
43.         new Ingredient(43, "Barbecue Sauce")};
44.     private Ingredient[] ingredients_4 = {new Ingredient(29, "Olive oil"),
45.         new Ingredient(34, "Soy Sauce"),
46.         new Ingredient(20, "Thyme"),
47.         new Ingredient(38, "Mushrooms Cream")};
48.     private WineInfoViewInterface wine_info_view_interface;
49.     private ClientRequests parse;
50.     private SharedPreferences sharedPref;
51.     private Wine wine;
52.
53.     /*
54.         Constructor():
55.         To be able to make changes to the view,
56.         it takes as argument an object of type
57.         WineInfoViewInterface.
58.         It also initializes the connection to the server
59.         and the Shared Preferences.
60.     */
61.
62.     public WineInfoPresenter(WineInfoViewInterface wine_info_view_interface)
63.     {
64.         this.wine_info_view_interface = wine_info_view_interface;
65.         parse = new ClientRequests(wine_info_view_interface.getActivity());
66.         sharedPref = wine_info_view_interface.getActivity().getSharedPreference
67. es(
68.         "com.tastevin.android.tastevin.PREFERENCE_FILE_KEY", Context.M
69.         ODE_PRIVATE);
70.     }
71.
72.     /*
73.         createIngredientsArray():
74.         This method is called by the view in order to create
75.         the multidimensional String array that contains the names of
76.         the ingredients.
77.     */
78.
79.     public void createIngredientsArray()
80.     {
81.         String[][] temp = new String[4][ingredients_1.length+1];

```



```

80.     temp[0] = new String[]{"Missing", ingredients_1[0].getName(), ingredie
nts_1[1].getName(), ingredients_1[2].getName(), ingredients_1[3].getName()};
81.     temp[1] = new String[]{"Missing", ingredients_2[0].getName(), ingredie
nts_2[1].getName(), ingredients_2[2].getName(), ingredients_2[3].getName()};
82.     temp[2] = new String[]{"Missing", ingredients_3[0].getName(), ingredie
nts_3[1].getName(), ingredients_3[2].getName(), ingredients_3[3].getName(), in
gredients_3[4].getName()};
83.     temp[3] = new String[]{"Missing", ingredients_4[0].getName(), ingredie
nts_4[1].getName(), ingredients_4[2].getName(), ingredients_4[3].getName()};
84.     wine_info_view_interface.createIngrArray(temp);
85. }
86.
87. /*
88.     setTitleAlertDialog():
89.         This method is called by the view in order get the
90.         name of the wine and set it as the title of a dialog.
91. */
92.
93. public void setTitleAlertDialog()
94. {
95.     wine_info_view_interface.setTitleAlert(wine.getName());
96. }
97.
98. /*
99.     addMatching():
100.         This method is called by the user in order to add a
101.         matching with the chosen ingredients. First, it takes the
102.         the ingredients that were selected. If no ingredient
103.         was chosen, it calls the view method "displayToast" to
104.         show a Toast to the user to tell him that he/she has
105.         to pick an ingredient. Then is create an object of type
106.         "Matching" and then it sends it to the server.
107.         If the connection has problems, the presenter will use
108.         the view method "displayToast" to create a Toast
109.         that alerts the user of the network error.
110.         If the connection was return Success, the request was
111.         successful, otherwise it will display an error.
112. */
113.
114. public void addMatching(ArrayList<Integer> position)
115. {
116.     ArrayList<Ingredient> ingredientMatching = new ArrayList<>();
117.     int wasSelected = 0;
118.
119.     for(int i = 0; i < position.size(); i++)
120.     {
121.         if(position.get(i) != 0)
122.         {
123.             switch (i)
124.             {
125.                 case 0:
126.                     ingredientMatching.add(ingredients_1[position.g
et(i)-1]);
127.                     wasSelected++;
128.                     break;
129.                 case 1:

```

```

130.                ingredientMatching.add(ingredients_2[position.g
    et(i)-1]);
131.                wasSelected++;
132.                break;
133.                case 2:
134.                ingredientMatching.add(ingredients_3[position.g
    et(i)-1]);
135.                wasSelected++;
136.                break;
137.                case 3:
138.                ingredientMatching.add(ingredients_4[position.g
    et(i)-1]);
139.                wasSelected++;
140.                break;
141.            }
142.        }
143.    }
144.    if(wasSelected == 0)
145.        wine_info_view_interface.displayToast("Select at least one
    ingredient");
146.    else
147.    {
148.        Response.Listener<JSONObject> listenerResponse = new Respon
    se.Listener<JSONObject>() {
149.            public void onResponse(JSONObject response)
150.            {
151.                if(response.toString().contains("Success"))
152.                    wine_info_view_interface.displayToast("Matching
    Added");
153.                else
154.                    wine_info_view_interface.displayToast("Network
    Error");
155.            }
156.        };
157.        Response.ErrorListener errorListener = new Response.ErrorLi
    stener() {
158.            @Override
159.            public void onErrorResponse(VolleyError error) {
160.                wine_info_view_interface.displayToast("Network Erro
    r");
161.            }
162.        };
163.        int userID = sharedPref.getInt("userID", 0);
164.        Matching matching = new Matching(wine.getId(), userID, ingr
    edientMatching);
165.        parse.insertMatching(matching, listenerResponse, errorListe
    ner);
166.    }
167.    }
168.
169.    /*
170.    setDialog():
171.    This method is called by the view in order get the
172.    wine object for the dialog.
173.    */
174.
175.    public void setDialog(View v)

```

```

176.         {
177.             wine_info_view_interface.setDialog(v, wine);
178.         }
179.
180.         /*
181.             getExtras():
182.             This method sends a request to the server
183.             in order to get the info of the wine with the
184.             id given by the extra of the intent.
185.             If the connection has problems, the presenter will use
186.             the view method "displayToast" to create a Toast
187.             that alerts the user of the network error.
188.             If the connection was successful, the JSON Object is
189.             converted to a Wine Object using the method "parseJSONWine"
190.             then two methods are the view are called in order to create
191.             the view and the first sub view of the fragment.
192.         */
193.
194.         public void getExtras(Intent intent)
195.         {
196.             Response.Listener<String> listenerResponse = new Response.Listener<String>() {
197.
198.                 @Override
199.                 public void onResponse(String response)
200.                 {
201.                     try {
202.                         wine = Parser.parseJSONWine(response);
203.                         wine_info_view_interface.createInfo(wine);
204.                         wine_info_view_interface.fragmentOne();
205.                     } catch (JSONException e) {
206.                         wine_info_view_interface.displayToast("Network Error");
207.                     }
208.                 }
209.             };
210.             Response.ErrorListener errorListener = new Response.ErrorListener() {
211.
212.                 @Override
213.                 public void onErrorResponse(VolleyError error) {
214.                     wine_info_view_interface.displayToast("Error Network");
215.                 }
216.             };
217.             int id = intent.getExtras().getInt("id");
218.             parse.getWineInfo(id, listenerResponse, errorListener);
219.         }
220.         /*
221.             addToCart():
222.             This method stores the wines info and the
223.             number of bottles the user would like to
224.             purchase. The info is stored into Shared Preferences
225.             and they will then be used in the cart activity.
226.         */

```



```
227.
228.     public void addToCart(Wine wine, int nbr)
229.     {
230.         SharedPreferences.Editor editor = sharedPref.edit();
231.         int index = sharedPref.getInt("winesNbr", 0);
232.         boolean wasPresent = false;
233.         for(int i = 0; i < index; i++)
234.         {
235.             if(sharedPref.getString("wineName"+(i+1), "Null").equals(wi
ne.getName()))
236.             {
237.                 wasPresent = true;
238.                 int number = sharedPref.getInt("wineNbr"+(i+1), 0);
239.                 editor.putInt("wineNbr"+(i+1), number+nbr);
240.                 float tot = sharedPref.getFloat("winesTOT", 0);
241.                 editor.putFloat("winesTOT", tot+(wine.getPrice()*nbr));
242.
243.                 editor.commit();
244.                 break;
245.             }
246.             if(wasPresent == false)
247.             {
248.                 editor.putInt("winesNbr", (index+1));
249.                 editor.putInt("wineID"+(index+1), wine.getId());
250.                 editor.putInt("wineNbr"+(index+1), nbr);
251.                 editor.putString("wineName"+(index+1), wine.getName());
252.                 editor.putFloat("winePrice"+(index+1), wine.getPrice());
253.                 float tot = sharedPref.getFloat("winesTOT", 0);
254.                 tot += wine.getPrice() * nbr;
255.                 editor.putFloat("winesTOT", tot);
256.                 editor.commit();
257.             }
258.         }
259.
260.         /*
261.         fragments():
262.         This method takes a fragment as argument
263.         and it's called when the user wants from pass
264.         from the SubWineInfoView to the WineReviewsView
265.         or the opposite. It sets a bundle with the information
266.         that these two fragments need in order to display
267.         the data.
268.         */
269.
270.         public void fragments(Fragment fragment)
271.         {
272.             if(fragment instanceof SubWineInfoView)
273.             {
274.                 Bundle bundle = new Bundle();
275.                 bundle.putString("winery", wine.getWinery());
276.                 bundle.putString("type", wine.getType());
277.                 bundle.putString("year", String.valueOf(wine.getYear()));
278.                 bundle.putString("composition", wine.getComposition());
279.                 bundle.putString("alcoholContent", String.valueOf(wine.getA
lcoholContent()));
280.                 fragment.setArguments(bundle);
```

```

281.         }
282.     else
283.     {
284.         Bundle bundle = new Bundle();
285.         bundle.putInt("id", wine.getId());
286.         bundle.putString("name", wine.getName());
287.         fragment.setArguments(bundle);
288.     }
289. }
290. }

```

WineInfo Presenter Interface

```

1. package com.tastevin.android.tastevin.WineInfo_Presenter;
2.
3. import android.content.Intent;
4. import android.support.v4.app.Fragment;
5. import android.view.View;
6.
7. import com.tastevin.android.tastevin.Model.Wine;
8.
9. import java.util.ArrayList;
10.
11. public interface WineInfoPresenterInterface
12. {
13.     void addMatching(ArrayList<Integer> ingredients);
14.     void getExtras(Intent intent);
15.     void fragments(Fragment fragment);
16.     void setTitleAlterDialog();
17.     void createIngredientsArray();
18.     void addToCart(Wine wine, int nbr);
19.     void setDialog(View v);
20. }

```

WineInfo View

```

1. package com.tastevin.android.tastevin.WineInfo_View;
2.
3. import android.app.Activity;
4. import android.app.AlertDialog;
5. import android.content.DialogInterface;
6. import android.os.Bundle;
7. import android.support.design.widget.BottomSheetDialog;
8. import android.support.v4.app.Fragment;
9. import android.support.v4.app.FragmentManager;
10. import android.support.v4.app.FragmentTransaction;
11. import android.support.v4.view.GravityCompat;
12. import android.support.v4.widget.DrawerLayout;
13. import android.support.v7.app.AppCompatActivity;
14. import android.view.LayoutInflater;
15. import android.view.Menu;
16. import android.view.MenuInflater;
17. import android.view.MenuItem;
18. import android.view.View;

```



```

19. import android.view.ViewGroup;
20. import android.widget.AdapterView;
21. import android.widget.Button;
22. import android.widget.ImageButton;
23. import android.widget.RatingBar;
24. import android.widget.Spinner;
25. import android.widget.TextView;
26. import android.widget.Toast;
27.
28. import com.tastevin.android.tastevin.Model.DashboardAdapter;
29. import com.tastevin.android.tastevin.Model.Wine;
30. import com.tastevin.android.tastevin.R;
31. import com.tastevin.android.tastevin.WineInfo_Presenter.WineInfoPresenter;
32. import com.tastevin.android.tastevin.WineInfo_Presenter.WineInfoPresenterInter
    face;
33. import com.tastevin.android.tastevin.WineInfo_View.SubWineInfo_View.SubWineInf
    oView;
34. import com.tastevin.android.tastevin.WineInfo_View.WineReviews_View.WineReview
    sView;
35.
36. import java.util.ArrayList;
37.
38.     /*
39.         View of the WineInfo Activity.
40.     */
41.
42. public class WineInfoView extends AppCompatActivity implements WineInfoViewInt
    erface
43. {
44.     private AlertDialog.Builder builder;
45.     private WineInfoPresenterInterface wineInfoPresenter;
46.     private String[] ingr1, ingr2, ingr3, ingr4;
47.     private BottomSheetDialog dialog;
48.     private Button buttonBack, buttonAccept;
49.     private TextView textViewNbrBottles;
50.
51.     /*
52.         onCreate():
53.         It creates the layout and initialize the presenter.
54.     */
55.
56.     protected void onCreate(Bundle savedInstanceState)
57.     {
58.         super.onCreate(savedInstanceState);
59.         setContentView(R.layout.activity_wine_info);
60.         wineInfoPresenter = new WineInfoPresenter((WineInfoViewInterface) getA
    ctivity());
61.         getExtras();
62.
63.         Button buttonPrice = findViewById(R.id.buttonPrice);
64.         buttonPrice.setOnClickListener(new View.OnClickListener() {
65.             @Override
66.             public void onClick(View v) {
67.                 wineInfoPresenter.setDialog(v);
68.             }
69.         });
70.         Button buttonFragmentOne = findViewById(R.id.button1);

```

```

71.         buttonFragmentOne.setOnClickListener(new View.OnClickListener() {
72.             @Override
73.             public void onClick(View v) {
74.                 fragmentOne();
75.             }
76.         });
77.         Button buttonFragmentTwo = findViewById(R.id.button2);
78.         buttonFragmentTwo.setOnClickListener(new View.OnClickListener() {
79.             @Override
80.             public void onClick(View v) {
81.                 fragmentTwo();
82.             }
83.         });
84.     }
85.
86.     /*
87.         createInfo():
88.         It sets the info of the wine.
89.     */
90.
91.     public void createInfo(Wine wine)
92.     {
93.         TextView textView = findViewById(R.id.textViewHead);
94.         textView.setText(wine.getName());
95.         RatingBar ratingBar = findViewById(R.id.ratingBar);
96.         ratingBar.setRating(wine.getRating());
97.
98.         Button button = findViewById(R.id.buttonPrice);
99.         button.setText("$ " + wine.getPrice());
100.    }
101.
102.    /*
103.        onCreateOptionsMenu():
104.        It sets the layout of the menu.
105.    */
106.
107.    public boolean onCreateOptionsMenu(Menu menu) {
108.        MenuInflater inflater = getMenuInflater();
109.        inflater.inflate(R.menu.wine_info, menu);
110.        return true;
111.    }
112.
113.    /*
114.        onOptionsItemSelected():
115.        It sets the actions to take when
116.        an item of the menu is clicked.
117.    */
118.
119.    public boolean onOptionsItemSelected(MenuItem item) {
120.        int id = item.getItemId();
121.
122.        if (id == R.id.action_add)
123.        {
124.            createMatching();
125.            return true;
126.        }
127.

```

```

128.         return super.onOptionsItemSelected(item);
129.     }
130.
131.     /*
132.         setDialog():
133.         It opens an AlertDialog when the user
134.         wants to insert the wine to the cart.
135.     */
136.
137.     public void setDialog(View v, Wine wine)
138.     {
139.         View finalConvertView = LayoutInflater.from(v.getContext()).inflate(R.layout.fragment_bottles_nbr_dialog, (ViewGroup) v.getParent(), false);
140.         buttonBack = finalConvertView.findViewById(R.id.buttonBack);
141.         buttonAccept = finalConvertView.findViewById(R.id.buttonAccept);
142.         ;
143.         ImageButton imageButtonAdd = finalConvertView.findViewById(R.id.imageButtonAdd);
144.         ImageButton imageButtonSubtract = finalConvertView.findViewById(R.id.imageButtonSubtract);
145.         TextView nbrBottles = finalConvertView.findViewById(R.id.textViewNbrBottles);
146.         ButtonClick buttonClick = new ButtonClick(wine);
147.         buttonAccept.setOnClickListener(buttonClick);
148.         buttonBack.setOnClickListener(buttonClick);
149.         imageButtonAdd.setOnClickListener(buttonClick);
150.         imageButtonSubtract.setOnClickListener(buttonClick);
151.
152.         dialog = new BottomSheetDialog(this);
153.         dialog.setContentView(finalConvertView);
154.         dialog.setTitle("Add Wine");
155.         dialog.show();
156.     }
157.
158.     /*
159.         fragmentOne():
160.         This method sets the base fragment to the
161.         subwineinfo.
162.     */
163.
164.     public void fragmentOne()
165.     {
166.         SubWineInfoView subWineInfoView = new SubWineInfoView();
167.         wineInfoPresenter.fragments(subWineInfoView);
168.         FragmentManager fragmentManager = getSupportFragmentManager();
169.         FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();
170.         fragmentTransaction.replace(R.id.fragment_switch, subWineInfoView);
171.         fragmentTransaction.commit();
172.     }
173.
174.     /*
175.         fragmentTwo():

```



```

176.             This method sets the base fragment to the
177.             winereviews.
178.         */
179.
180.         public void fragmentTwo()
181.         {
182.             Fragment fragmentTwo = new WineReviewsView();
183.             wineInfoPresenter.fragments(fragmentTwo);
184.             FragmentManager fragmentManager = getSupportFragmentManager();
185.             FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();
186.             fragmentTransaction.replace(R.id.fragment_switch, fragmentTwo);
187.             fragmentTransaction.commit();
188.         }
189.
190.         /*
191.             createMatching():
192.             This method creates an AlertDialog
193.             when the user wants to add a matching.
194.         */
195.
196.         public void createMatching()
197.         {
198.             builder = new AlertDialog.Builder(WineInfoView.this);
199.             LayoutInflater inflater = getLayoutInflater();
200.             final View theView = inflater.inflate(R.layout.fragment_add_mat
201. ching, null);
202.             wineInfoPresenter.setTitleAlertDialog();
203.             builder.setView(theView);
204.             builder.setCancelable(false);
205.             final Spinner spinner1 = theView.findViewById(R.id.button_ingre
206. dient_1);
207.             wineInfoPresenter.createIngredientsArray();
208.             ArrayAdapter<String> adapter1 = new ArrayAdapter<>(
209.                 this, R.layout.spinner_item, ingr1);
210.             final Spinner spinner2 = theView.findViewById(R.id.button_ingre
211. dient_2);
212.             ArrayAdapter<String> adapter2 = new ArrayAdapter<>(
213.                 this, R.layout.spinner_item, ingr2);
214.             final Spinner spinner3 = theView.findViewById(R.id.button_ingre
215. dient_3);
216.             ArrayAdapter<String> adapter3 = new ArrayAdapter<>(
217.                 this, R.layout.spinner_item, ingr3);
218.             final Spinner spinner4 = theView.findViewById(R.id.button_ingre
219. dient_4);
220.             ArrayAdapter<String> adapter4 = new ArrayAdapter<>(
221.                 this, R.layout.spinner_item, ingr4);
222.             adapter1.setDropDownViewResource(android.R.layout.simple_spinne
223. r_dropdown_item);
224.             spinner1.setAdapter(adapter1);
225.             adapter2.setDropDownViewResource(android.R.layout.simple_spinne
226. r_dropdown_item);

```

```

223.         spinner2.setAdapter(adapter2);
224.         adapter3.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
225.         spinner3.setAdapter(adapter3);
226.         adapter4.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
227.         spinner4.setAdapter(adapter4);
228.
229.         builder.setPositiveButton("Ok", new DialogInterface.OnClickListener() {
230.             @Override
231.             public void onClick(DialogInterface dialog, int which)
232.             {
233.                 ArrayList<Integer> ingredientsPos = new ArrayList<>();
234.                 ingredientsPos.add(spinner1.getSelectedItemPosition());
235.                 ingredientsPos.add(spinner2.getSelectedItemPosition());
236.                 ingredientsPos.add(spinner3.getSelectedItemPosition());
237.                 ingredientsPos.add(spinner4.getSelectedItemPosition());
238.                 wineInfoPresenter.addMatching(ingredientsPos);
239.             }
240.         });
241.         builder.setNegativeButton("Back", new DialogInterface.OnClickListener() {
242.             @Override
243.             public void onClick(DialogInterface dialog, int which) {
244.                 dialog.dismiss();
245.             }
246.         });
247.         AlertDialog alertDialog = builder.create();
248.         alertDialog.show();
249.     }
250.
251.     /**
252.      * setTitleAlert():
253.      * This method sets the title if the
254.      * AlertDialog
255.      */
256.
257.     public void setTitleAlert(String name)
258.     {
259.         builder.setTitle(name);
260.     }
261.
262.     /**
263.      * createIngrArray():
264.      * This method creates the string arrays
265.      * for the spinners that contains the
266.      * ingredients.
267.      */
268.
269.     public void createIngrArray(String[][] ingr)
270.     {

```

```

271.         ingr1 = ingr[0];
272.         ingr2 = ingr[1];
273.         ingr3 = ingr[2];
274.         ingr4 = ingr[3];
275.     }
276.
277.     /*
278.     displayToast():
279.     This method displays a toast with a message.
280.     */
281.
282.     public void displayToast(String text)
283.     {
284.         Toast.makeText(this, text, Toast.LENGTH_LONG).show();
285.     }
286.
287.     /*
288.     getActivity():
289.     This method returns the activity of wineInfo.
290.     */
291.
292.     public Activity getActivity()
293.     {
294.         return this;
295.     }
296.
297.     /*
298.     getActivity():
299.     This method passes the intent to the
300.     presenter.
301.     */
302.
303.     public void getExtras()
304.     {
305.         wineInfoPresenter.getExtras(getIntent());
306.     }
307.
308.     class ButtonClick implements View.OnClickListener
309.     {
310.         int nbr = 1;
311.         Wine wine;
312.
313.         public ButtonClick(Wine wine)
314.         {
315.             this.wine = wine;
316.         }
317.
318.         public void onClick(View v)
319.         {
320.             switch (v.getId())
321.             {
322.                 case R.id.buttonBack:
323.                     dialog.dismiss();
324.                     textViewNbrBottles.setText("1");
325.                     nbr = 1;
326.                     break;
327.                 case R.id.imageButtonSubtract:

```

```

328.         if(nbr > 1)
329.         {
330.             nbr = nbr -1;
331.             textViewNbrBottles.setText(String.valueOf(nbr))
332.         };
333.         break;
334.     case R.id.imageButtonAdd:
335.         if(nbr >= 1)
336.         {
337.             nbr = nbr + 1;
338.             textViewNbrBottles.setText(String.valueOf(nbr))
339.         };
340.         break;
341.     case R.id.buttonAccept:
342.         wineInfoPresenter.addToCart(wine, nbr);
343.         nbr = 1;
344.         textViewNbrBottles.setText(String.valueOf(nbr));
345.         dialog.dismiss();
346.         break;
347.     }
348. }
349. }
350. }

```

SubWineInfo Presenter

```

1. package com.tastevin.android.tastevin.WineInfo_Presenter.SubWineInfo_Presenter
2. ;
3. import android.os.Bundle;
4.
5. import com.tastevin.android.tastevin.WineInfo_View.SubWineInfo_View.SubWineInfo
6. oViewInterface;
7.
8. /*
9.  * Presenter of the SubWineInfo Fragment.
10. */
11. public class SubWineInfoPresenter implements SubWineInfoPresenterInterface
12. {
13.     private SubWineInfoViewInterface fragment_one_view_interface;
14.
15.     /*
16.      * Constructor():
17.      * To be able to make changes to the view,
18.      * it takes as argument an object of type
19.      * SubWineInfoViewInterface.
20.      */
21.
22.     public SubWineInfoPresenter(SubWineInfoViewInterface fragment_one_view_int
23.     erface)
24.     {
25.         this.fragment_one_view_interface = fragment_one_view_interface;

```

```

25.     }
26.
27.     /*
28.         getExtras():
29.             This method gets the info from the intent.
30.     */
31.
32.     public void getExtras(Bundle bundle)
33.     {
34.         String[] info = new String[5];
35.
36.         info[0] = (String) bundle.get("type");
37.         info[1] = (String) bundle.get("year");
38.         info[2] = (String) bundle.get("alcoholContent");
39.         info[3] = (String) bundle.get("composition");
40.         info[4] = (String) bundle.get("winery");
41.
42.         fragment_one_view_interface.setWineInfo(info);
43.     }
44.
45. }

```

SubWineInfo Presenter Interface

```

1. package com.tastevin.android.tastevin.WineInfo_Presenter.SubWineInfo_Presenter
   ;
2.
3. import android.os.Bundle;
4.
5. public interface SubWineInfoPresenterInterface {
6.
7.     void getExtras(Bundle bundle);
8. }

```

SubWineInfo View

```

1. package com.tastevin.android.tastevin.WineInfo_View.SubWineInfo_View;
2.
3. import android.os.Bundle;
4. import android.support.annotation.Nullable;
5. import android.support.v4.app.Fragment;
6. import android.view.LayoutInflater;
7. import android.view.View;
8. import android.view.ViewGroup;
9. import android.widget.TextView;
10.
11. import com.tastevin.android.tastevin.R;
12. import com.tastevin.android.tastevin.WineInfo_Presenter.SubWineInfo_Presenter.
    SubWineInfoPresenter;
13. import com.tastevin.android.tastevin.WineInfo_Presenter.SubWineInfo_Presenter.
    SubWineInfoPresenterInterface;
14.
15.     /*
16.         View of the SubWineInfo Fragment.

```

```
17.    */
18.
19. public class SubWineInfoView extends Fragment implements SubWineInfoViewInterface {
20.
21.     SubWineInfoPresenterInterface fragment_one_presenter_interface;
22.
23.     /*
24.         onCreateView():
25.         It creates the layout and initialize the presenter
26.         and the ListView.
27.     */
28.
29.     public View onCreateView(LayoutInflater inflater, ViewGroup container,
30.                             Bundle savedInstanceState) {
31.         fragment_one_presenter_interface = new SubWineInfoPresenter(this);
32.         return inflater.inflate(R.layout.fragment_one, container, false);
33.     }
34.
35.
36.     /*
37.         onViewCreated():
38.         It asks the presenter to get the other info of the wine.
39.     */
40.
41.     public void onViewCreated(View view, @Nullable Bundle savedInstanceState)
42.     {
43.         super.onViewCreated(view, savedInstanceState);
44.         fragment_one_presenter_interface.getExtras(getArguments());
45.     }
46.
47.     /*
48.         setWineInfo():
49.         It displays the info of the wine.
50.     */
51.
52.     public void setWineInfo(String[] info)
53.     {
54.         TextView textView = getView().findViewById(R.id.textView11);
55.         TextView textView2 = getView().findViewById(R.id.textView12);
56.         TextView textView3 = getView().findViewById(R.id.textView13);
57.         TextView textView4 = getView().findViewById(R.id.textView14);
58.         TextView textView5 = getView().findViewById(R.id.textView15);
59.
60.         textView.setText(info[4]);
61.         textView2.setText(info[0]);
62.         textView3.setText(info[1]);
63.         textView4.setText(info[3]);
64.         textView5.setText(info[2]);
65.     }
66. }
```

SubWine View Interface

```

1. package com.tastevin.android.tastevin.WineInfo_View.SubWineInfo_View;
2.
3. import android.app.Activity;
4.
5. public interface SubWineInfoViewInterface
6. {
7.     Activity getActivity();
8.     void setWineInfo(String[] info);
9. }
```

WineReviews Presenter

```

1. package com.tastevin.android.tastevin.WineInfo_Presenter.WineReviews_Presenter
2. ;
3. import android.content.Context;
4. import android.content.SharedPreferences;
5. import android.os.Bundle;
6.
7. import com.android.volley.Response;
8. import com.android.volley.VolleyError;
9. import com.tastevin.android.tastevin.Model.ClientRequests;
10. import com.tastevin.android.tastevin.Model.Parser;
11. import com.tastevin.android.tastevin.Model.Review;
12. import com.tastevin.android.tastevin.WineInfo_View.WineReviews_View.WineReview
    sViewInterface;
13.
14. import org.json.JSONException;
15. import org.json.JSONObject;
16.
17. import java.text.ParseException;
18. import java.util.ArrayList;
19.
20. /*
21.     Presenter of the WineReviewsPresenter Fragment.
22. */
23.
24. public class WineReviewsPresenter implements WineReviewsPresenterInterface
25. {
26.     private WineReviewsViewInterface fragment_wine_reviews_view_interface;
27.     private ClientRequests parse;
28.     private ArrayList<Object> reviews;
29.     private String name;
30.     private int id;
31.
32.     /*
33.         Constructor():
34.         To be able to make changes to the view,
35.         it takes as argument an object of type
36.         WineReviewsViewInterface.
37.         It also initialize the connection to the server.
38.     */
39. }
```

```

40.     public WineReviewsPresenter(WineReviewsViewInterface fragment_wine_reviews
    _view_interface)
41.     {
42.         this.fragment_wine_reviews_view_interface = fragment_wine_reviews_view
    _interface;
43.         parse = new ClientRequests(fragment_wine_reviews_view_interface.getAct
    ivity());
44.     }
45.
46.     /*
47.         setTitleAlterDialog():
48.         This method passes the name of wine to be set
49.         as title of an AlterDialog.
50.     */
51.
52.     public void setTitleAlterDialog()
53.     {
54.         fragment_wine_reviews_view_interface.setTitleAlert(name);
55.     }
56.
57.     /*
58.         addReview():
59.         This method sends the requests to the server
60.         in order to add a review.
61.     */
62.
63.     public void addReview(int rating, String comment)
64.     {
65.         if(comment.length() > 200)
66.             fragment_wine_reviews_view_interface.displayToast("Too many words"
    );
67.         else if(rating == 0)
68.             fragment_wine_reviews_view_interface.displayToast("Select a rating
    ");
69.         else
70.         {
71.             SharedPreferences sharedPref = fragment_wine_reviews_view_interfac
    e.getActivity().
72.             getSharedPreferences("com.tastevin.android.tastevin.PREFER
    ENCE_FILE_KEY",
73.                                 Context.MODE_PRIVATE);
74.             int userID = sharedPref.getInt("userID", 0);
75.             Review review = new Review(rating, comment, id, userID);
76.             Response.Listener<JSONObject> listenerResponse = new Response.List
    ener<JSONObject>() {
77.
78.                 public void onResponse(JSONObject response)
79.                 {
80.                     if(response.toString().contains("Success"))
81.                         fragment_wine_reviews_view_interface.displayToast("Rev
    iew added");
82.                     else
83.                         fragment_wine_reviews_view_interface.displayToast("Net
    work Error");
84.                 }
85.             };

```



```

86.         Response.ErrorListener errorListener = new Response.ErrorListener(
87.     ) {
88.         @Override
89.         public void onErrorResponse(VolleyError error) {
90.             fragment_wine_reviews_view_interface.displayToast("Network
Error");
91.         }
92.     };
93.     try {
94.         parse.insertRating(review, listenerResponse, errorListener);
95.     } catch (JSONException e) {
96.         fragment_wine_reviews_view_interface.displayToast("Network Err
or");
97.     }
98. }
99.
100.    /*
101.    getReviews():
102.    This method gets the reviews of the specific
103.    wine from the server.
104.    */
105.
106.    public void getReviews(Bundle bundle)
107.    {
108.        id = bundle.getInt("id");
109.        name = bundle.getString("name");
110.
111.        Response.Listener<String> listenerResponse = new Response.Listener<String>() {
112.            @Override
113.            public void onResponse(String response)
114.            {
115.                try {
116.                    reviews = Parser.parseJSONReviews(response);
117.                    fragment_wine_reviews_view_interface.setListReviews
(reviews);
118.                } catch (JSONException e) {
119.                    fragment_wine_reviews_view_interface.displayToast("
Network Error");
120.                } catch (ParseException e) {
121.                    fragment_wine_reviews_view_interface.displayToast("
Network Error");
122.                }
123.            }
124.        };
125.        Response.ErrorListener errorListener = new Response.ErrorListen
er() {
126.            @Override
127.            public void onErrorResponse(VolleyError error) {
128.                fragment_wine_reviews_view_interface.displayToast("Netw
ork Error");
129.            }
130.        };
131.        parse.getReviewList(id, listenerResponse, errorListener);
132.    }
133. }

```

WineReviews Presenter Interface

```

1. package com.tastevin.android.tastevin.WineInfo_Presenter.WineReviews_Presenter
   ;
2.
3. import android.os.Bundle;
4.
5. public interface WineReviewsPresenterInterface
6. {
7.     void getReviews(Bundle bundle);
8.     void setTitleAlterDialog();
9.     void addReview(int rating, String comment);
10. }
```

WineReviews View

```

1. package com.tastevin.android.tastevin.WineInfo_View.WineReviews_View;
2.
3. import android.app.AlertDialog;
4. import android.content.DialogInterface;
5. import android.os.Bundle;
6. import android.support.annotation.Nullable;
7. import android.support.v4.app.Fragment;
8. import android.view.LayoutInflater;
9. import android.view.View;
10. import android.view.ViewGroup;
11. import android.widget.Button;
12. import android.widget.EditText;
13. import android.widget.ListView;
14. import android.widget.RatingBar;
15. import android.widget.Toast;
16.
17. import com.tastevin.android.tastevin.Model.DashboardAdapter;
18. import com.tastevin.android.tastevin.R;
19. import com.tastevin.android.tastevin.WineInfo_Presenter.WineReviews_Presenter.
    WineReviewsPresenter;
20. import com.tastevin.android.tastevin.WineInfo_Presenter.WineReviews_Presenter.
    WineReviewsPresenterInterface;
21.
22. import java.util.ArrayList;
23.
24.     /*
25.      View of the WineReviews Fragment.
26.     */
27.
28. public class WineReviewsView extends Fragment implements WineReviewsViewInterf
    ace
29. {
30.     private ListView listView;
31.     private DashboardAdapter adapter;
32.     private WineReviewsPresenterInterface fragment_two_presenter_interface;
33.     private AlertDialog.Builder builder;
34.
35.     /*
36.      onCreateView():
```

```

37.         It creates the layout and initialize the presenter.
38.     */
39.
40.     public View onCreateView(LayoutInflater inflater, ViewGroup container,
41.                             Bundle savedInstanceState) {
42.
43.         View view = inflater.inflate(R.layout.fragment_two, container, false);
44.
45.         fragment_two_presenter_interface = new WineReviewsPresenter(this);
46.         return view;
47.     }
48.
49.     /*
50.     onViewCreated():
51.         It sets the list of reviews.
52.     */
53.
54.     public void onViewCreated(View view, @Nullable Bundle savedInstanceState)
55.     {
56.         super.onViewCreated(view, savedInstanceState);
57.
58.         listView = view.findViewById(R.id.recyclerViewReviews);
59.         fragment_two_presenter_interface.getReviews(getArguments());
60.
61.         Button button = view.findViewById(R.id.addReview);
62.         button.setOnClickListener(new View.OnClickListener() {
63.             @Override
64.             public void onClick(View v) {
65.                 buildAlertDialog();
66.             }
67.         });
68.     }
69.
70.     /*
71.     buildAlertDialog():
72.         It creates the AlertDialog to add a review.
73.     */
74.
75.     public void buildAlertDialog()
76.     {
77.         builder = new AlertDialog.Builder(getActivity());
78.         fragment_two_presenter_interface.setTitleAlterDialog();
79.         LayoutInflater inflater = getActivity().getLayoutInflater();
80.         final View theView = inflater.inflate(R.layout.fragment_add_review, nu
81.         11);
82.         builder.setView(theView);
83.         final RatingBar ratingBar = theView.findViewById(R.id.ratingBar_add);
84.         final EditText editText = theView.findViewById(R.id.editTextComment);
85.
86.         builder.setPositiveButton("Ok", new DialogInterface.OnClickListener()
87.         {
88.             @Override
89.             public void onClick(DialogInterface dialog, int which)

```



```

88.         {
89.             if(editText.getText().equals(null))
90.                 fragment_two_presenter_interface.addReview((int)ratingBar.
getRating(), "None");
91.             else
92.                 fragment_two_presenter_interface.addReview((int)ratingBar.
getRating(), String.valueOf(editText.getText()));
93.         }
94.     });
95.     builder.setNeutralButton("Back", new DialogInterface.OnClickListener()
{
96.         @Override
97.         public void onClick(DialogInterface dialog, int which) {
98.             dialog.dismiss();
99.         }
100.     });
101.     AlertDialog alertDialog = builder.create();
102.     alertDialog.show();
103. }
104.
105. /*
106.     setTitleAlert():
107.         It sets the title of the AlterDialog.
108. */
109.
110. public void setTitleAlert(String name)
111. {
112.     builder.setTitle(name);
113. }
114.
115. /*
116.     setTitleAlert():
117.         It initializes the adapter for the ListView.
118. */
119.
120. public void setListReviews(ArrayList<Object> reviews)
121. {
122.     adapter = new DashboardAdapter(getContext(), reviews);
123.     listView.setAdapter(adapter);
124. }
125.
126. /*
127.     displayToast():
128.         This method displays a toast with a message.
129. */
130.
131. public void displayToast(String text)
132. {
133.     Toast.makeText(getActivity(), text, Toast.LENGTH_LONG).show();
134. }
135. }

```

WineReviews View Interface

```

1. package com.tastevin.android.tastevin.WineInfo_View.WineReviews_View;
2.
3. import android.app.Activity;
4.
5. import java.util.ArrayList;
6.
7. public interface WineReviewsViewInterface {
8.     Activity getActivity();
9.     void setListReviews(ArrayList<Object> reviews);
10.    void displayToast(String text);
11.    void setTitleAlert(String name);
12. }
```

Activities

```

1. package com.tastevin.android.tastevin.Model;
2.
3. import java.util.Date;
4.
5.     /*
6.         Abstract class for the activities.
7.     */
8.
9. public abstract class Activities
10. {
11.     abstract Date getDate();
12.
13. }
```

CheckoutAdapter

```

1. package com.tastevin.android.tastevin.Model;
2.
3. import android.content.Context;
4. import android.support.annotation.NonNull;
5. import android.support.annotation.Nullable;
6. import android.view.LayoutInflater;
7. import android.view.View;
8. import android.view.ViewGroup;
9. import android.widget.ArrayAdapter;
10. import android.widget.TextView;
11.
12. import com.tastevin.android.tastevin.R;
13.
14. import java.util.ArrayList;
15.
16.     /*
17.         Adapter for the ListView of the Cart.
18.     */
19.
20. public class CheckoutAdapter extends ArrayAdapter<Wine> {
```

```

21.
22.     /*
23.         Constructor()
24.     */
25.
26.     public CheckoutAdapter(Context context, ArrayList<Wine> wines)
27.     {
28.         super(context, 0, wines);
29.     }
30.
31.     /*
32.         getView():
33.         This method returns the View that will
34.         be used for the ListView.
35.     */
36.
37.     public View getView(int position, @Nullable View convertView, @NonNull ViewGroup parent) {
38.
39.         Wine wine = getItem(position);
40.         if(convertView == null)
41.             convertView = LayoutInflater.from(getContext()).inflate(R.layout.list_item_checkout, parent, false);
42.
43.         TextView textView = convertView.findViewById(R.id.textViewHead2);
44.         TextView textView2 = convertView.findViewById(R.id.textViewH);
45.
46.         textView.setText(wine.getName());
47.         textView2.setText(String.valueOf(wine.getNbr()));
48.
49.         return convertView;
50.     }
51. }

```

ClientRequests

```

1. package com.tastevin.android.tastevin.Model;
2.
3. import android.content.Context;
4.
5. import com.android.volley.Request;
6. import com.android.volley.RequestQueue;
7. import com.android.volley.Response;
8. import com.android.volley.toolbox.JsonObjectRequest;
9. import com.android.volley.toolbox.StringRequest;
10. import com.android.volley.toolbox.Volley;
11.
12. import org.json.JSONArray;
13. import org.json.JSONException;
14. import org.json.JSONObject;
15.
16. import java.util.HashMap;
17. import java.util.Map;
18.
19.     /*

```



```
20.      Class for the HTTP Request of the client.
21.      */
22.
23. public class ClientRequests {
24.
25.     private RequestQueue queue;
26.
27.     /*
28.     Constructor()
29.     */
30.
31.     public ClientRequests(Context activity)
32.     {
33.         queue = Volley.newRequestQueue(activity);
34.     }
35.
36.     /*
37.     login():
38.     This method sends a requests to the server
39.     in order to verify the identity of the user and
40.     retrieve his/her information.
41.     */
42.
43.     public void login(String email, Response.Listener<String> listenerResponse
44. ,
45.                     Response.ErrorListener errorListener){
46.         final String[] headers = {"REQUEST", "LOGIN", "EMAIL", email};
47.         String url = "http://68.66.253.202:8080/";
48.         StringRequest stringRequest = new StringRequest(Request.Method.GET, ur
49. 1,
50.                     listenerResponse, errorListener)
51.         {
52.             public Map<String, String> getHeaders() {
53.                 Map<String, String> params = new HashMap<>();
54.                 params.put("Authorization", "Your authorization");
55.                 for(int i = 0; i < headers.length; i++)
56.                     params.put(headers[i], headers[++i]);
57.                 params.put("cache-control", "no-cache");
58.                 return params;
59.             }
60.         };
61.         queue.add(stringRequest);
62.
63.     /*
64.     getWineInfo():
65.     This method sends a requests to the server
66.     in order to get the information of the wine
67.     with the id given as parameter.
68.     */
69.
70.     public void getWineInfo(int id, Response.Listener<String> listenerResponse
71. ,
72.                     Response.ErrorListener errorListener){
73.         final String[] headers = {"REQUEST", "WINEINFO", "NBR", String.valueOf
(id));
```

```

73.         String url = "http://68.66.253.202:8080/";
74.         StringRequest stringRequest = new StringRequest(Request.Method.GET, url
1,
75.                                     listenerResponse, errorListener)
76.         {
77.             public Map<String, String> getHeaders() {
78.                 Map<String, String> params = new HashMap<>();
79.                 params.put("Authorization", "Your authorization");
80.                 for(int i = 0; i < headers.length; i++)
81.                     params.put(headers[i], headers[++i]);
82.                 params.put("cache-control", "no-cache");
83.                 return params;
84.             }
85.         };
86.
87.         queue.add(stringRequest);
88.     }
89.
90.     /*
91.     getWineList():
92.         This method sends a requests to the server
93.         in order to get the list of wines present
94.         in the shop.
95.     */
96.
97.     public void getWineList(Response.Listener<String> listenerResponse,
98.                             Response.ErrorListener errorListener)
99.     {
100.         final String[] headers = {"REQUEST", "WINELIST"};
101.         String url = "http://68.66.253.202:8080/";
102.         StringRequest stringRequest = new StringRequest(Request.Method.
GET, url,
103.                                                         listenerResponse, errorListen
er) {
104.             public Map<String, String> getHeaders()
105.             {
106.                 Map<String, String> params = new HashMap<>();
107.                 params.put("Authorization", "Your authorization");
108.                 for(int i = 0; i < headers.length; i++)
109.                     params.put(headers[i], headers[++i]);
110.                 params.put("cache-control", "no-cache");
111.                 return params;
112.             }
113.         };
114.         queue.add(stringRequest);
115.     }
116.
117.     /*
118.     getRecommendationList():
119.         This method sends a requests to the server
120.         in order to get the list of recommended wines
121.         for the user with the id given as parameter.
122.     */
123.
124.     public void getRecommendationList(int id, Response.Listener<String>
listenerResponse,

```



```

125.                                     Response.ErrorListener er
rorListener)
126.     {
127.         final String[] headers = {"REQUEST", "RECOMMENDATIONLIST", "NBR
", String.valueOf(id));
128.         String url = "http://68.66.253.202:8080/";
129.         StringRequest stringRequest = new StringRequest(Request.Method.
GET, url,
130.                                     listenerResponse, errorListen
er)
131.         {
132.             public Map<String, String> getHeaders()
133.             {
134.                 Map<String, String> params = new HashMap<>();
135.                 params.put("Authorization", "Your authorization");
136.                 for(int i = 0; i < headers.length; i++)
137.                     params.put(headers[i], headers[++i]);
138.                 params.put("cache-control", "no-cache");
139.                 return params;
140.             }
141.         };
142.         queue.add(stringRequest);
143.     }
144.
145.     /*
146.         getMatchedList():
147.         This method sends a requests to the server
148.         in order to get the list of wines that can be
149.         matched with the ingredients and the user's tastes
150.         given as parameters.
151.     */
152.
153.     public void getMatchedList(int ingr1, int ingr2, int ingr3, int ing
r4, int id,
154.                                Response.Listener<String> listenerRespon
se,
155.                                Response.ErrorListener errorListener)
156.     {
157.         final String[] headers = {"REQUEST", "MATCHEDLIST", "NBR1",
String.valueOf(ingr1), "NBR2",
158.                                String.valueOf(ingr2),
159.                                "NBR3", String.valueOf(ingr3),
"NBR4",
160.                                String.valueOf(ingr4), "NBR5",
String.valueOf(id));
161.
162.         String url = "http://68.66.253.202:8080/";
163.         StringRequest stringRequest = new StringRequest(Request.Method.
GET, url,
164.                                     listenerResponse, errorListen
er) {
165.             public Map<String, String> getHeaders()
166.             {
167.                 Map<String, String> params = new HashMap<>();
168.                 params.put("Authorization", "Your authorization");
169.                 for(int i = 0; i < headers.length; i++)
170.                     params.put(headers[i], headers[++i]);

```

```

171.         params.put("cache-control", "no-cache");
172.         return params;
173.     }
174. };
175.     queue.add(stringRequest);
176. }
177.
178. /*
179.     getReviewList():
180.         This method sends a requests to the server
181.         in order to get the list of reviews of the wine
182.         with the id given as parameter.
183. */
184.
185.     public void getReviewList(int id, Response.Listener<String> listene
rResponse,
186.                               Response.ErrorListener errorListe
ner)
187.     {
188.         final String[] headers = {"REQUEST", "REVIEWLIST", "NBR", Strin
g.valueOf(id)};
189.         String url = "http://68.66.253.202:8080/";
190.         StringRequest stringRequest = new StringRequest(Request.Method.
GET, url,
191.                                                         listenerResponse, errorListen
er) {
192.             public Map<String, String> getHeaders()
193.             {
194.                 Map<String, String> params = new HashMap<>();
195.                 params.put("Authorization", "Your authorization");
196.                 for(int i = 0; i < headers.length; i++)
197.                     params.put(headers[i], headers[++i]);
198.                 params.put("cache-control", "no-cache");
199.                 return params;
200.             }
201.         };
202.         queue.add(stringRequest);
203.     }
204.
205. /*
206.     getFriendList():
207.         This method sends a requests to the server
208.         in order to get the list of the latest
209.         activities of the user's friends.
210. */
211.
212.     public void getFriendList(int id, Response.Listener<String> listene
rResponse,
213.                               Response.ErrorListener errorListe
ner)
214.     {
215.         final String[] headers = {"REQUEST", "LISTFRIENDS", "NBR", Stri
ng.valueOf(id)};
216.         String url = "http://68.66.253.202:8080/";
217.         StringRequest stringRequest = new StringRequest(Request.Method.
GET, url,

```

```

218.                                     listenerResponse, errorListen
er)
219.     {
220.         public Map<String, String> getHeaders()
221.         {
222.             Map<String, String> params = new HashMap<>();
223.             params.put("Authorization", "Your authorization");
224.             for(int i = 0; i < headers.length; i++)
225.                 params.put(headers[i], headers[++i]);
226.             params.put("cache-control", "no-cache");
227.             return params;
228.         }
229.     };
230.     queue.add(stringRequest);
231. }
232.
233. /*
234.     insertRating():
235.     This method sends a requests to the server
236.     in order to insert a new rating in the
237.     database.
238. */
239.
240.     public void insertRating(Review review, Response.Listener<JSONObjec
t> listenerResponse,
241.                             Response.ErrorListener erro
rListener)
242.         throws JSONException
243.     {
244.         final String[] headers = {"REQUEST", "INSERTRATING"};
245.         String url = "http://68.66.253.202:8080/";
246.         JSONObject json = new JSONObject();
247.         json.put("text", review.getComment());
248.         json.put("rating", review.getRating());
249.         json.put("userID", review.getUserID());
250.         json.put("wineID", review.getWineID());
251.
252.         JsonObjectRequest jsonObjectRequest = new JsonObjectRequest(Req
uest.Method.POST, url, json,
253.                             listenerResponse, errorLi
stener)
254.         {
255.             public Map<String, String> getHeaders()
256.             {
257.                 Map<String, String> params = new HashMap<>();
258.                 params.put("Authorization", "Your authorization");
259.                 for(int i = 0; i < headers.length; i++)
260.                     params.put(headers[i], headers[++i]);
261.                 params.put("cache-control", "no-cache");
262.                 return params;
263.             }
264.         };
265.         queue.add(jsonObjectRequest);
266.     }
267.
268. /*
269.     insertMatching():

```

```

270.         This method sends a requests to the server
271.         in order to insert a new matching in the
272.         database.
273.     */
274.
275.     public void insertMatching(Matching matching, Response.Listener<JSONObject> listenerResponse,
276.                               Response.ErrorListener errorListener)
277.     {
278.         final String[] headers = {"REQUEST", "INSERTMATCHING"};
279.         String url = "http://68.66.253.202:8080/";
280.         JSONObject json = new JSONObject();
281.         try {
282.             json.put("userID", matching.getUserID());
283.             json.put("wineID", matching.getWineID());
284.
285.             JSONArray jsonArray = new JSONArray();
286.             for (int i=0; i < matching.getIngredientsArray().size(); i++) {
287.                 JSONObject object = new JSONObject();
288.                 object.put("id", matching.getIngredientsArray().get(i).getId());
289.                 object.put("name", matching.getIngredientsArray().get(i).getName());
290.                 jsonArray.put(object);
291.             }
292.
293.             json.put("ingredientsArray", jsonArray);
294.         } catch (JSONException e) {
295.             e.printStackTrace();
296.         }
297.
298.         JsonObjectRequest jsonObjectRequest = new JsonObjectRequest(Request.Method.POST, url, json,
299.                               listenerResponse, errorListener)
300.         {
301.             public Map<String, String> getHeaders()
302.             {
303.                 Map<String, String> params = new HashMap<>();
304.                 params.put("Authorization", "Your authorization");
305.                 for(int i = 0; i < headers.length; i++)
306.                     params.put(headers[i], headers[++i]);
307.                 params.put("cache-control", "no-cache");
308.                 return params;
309.             }
310.         };
311.         queue.add(jsonObjectRequest);
312.     }
313.
314.     /*
315.     insertPurchase():
316.     This method sends a requests to the server
317.     in order to insert a new purchase to the
318.     database.
319.     */

```

```

320.
321.     public void insertPurchase(Purchase purchase, Response.Listener<JSON
322.     NOBJECT> listenerResponse,
323.                                Response.ErrorListene
324.                                r errorListener)
325.     {
326.         final String[] headers = {"REQUEST", "INSERTPURCHASING"};
327.         String url = "http://68.66.253.202:8080/";
328.         JSONObject json = new JSONObject();
329.         try {
330.             json.put("userID", purchase.getUserID());
331.             JSONArray jsonArray = new JSONArray();
332.             for (int i=0; i < purchase.getWineArrayList().size(); i++)
333.             {
334.                 JSONObject object = new JSONObject();
335.                 object.put("id", purchase.getWineArrayList().get(i).get
336.                 Id());
337.                 jsonArray.put(object);
338.             }
339.             json.put("winesArray", jsonArray);
340.         } catch (JSONException e) {
341.             e.printStackTrace();
342.         }
343.         JsonObjectRequest jsonObjectRequest = new JsonObjectRequest(Request
344.         Method.POST, url, json,
345.                                listenerResponse, error
346.                                Listener)
347.         {
348.             public Map<String, String> getHeaders()
349.             {
350.                 Map<String, String> params = new HashMap<>();
351.                 params.put("Authorization", "Your authorization");
352.                 for(int i = 0; i < headers.length; i++)
353.                     params.put(headers[i], headers[++i]);
354.                 params.put("cache-control", "no-cache");
355.                 return params;
356.             }
357.         };
358.         queue.add(jsonObjectRequest);
359.     }

```

Config

```

1. package com.tastevin.android.tastevin.Model;
2.
3. public class Config
4. {
5.     public static final String PAYPAL_CLIENT_ID = "ATgqE0EjCsIEYJI45L-
6.     QeBegZ7PuXFZ3VpZDc-p1XU6oL41dldcOCRrFmS1G1zbf7RdDE3d7SEhGwIpn";
7. }

```

DashboardAdapter

```

1. package com.tastevin.android.tastevin.Model;
2.
3. import android.content.Context;
4. import android.content.Intent;
5. import android.support.design.widget.BottomSheetDialog;
6. import android.view.LayoutInflater;
7. import android.view.View;
8. import android.view.ViewGroup;
9. import android.widget.AdapterView;
10. import android.widget.Button;
11. import android.widget.ImageButton;
12. import android.widget.ImageView;
13. import android.widget.RatingBar;
14. import android.widget.TextView;
15.
16.
17. import com.tastevin.android.tastevin.R;
18. import com.tastevin.android.tastevin.Shop_Presenter.ShopPresenter;
19. import com.tastevin.android.tastevin.WineInfo_View.WineInfoView;
20.
21. import java.util.ArrayList;
22.
23.     /*
24.         Adapter for the ListView of the Dashboard.
25.     */
26.
27. public class DashboardAdapter extends ArrayAdapter<Object> {
28.
29.     /*
30.         Constructor()
31.     */
32.
33.     public DashboardAdapter(Context context, ArrayList<Object> activities)
34.     {
35.         super(context, 0, activities);
36.     }
37.
38.     /*
39.         getView():
40.         This methods returns the View that will
41.         be used for the ListView.
42.     */
43.
44.     public View getView(final int position, View convertView, ViewGroup parent
45.     )
46.     {
47.         if(getItem(position) instanceof Review)
48.         {
49.             convertView = LayoutInflater.from(parent.getContext()).inflate(R.l
ayout.list_item_reviews, parent, false);
50.             Review review = (Review) getItem(position);

```

```

50.
51.         TextView textView = convertView.findViewById(R.id.textViewNameWine
52.     );
53.         TextView textView2 = convertView.findViewById(R.id.textViewUsernam
54.     e);
55.         TextView textView3 = convertView.findViewById(R.id.textViewComment
56.     );
57.         RatingBar ratingBar = convertView.findViewById(R.id.ratingBar);
58.
59.         textView.setText(review.getUserName());
60.         textView2.setText(review.getWineName());
61.         textView3.setText(review.getComment());
62.         ratingBar.setRating(review.getRating());
63.     }
64.     else if(getItem(position) instanceof Purchase)
65.     {
66.         convertView = LayoutInflater.from(parent.getContext()).inflate(R.l
67.     ayout.list_item_purchases, parent, false);
68.         Purchase purchase = (Purchase) getItem(position);
69.
70.         TextView textView = convertView.findViewById(R.id.textViewWinename
71.     );
72.         TextView textView2 = convertView.findViewById(R.id.textView2ListWi
73.     nes);
74.
75.         textView.setText(purchase.getFirstName()+" "+purchase.getLastName(
76.     ));
77.         textView2.setText(purchase.getWines());
78.     }
79.     else if(getItem(position) instanceof Matching)
80.     {
81.         convertView = LayoutInflater.from(parent.getContext()).inflate(R.l
82.     ayout.list_item_matching, parent, false);
83.         Matching matching = (Matching) getItem(position);
84.
85.         TextView textView = convertView.findViewById(R.id.textViewUsername
86.     );
87.         TextView textView1 = convertView.findViewById(R.id.textViewWinenam
88.     e);
89.         TextView textView2 = convertView.findViewById(R.id.textViewIngredi
90.     ens);
91.
92.         textView.setText(matching.getFirstName()+" "+matching.getLastName(
93.     ));
94.         textView1.setText(matching.getWineName());
95.         textView2.setText(matching.getIngredients());
96.     }
97.     return convertView;
98. }
99. }

```

Filter

```

1. package com.tastevin.android.tastevin.Model;
2.

```

```

3. import java.util.ArrayList;
4. import java.util.Collections;
5. import java.util.Comparator;
6.
7.     /*
8.         Class for the filtering and sorting.
9.     */
10.
11. public class Filter {
12.
13.     /*
14.         filter():
15.         This method filters an ArrayList of wines
16.         based on the types, years, prices and ratings.
17.     */
18.
19.     public static ArrayList<Wine> filter(ArrayList<Wine> wines, ArrayList<String> types, int[] years, int[] prices, int rating)
20.     {
21.         ArrayList<Wine> filteredWines = new ArrayList<>(wines);
22.         Wine wine = null;
23.
24.         for(int k = 0; k < wines.size(); k++)
25.         {
26.             for(int y = 0; y < types.size(); y++)
27.             {
28.                 wine = wines.get(k);
29.                 if(wine.getType().equals(types.get(y)))
30.                     break;
31.                 else if(!(wine.getType().equals(types.get(y))) && y == types.size()-1)
32.                     filteredWines.remove(wines.get(k));
33.             }
34.             wine = wines.get(k);
35.             if(prices[0] == 1)
36.                 if (wine.getPrice() > prices[1] || wine.getPrice() < prices[2])
37.                     filteredWines.remove(wines.get(k));
38.             if(years[0] == 1)
39.                 if (wine.getYear() > years[1] || wine.getYear() < years[2])
40.                     filteredWines.remove(wines.get(k));
41.             if(rating != 0)
42.                 if (wine.getRating() != rating)
43.                     filteredWines.remove(wines.get(k));
44.         }
45.         return filteredWines;
46.     }
47.
48.     /*
49.         filterSearch():
50.         This method filters an ArrayList of wines
51.         based on the name.
52.     */
53.
54.     public static ArrayList<Wine> filterSearch(ArrayList<Wine> wines, String query)
55.     {

```




```

56.     ArrayList<Wine> filteredWines = new ArrayList<>(wines);
57.     Wine wine = null;
58.
59.     for(int k = 0; k < wines.size(); k++)
60.     {
61.         wine = wines.get(k);
62.         if(!(wine.getName().toLowerCase().substring(0,query.toCharArray().
length).equals(query.toLowerCase()))
63.             filteredWines.remove(wines.get(k));
64.     }
65.     return filteredWines;
66. }
67.
68.  /*
69.      sortActivities():
70.      This method sorts an ArrayList of objects
71.      that contains activities in a
72.      descending chronological order.
73.  */
74.
75.  public static ArrayList<Object> sortActivities(ArrayList<Object> activitie
s)
76.  {
77.      ArrayList<Activities> sortedActivities = new ArrayList<>();
78.      for(int i = 0; i < activities.size(); i++)
79.          sortedActivities.add((Activities) activities.get(i));
80.
81.      Collections.sort(sortedActivities, new Comparator<Activities>()
82.      {
83.          public int compare(Activities o1, Activities o2)
84.          {
85.              return o2.getDate().compareTo(o1.getDate());
86.          }
87.      });
88.      return new ArrayList<Object>(sortedActivities);
89.  }
90. }

```

Ingredient

```

1.  package com.tastevin.android.tastevin.Model;
2.
3.      /*
4.          Class for the Ingredient
5.      */
6.
7.  public class Ingredient {
8.
9.      private String name;
10.     private int id;
11.
12.     /*
13.         Constructor()
14.     */
15.

```

```

16.     public Ingredient(int id, String name) {
17.         this.name = name;
18.         this.id = id;
19.     }
20.
21.     /*
22.         Getters:
23.     */
24.
25.     public String getName() {
26.         return name;
27.     }
28.     public int getId() {
29.         return id;
30.     }
31.
32. }

```

Matching

```

1.  package com.tastevin.android.tastevin.Model;
2.
3.  import java.util.ArrayList;
4.  import java.util.Date;
5.
6.      /*
7.          Class for the Matching
8.      */
9.
10. public class Matching extends Activities {
11.
12.     private int matchID, wineID, userID;
13.     private ArrayList<Ingredient> ingredients;
14.     private String wineName, firstName, lastName;
15.     private Date date;
16.
17.     /*
18.         Constructor with full info()
19.     */
20.
21.     public Matching(int matchID, int wineID, int userID, String wineName,
22.                     String firstName, String lastName, Date date, ArrayList<In
23. gredient> ingredients) {
24.         this.matchID = matchID;
25.         this.wineID = wineID;
26.         this.userID = userID;
27.         this.ingredients = ingredients;
28.         this.wineName = wineName;
29.         this.firstName = firstName;
30.         this.lastName = lastName;
31.         this.date = date;
32.     }
33.
34.     /*
35.         Constructor with less info()

```

```

35.    */
36.
37.    public Matching(int wineID, int userID, ArrayList<Ingredient> ingredients)
38.    {
39.        this.wineID = wineID;
40.        this.userID = userID;
41.        this.ingredients = ingredients;
42.    }
43.    /*
44.    Getters:
45.    */
46.
47.    public ArrayList<Ingredient> getIngredientsArray()
48.    {
49.        return ingredients;
50.    }
51.    public int getMatchID() {
52.        return matchID;
53.    }
54.    public int getWineID() {
55.        return wineID;
56.    }
57.    public int getUserID() {
58.        return userID;
59.    }
60.    public Ingredient getIngredient(int id)
61.    {
62.        return ingredients.get(id);
63.    }
64.    public String getIngredients()
65.    {
66.        String temp = "";
67.        int i;
68.        for(i = 0; i < ingredients.size()-1; i++)
69.            temp += ingredients.get(i).getName()+" ";
70.        temp += ingredients.get(i).getName();
71.        return temp;
72.    }
73.    public String getWineName() {
74.        return wineName;
75.    }
76.    public String getFirstName() {
77.        return firstName;
78.    }
79.    public String getLastName() {
80.        return lastName;
81.    }
82.    public Date getDate() {
83.        return date;
84.    }
85.
86. }

```

Parser

```

1. package com.tastevin.android.tastevin.Model;
2.
3. import android.util.Log;
4.
5. import org.json.JSONArray;
6. import org.json.JSONException;
7. import org.json.JSONObject;
8.
9. import java.text.ParseException;
10. import java.text.SimpleDateFormat;
11. import java.util.ArrayList;
12. import java.util.Date;
13.
14.     /*
15.      * Class for the Parsing of JSONObject
16.      */
17.
18. public class Parser
19. {
20.     /*
21.      * parseJSONActivities():
22.      * This method is used for the parsing of JSONObject
23.      * that contains a list of Activities.
24.      */
25.
26.     public static ArrayList<Object> parseJSONActivities(String temp) throws JS
27.         ONException, ParseException
28.     {
29.         ArrayList<Object> activities = new ArrayList<>();
30.         ArrayList<Purchase> purchases = new ArrayList<>();
31.         ArrayList<Matching> matchings = new ArrayList<>();
32.         ArrayList<Review> reviews = new ArrayList<>();
33.         JSONObject object = new JSONObject(temp);
34.         JSONArray jsonArray = (JSONArray) object.getJSONArray("reviews").get(0
35.     );
36.
37.         for(int i = 0; i < jsonArray.length(); i++)
38.         {
39.             JSONObject object1 = jsonArray.getJSONObject(i);
40.             String dateStr = object1.getString("date");
41.             SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
42.             Date date = sdf.parse(dateStr);
43.             reviews.add(new Review(object1.getInt("id"), object1.getInt("rating"), object1.getString("comment"), object1.getString("userName"), object1.getString("wineName"), object1.getInt("wineID"), object1.getInt("userID"), date));
44.         }
45.         JSONArray jsonArray2 = (JSONArray) object.getJSONArray("purchases").get(0);
46.
47.         for(int i = 0; i < jsonArray2.length(); i++)
48.         {
49.             JSONObject object1 = jsonArray2.getJSONObject(i);
50.             String dateStr = object1.getString("date");
51.             SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
52.             Date date = sdf.parse(dateStr);

```



```

51.         ArrayList<Wine> wines = new ArrayList<>();
52.         JSONArray array = object1.getJSONArray("wines");
53.         for(int j = 0; j < array.length(); j++)
54.         {
55.             JSONObject win = array.getJSONObject(j);
56.             wines.add(new Wine(win.getInt("id"), win.getString("name")));
57.         }
58.         purchases.add(new Purchase(object1.getInt("purchaseID"), object1.g
etInt("userID"), object1.getString("firstName"), object1.getString("lastName")
,
59.             wines, object1.getInt("price"), date));
60.     }
61.     JSONArray jsonArray3 = (JSONArray) object.getJSONArray("matching").get
(0);
62.     for(int i = 0; i < jsonArray3.length(); i++)
63.     {
64.         JSONObject object1 = jsonArray3.getJSONObject(i);
65.         String dateStr = object1.getString("date");
66.         SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
67.         Date date = sdf.parse(dateStr);
68.         ArrayList<Ingredient> ingredients = new ArrayList<>();
69.         JSONArray array = object1.getJSONArray("ingredientsArray");
70.         for(int j = 0; j < array.length(); j++)
71.         {
72.             JSONObject ingr = array.getJSONObject(j);
73.             ingredients.add(new Ingredient(ingr.getInt("id"), ingr.getStri
ng("name")));
74.         }
75.         matchings.add(new Matching(object1.getInt("matchID"), object1.getI
nt("wineID"), object1.getInt("userID"), object1.getString("wineName"),
76.             object1.getString("firstName"), object1.getString("lastNam
e"), date, ingredients));
77.     }
78.     activities.addAll(purchases);
79.     activities.addAll(matchings);
80.     activities.addAll(reviews);
81.
82.     return activities;
83. }
84.
85. /*
86.     parseJSONWines():
87.         This method is used for the parsing of JSONObject
88.         that contains a list of wines.
89. */
90.
91. public static ArrayList<Wine> parseJSONWines(String temp) throws JSONExcep
tion
92. {
93.     ArrayList<Wine> wines = new ArrayList<>();
94.     JSONObject object = new JSONObject(temp);
95.     JSONArray jsonArray = (JSONArray) object.getJSONArray("wines").get(0);
96.     for(int i = 0; i < jsonArray.length(); i++) {
97.         JSONObject object1 = jsonArray.getJSONObject(i);

```



```

98.         wines.add(new Wine(object1.getInt("id"), object1.getString("name")
, object1.getString("winery"),
99.             object1.getString("type"), Float.parseFloat(object1.getStr
ing("price")),
100.             Float.parseFloat(object1.getString("alcoholContent"
)), object1.getInt("rating"),
101.             object1.getInt("year"), object1.getInt("reviewsNbr"
), object1.getString("composition")));
102.     }
103.     return wines;
104. }
105.
106. /*
107.     parseJSONWines():
108.         This method is used for the parsing of JSONObject
109.         that contains the info of a wine.
110. */
111.
112.     public static Wine parseJSONWine(String temp) throws JSONException
113.     {
114.         JSONObject object = new JSONObject(temp);
115.         Wine wine = new Wine(object.getInt("id"), object.getString("nam
e"), object.getString("winery"),
116.             object.getString("type"), Float.parseFloat(object.getSt
ring("price")),
117.             Float.parseFloat(object.getString("alcoholContent")), o
bject.getInt("rating"),
118.             object.getInt("year"), object.getInt("reviewsNbr"), obj
ect.getString("composition"));
119.         return wine;
120.     }
121.
122. /*
123.     parseJSONUser():
124.         This method is used for the parsing of JSONObject
125.         that contains the info of a user.
126. */
127.
128.     public static User parseJSONUser(String temp, String mail) throws J
SONException, ParseException {
129.         JSONObject object = new JSONObject(temp);
130.         String dateStr = object.getString("birthday");
131.         SimpleDateFormat sdf = new SimpleDateFormat("yyyyMMdd");
132.         Date parsed = sdf.parse(dateStr);
133.         java.sql.Date date = new java.sql.Date(parsed.getTime());
134.         return new User(object.getInt("id"), object.getString("firstNam
e"), object.getString("lastName"),
135.             mail, date, object.getString("gender").charAt(0), objec
t.getInt("sweet"),
136.             object.getInt("sour"), object.getInt("bitter"), object.
getInt("salty"), object.getInt("umami"));
137.     }
138.
139. /*
140.     parseJSONReviews():
141.         This method is used for the parsing of JSONObject

```



```
142.         that contains the info of a reviews.
143.     */
144.
145.     public static ArrayList<Object> parseJSONReviews(String temp) throw
146.     s JSONException, ParseException
147.     {
148.         ArrayList<Object> reviews = new ArrayList<>();
149.         JSONObject object = new JSONObject(temp);
150.         JSONArray jsonArray = (JSONArray) object.getJSONArray("reviews"
151.         ).get(0);
152.         for(int i = 0; i < jsonArray.length(); i++) {
153.             JSONObject object1 = jsonArray.getJSONObject(i);
154.             String dateStr = object1.getString("date");
155.             SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-
156.             dd");
157.             Date date = sdf.parse(dateStr);
158.             Log.e("HERE", ""+object1.getString("userName"));
159.             reviews.add(new Review(object1.getInt("id"), object1.getInt
160.             ("rating"), object1.getString("comment"), object1.getString("userName"),
161.             object1.getString("wineName"), object1.getInt("wine
162.             ID"), object1.getInt("userID"), date));
163.         }
164.         return reviews;
165.     }
166. }
```

Purchase

```
1. package com.tastevin.android.tastevin.Model;
2.
3. import java.util.ArrayList;
4. import java.util.Date;
5.
6.     /*
7.     Class for the Ingredient
8.     */
9.
10. public class Purchase extends Activities {
11.
12.     private int purchaseID, userID;
13.     private float price;
14.     private ArrayList<Wine> wines;
15.     private String firstName, lastName;
16.     private Date date;
17.
18.     /*
19.     Constructor with full info()
20.     */
21.
22.     public Purchase(int purchaseID, int userID, String firstName, String lastN
23.     ame, ArrayList<Wine> wines, int price, Date date) {
24.         this.purchaseID = purchaseID;
25.         this.userID = userID;
26.         this.firstName = firstName;
27.         this.lastName = lastName;
```



```

27.         this.date = date;
28.         this.wines = wines;
29.         this.price = price;
30.     }
31.
32.     /*
33.      Constructor with less info()
34.     */
35.
36.     public Purchase(int userID)
37.     {
38.         this.userID = userID;
39.         wines = new ArrayList<>();
40.     }
41.
42.     /*
43.      Getters:
44.     */
45.
46.     public int getPurchaseID() {
47.         return purchaseID;
48.     }
49.
50.     public int getWinesSize()
51.     {
52.         return wines.size();
53.     }
54.
55.     public Wine getWine(int id)
56.     {
57.         return wines.get(id);
58.     }
59.
60.     public int getUserID() {
61.         return userID;
62.     }
63.
64.     public float getPrice()
65.     {
66.         return price;
67.     }
68.
69.     public String getFirstName() {
70.         return firstName;
71.     }
72.
73.     public String getLastName() {
74.         return lastName;
75.     }
76.
77.     public Date getDate() {
78.         return date;
79.     }
80.
81.     public ArrayList<String> getWineList() {
82.         ArrayList<String> wines_string = new ArrayList<>();
83.         for(int i = 0; i < wines.size(); i++)

```




```

84.         wines_string.add(wines.get(i).getName());
85.     return wines_string;
86. }
87.
88. public ArrayList<Wine> getWineArrayList()
89. {
90.     return wines;
91. }
92.
93. public String getWines() {
94.     String temp = "";
95.     int i;
96.     for(i = 0; i < wines.size()-1; i++)
97.         temp += wines.get(i).getName()+" ";
98.     temp += wines.get(i).getName();
99.     return temp;
100. }
101.
102. /*
103.     Setters:
104. */
105.
106. public void addWine(Wine wine)
107. {
108.     wines.add(wine);
109. }
110. }

```

Review

```

1. package com.tastevin.android.tastevin.Model;
2.
3. import java.util.Date;
4.
5. /*
6.     Class for the Review
7. */
8.
9. public class Review extends Activities {
10.
11.     private int id, rating, wineID, userID;
12.     private String userName, wineName, comment;
13.     private Date date;
14.
15.     /*
16.     Constructor()
17.     */
18.
19.     public Review(int id, int rating, String comment, String userName, String wineName, int wineID, int userID, Date date)
20.     {
21.         this.id = id;
22.         this.rating = rating;
23.         this.userName = userName;
24.         this.wineName = wineName;

```



```

25.         this.date = date;
26.         this.wineID = wineID;
27.         this.comment = comment;
28.         this.userID = userID;
29.     }
30.
31.     /*
32.     Constructor with less info()
33.     */
34.     public Review(int rating, String comment, int wineID, int userID)
35.     {
36.         this.rating = rating;
37.         this.wineID = wineID;
38.         this.comment = comment;
39.         this.userID = userID;
40.     }
41.
42.     /*
43.     Getters:
44.     */
45.
46.     public String getComment()
47.     {
48.         return comment;
49.     }
50.
51.     public int getId()
52.     {
53.         return id;
54.     }
55.
56.     public int getRating()
57.     {
58.         return rating;
59.     }
60.
61.     public String getUserName()
62.     {
63.         return userName;
64.     }
65.
66.     public String getWineName()
67.     {
68.         return wineName;
69.     }
70.
71.     public Date getDate()
72.     {
73.         return date;
74.     }
75.
76.     public int getWineID() {
77.         return wineID;
78.     }
79.
80.     public int getUserID() {return userID; }
81. }

```

User

```

1. package com.tastevin.android.tastevin.Model;
2.
3. import java.util.Date;
4.
5.     /*
6.         Class for the User
7.     */
8.
9. public class User{
10.
11.     private String firstName, lastName, email;
12.     private int id, sweet, sour, bitter, salty, umami;
13.     private Date birthday;
14.     private char gender;
15.
16.     /*
17.         Constructor()
18.     */
19.
20.     public User(int id, String firstName, String lastName, String email, Date
        birthday,
21.                 char gender, int sweet, int sour, int bitter, int salty, int u
        mami)
22.     {
23.         this.id = id;
24.         this.firstName = firstName;
25.         this.lastName = lastName;
26.         this.email = email;
27.         this.birthday = birthday;
28.         this.gender = gender;
29.         this.sweet = sweet;
30.         this.sour = sour;
31.         this.bitter = bitter;
32.         this.salty = salty;
33.         this.umami = umami;
34.     }
35.
36.     /*
37.         Getters:
38.     */
39.
40.     public String getEmail()
41.     {
42.         return email;
43.     }
44.
45.     public int getSweet()
46.     {
47.         return sweet;
48.     }
49.
50.     public int getSour()
51.     {
52.         return sour;
53.     }

```

```

54.
55.     public int getBitter()
56.     {
57.         return bitter;
58.     }
59.
60.     public int getSalty()
61.     {
62.         return salty;
63.     }
64.
65.     public int getUmami()
66.     {
67.         return umami;
68.     }
69.
70.     public String getBirthday()
71.     {
72.         return birthday.toString();
73.     }
74.
75.     public char getGender()
76.     {
77.         return gender;
78.     }
79.
80.     public String getFirstName() {
81.
82.         return firstName;
83.     }
84.
85.     public String getLastName()
86.     {
87.         return lastName;
88.     }
89.
90.     public int getId()
91.     {
92.         return id;
93.     }
94. }

```

Wine

```

1. package com.tastevin.android.tastevin.Model;
2.
3. import android.os.Parcel;
4. import android.os.Parcelable;
5.
6.     /*
7.         Class for the Wine
8.     */
9.
10. public class Wine implements Parcelable {
11.

```



```

12.     private int id, year, rating, reviewsNbr;
13.     private String name, winery, type, composition;
14.     private float price, alcoholContent;
15.     private int nbr;
16.
17.     /*
18.      * Constructor()
19.      */
20.
21.     public Wine(int id, String name, String winery, String type, float price,
22.                float alcoholContent, int rating, int year, int reviewsNbr, String composition)
23.     {
24.         this.id = id;
25.         this.name = name;
26.         this.winery = winery;
27.         this.type = type;
28.         this.price = price;
29.         this.alcoholContent = alcoholContent;
30.         this.composition = composition;
31.         this.rating = rating;
32.         this.year = year;
33.         this.reviewsNbr = reviewsNbr;
34.     }
35.
36.     /*
37.      * Constructors with less info ()
38.      */
39.
40.     public Wine(int id, String name, int nbr)
41.     {
42.         this.id = id;
43.         this.name = name;
44.         this.nbr = nbr;
45.     }
46.
47.     public Wine(int id, String name)
48.     {
49.         this.id = id;
50.         this.name = name;
51.     }
52.
53.     /*
54.      * Parcelable()
55.      */
56.
57.     protected Wine(Parcel in) {
58.         id = in.readInt();
59.         year = in.readInt();
60.         rating = in.readInt();
61.         reviewsNbr = in.readInt();
62.         name = in.readString();
63.         winery = in.readString();
64.         type = in.readString();
65.         composition = in.readString();
66.         price = in.readFloat();

```



```

67.         alcoholContent = in.readFloat();
68.         nbr = in.readInt();
69.     }
70.
71.     public static final Creator<Wine> CREATOR = new Creator<Wine>() {
72.         @Override
73.         public Wine createFromParcel(Parcel in) {
74.             return new Wine(in);
75.         }
76.
77.         @Override
78.         public Wine[] newArray(int size) {
79.             return new Wine[size];
80.         }
81.     };
82.
83.     /*
84.     Getters:
85.     */
86.
87.     public int getId()
88.     {
89.         return id;
90.     }
91.
92.     public int getYear()
93.     {
94.         return year;
95.     }
96.
97.     public String getName()
98.     {
99.         return name;
100.    }
101.
102.    public String getWinery()
103.    {
104.        return winery;
105.    }
106.
107.    public String getType()
108.    {
109.        return type;
110.    }
111.
112.    public float getPrice()
113.    {
114.        return price;
115.    }
116.
117.    public float getAlcoholContent()
118.    {
119.        return alcoholContent;
120.    }
121.
122.    public String getComposition()
123.    {

```

```

124.         return composition;
125.     }
126.
127.     public int getRating()
128.     {
129.         return rating;
130.     }
131.
132.     public int getReviewsNbr() {
133.         return reviewsNbr;
134.     }
135.
136.     public int getNbr()
137.     {
138.         return nbr;
139.     }
140.
141.     @Override
142.     public int describeContents() {
143.         return 0;
144.     }
145.
146.     @Override
147.     public void writeToParcel(Parcel dest, int flags) {
148.         dest.writeInt(id);
149.         dest.writeInt(year);
150.         dest.writeInt(rating);
151.         dest.writeInt(reviewsNbr);
152.         dest.writeString(name);
153.         dest.writeString(winery);
154.         dest.writeString(type);
155.         dest.writeString(composition);
156.         dest.writeFloat(price);
157.         dest.writeFloat(alcoholContent);
158.         dest.writeInt(nbr);
159.     }
160. }
```

WineAdapter

```

1. package com.tastevin.android.tastevin.Model;
2.
3. import android.content.Context;
4. import android.content.Intent;
5. import android.support.annotation.NonNull;
6. import android.support.annotation.Nullable;
7. import android.support.design.widget.BottomSheetDialog;
8. import android.view.LayoutInflater;
9. import android.view.View;
10. import android.view.ViewGroup;
11. import android.widget.AdapterView;
12. import android.widget.Button;
13. import android.widget.ImageButton;
14. import android.widget.ImageView;
15. import android.widget.RatingBar;
```

```

16. import android.widget.TextView;
17. import android.widget.Toast;
18.
19. import com.tastevin.android.tastevin.R;
20. import com.tastevin.android.tastevin.Recommendation_Presenter.RecommendationPr
    esenter;
21. import com.tastevin.android.tastevin.Recommendation_View.RecommendationView;
22. import com.tastevin.android.tastevin.Shop_Presenter.ShopPresenter;
23. import com.tastevin.android.tastevin.WineInfo_View.WineInfoView;
24.
25. import java.util.ArrayList;
26.
27.     /*
28.         Adapter for the ListView of the Recommendations and Shop.
29.     */
30.
31. public class WineAdapter extends ArrayAdapter<Wine>
32. {
33.     private RecommendationPresenter recommendationPresenter;
34.     private ShopPresenter shopPresenter;
35.     private BottomSheetDialog dialog;
36.     private Button backButton;
37.     private Button buttonAccept;
38.     private TextView textViewNbrBottles;
39.     private String id;
40.
41.     /*
42.         Constructor()
43.     */
44.
45.     public WineAdapter(Context context, ArrayList<Wine> wines, String id)
46.     {
47.         super(context, 0, wines);
48.         this.id = id;
49.     }
50.
51.     /*
52.         getView():
53.             This method returns the View that will
54.             be used for the ListView.
55.     */
56.
57.     public View getView(int position, @Nullable View convertView, @NonNull Vie
        wGroup parent) {
58.
59.         final Wine wine = getItem(position);
60.         if(convertView == null)
61.             convertView = LayoutInflater.from(getContext()).inflate(R.layout.l
                ist_item_recommendation, parent, false);
62.
63.         TextView textView = convertView.findViewById(R.id.textViewHead);
64.         TextView textView2 = convertView.findViewById(R.id.textViewWineryYear)
        ;
65.         TextView textView3 = convertView.findViewById(R.id.textViewNbrReviews)
        ;
66.         RatingBar ratingBar = convertView.findViewById(R.id.ratingBar);
67.         Button button = convertView.findViewById(R.id.ButtonPrice);

```



```

68.     ImageView imageView = convertView.findViewById(R.id.img_wine_list);
69.
70.     textView.setText(wine.getName());
71.     textView2.setText(wine.getWinery()+" "+wine.getYear());
72.     textView3.setText(""+wine.getReviewsNbr()+" Reviews");
73.     ratingBar.setRating(wine.getRating());
74.     button.setText(Float.toString(wine.getPrice()));
75.
76.     button.setOnClickListener(new View.OnClickListener() {
77.         @Override
78.         public void onClick(View v) {
79.             setDialog(v, wine);
80.         }
81.     });
82.     imageView.setOnClickListener(new View.OnClickListener() {
83.         @Override
84.         public void onClick(View v) {
85.             Intent intent = new Intent(getContext(), WineInfoView.class);
86.             intent.putExtra("id", wine.getId());
87.             getContext().startActivity(intent);
88.         }
89.     });
90.
91.     if(id.equals("Recommendation"))
92.     {
93.         switch (position)
94.         {
95.             case 0:
96.                 textView2.setText("Matched 1");
97.                 imageView.setImageResource(R.drawable.ic_gold_medal);
98.                 break;
99.             case 1:
100.                 textView2.setText("Matched 2");
101.                 imageView.setImageResource(R.drawable.ic_silver_med
al);
102.                 break;
103.             case 2:
104.                 textView2.setText("Matched 3");
105.                 imageView.setImageResource(R.drawable.ic_bronze_med
al);
106.                 break;
107.         }
108.     }
109.     return convertView;
110. }
111.
112. /*
113.     setRecommendationPresenter():
114.         This method sets the presenter of the
115.         recommendation if the adapter is used in
116.         the recommendations
117. */
118.
119.     public void setRecommendationPresenter(RecommendationPresenter reco
mmendationPresenter)
120.     {

```

```

121.         this.recommendationPresenter = recommendationPresenter;
122.     }
123.
124.     /*
125.     setShopPresenter():
126.         This method sets the presenter of the
127.         shop if the adapter is used in
128.         the shop
129.     */
130.
131.     public void setShopPresenter(ShopPresenter shopPresenter)
132.     {
133.         this.shopPresenter = shopPresenter;
134.     }
135.
136.     /*
137.     setDialog():
138.         This method opens an AlertDialog
139.         if the user wants to add the wine
140.         to the cart.
141.     */
142.
143.     public void setDialog(View v, Wine wine)
144.     {
145.         View finalConvertView = LayoutInflater.from(v.getContext()).inflate(R.layout.fragment_bottles_nbr_dialog, (ViewGroup) v.getParent(), false);
146.         buttonBack = finalConvertView.findViewById(R.id.buttonBack);
147.         buttonAccept = finalConvertView.findViewById(R.id.buttonAccept);
148.         ;
149.         ImageButton imageButtonAdd = finalConvertView.findViewById(R.id.imageButtonAdd);
150.         ImageButton imageButtonSubtract = finalConvertView.findViewById(R.id.imageButtonSubtract);
151.         TextViewNbrBottles = finalConvertView.findViewById(R.id.textVie
152.         wbrBottles);
153.
154.         ButtonClick buttonClick = new ButtonClick(wine);
155.         buttonAccept.setOnClickListener(buttonClick);
156.         buttonBack.setOnClickListener(buttonClick);
157.         imageButtonAdd.setOnClickListener(buttonClick);
158.         imageButtonSubtract.setOnClickListener(buttonClick);
159.
160.         dialog = new BottomSheetDialog(getContext());
161.         dialog.setContentView(finalConvertView);
162.         dialog.setTitle("Add Wine");
163.         dialog.show();
164.     }
165.
166.     class ButtonClick implements View.OnClickListener
167.     {
168.         int nbr;
169.         Wine wine;
170.
171.         public ButtonClick(Wine wine)
172.         {
173.             this.wine = wine;

```



```

172.         nbr = 1;
173.     }
174.
175.     public void onClick(View v)
176.     {
177.         switch (v.getId())
178.         {
179.             case R.id.buttonBack:
180.                 dialog.dismiss();
181.                 textViewNbrBottles.setText("1");
182.                 nbr = 1;
183.                 break;
184.             case R.id.imageButtonSubtract:
185.                 if(nbr > 1)
186.                 {
187.                     nbr = nbr - 1;
188.                     textViewNbrBottles.setText(String.valueOf(nbr))
189.                 }
190.                 break;
191.             case R.id.imageButtonAdd:
192.                 if(nbr >= 1)
193.                 {
194.                     nbr = nbr + 1;
195.                     textViewNbrBottles.setText(String.valueOf(nbr))
196.                 }
197.                 break;
198.             case R.id.buttonAccept:
199.                 if(id.equals("Recommendation"))
200.                     recommendationPresenter.addToCart(wine, nbr);
201.                 else
202.                     shopPresenter.addToCart(wine, nbr);
203.                 nbr = 1;
204.                 textViewNbrBottles.setText(String.valueOf(nbr));
205.                 dialog.dismiss();
206.                 break;
207.         }
208.     }
209. }
210. }

```