



INTERNATIONAL
HELLENIC
UNIVERSITY

Development of a Smart Ordering System

Georgios Michailidis

**UNIVERSITY CENTER OF INTERNATIONAL PROGRAMMES OF STUDIES
SCHOOL OF SCIENCE AND TECHNOLOGY**

A thesis submitted for the degree of
Master of Science (MSc) in Mobile and Web Computing

MARCH 2021
Thessaloniki – Greece

Student Name: Georgios Michailidis
SID: 3306190002
Supervisor: Prof. Leonidas Akritidis

I hereby declare that the work submitted is mine and that where I have made use of another's work, I have attributed the source(s) according to the Regulations set in the Student's Handbook.

March 2021
Thessaloniki - Greece

Abstract

This dissertation was written as part of the MSc in Mobile & Web Computing at the International Hellenic University.

The commercial sector is undoubtedly one of the greatest in the economies of all countries and so retailers strive to increase their profits daily by attracting more customers and more sales. Their main problem is when they get out-of-stock on products that attract these customers, ending up losing them and thus losing their profits. That being said, one can clearly understand that the re-supply process of a retailer plays a crucial role to the wellbeing of the business and sticking to traditional ways, away from information technology, leads to an even worse situation. IoT (Internet of Things) devices and technologies should be used to transform the aforementioned process into a simple and easy task and minimize the stock-out problem that retailers have to deal with.

In this dissertation, an IoT system is proposed to help retailers to confront this problem and limit its consequences as much as possible. The system has two parts, the vendor system which, as the name implies, is used by vendors who provide supplies and the retailer system that is used by the retailers. These two IoT systems communicate with each other through the internet and provide a retailer with the ability to re-supply from all its vendors at once without having to organize which products should be sent to which vendor through his order. Furthermore, the option to place again a previous order is given, allowing the retailer to re-supply the next time even faster, since they tend to order the same supplies.

Keywords: smart, ordering system, iot, vendor, retailer

Georgios Michailidis
13/03/2021

Contents

ABSTRACT	III
CONTENTS	I
1. INTRODUCTION.....	1
2. RELATED WORK	2
3. PROBLEM DEFINITION.....	5
4. SOLUTION TO THE PROBLEM	7
4.1 VENDOR SYSTEM	10
4.2 RETAILER SYSTEM	19
4.3 CONNECTION OF THE TWO SYSTEMS	30
5. SYSTEM EVALUATION	30
6. CONCLUSIONS.....	33
BIBLIOGRAPHY	34
APPENDIX A – VENDOR SYSTEM SOURCE CODE	1
MIDDLEWARE CLASSES	1
<i>Middleware - Authenticate</i>	1
<i>Middleware - ApiAuthenticate</i>	2
CONTROLLER CLASSES.....	2
<i>AuthController</i>	3
<i>HomeController</i>	4
<i>ProductsController</i>	5
<i>CustomersController</i>	8
<i>OrdersController</i>	11
<i>UsersController</i>	15
<i>ApiController</i>	18
MODEL CLASSES.....	21
<i>Customers Model</i>	22
<i>Products Model</i>	22

<i>Orders Model</i>	23
<i>OrderItem Model</i>	24
APPENDIX B – RETAILER SYSTEM SOURCE CODE	25
ACTIVITIES	25
<i>Main Activity</i>	25
<i>Catalog Activity</i>	26
<i>Cart Activity</i>	28
<i>Vendors Activity</i>	30
<i>Orders Activity</i>	31
<i>OrderDetail Activity</i>	31
ASYNC TASK CLASSES	33
<i>ProductFetchTask</i>	33
<i>PlaceOrderTask</i>	35
OTHER CLASSES	37
<i>EndlessRecyclerViewScrollListener</i>	37
<i>VendorCoordinator</i>	39
<i>Model Class</i>	41

1. Introduction

Today, the commercial sector is one of the greatest and most important one in the economy of a country. The main goal of the businessmen in this sector, as well as of almost every human being, is to maximize their profit in their everyday lives. To achieve this goal, a business has to make sales in a constant or increasing rate, over time. A business has to keep attracting new customers and motivate existing ones to proceed to new purchases, in order for sales to be made. One can perceive that it is crucial for a retail store to have an adequate stock of products and in case that some products are out of stock, they must be replenished as quickly as possible. If that requirement cannot be met, customers will become dissatisfied and will reduce their purchases or, in the worst-case scenario, switch to competitor retailers.

It has been observed, that up to this day, many retailers still use conventional methodologies to operate in their business process. They keep track of their stock, literally on paper and resupply through phone calls or emails, contacting one by one their partner vendors. By sticking to these traditional ways of operation, they delay their re-stock process and dissatisfy existing or potential customers, leading them to switch to competitors and thus, losing profits. Consequently, it is wise to incorporate the ever-advancing IoT technologies and devices in their business process. IoT (or Internet of Things), refers to any kind of entity, devices, objects or even people that communicate through any kind of network, internet or local network, and exchange information and possibly do some processing on that information (K. K. Patel & S. M. Patel, 2016).

The goal of this dissertation is the development of a smart ordering system based on IoT technologies that will help to minimize the problem that retailers have to face. Similar systems have been developed to tackle this problem that deal with stock monitoring with the use of IoT sensors. There are also systems that provide a complete business-to-business and even business-to-customer electronic shopping solution, like PrestaShop. Other than that, it is crucial to deeply understand the main problem of the retailers and why it should be dealt swiftly and efficiently. More details are provided in the next chapters of this dissertation. Nevertheless, the most important part is the proposed system itself and how it helps with the problem. The system, which is named

Ventail, consists of two sub-systems, one web application for vendors and one mobile application for the retailers. These two applications can communicate with each other and provide an efficient way of resupplying the retailers. Finally, the evaluation of the entirety of the system is crucial as well and through tests and experiments, it was concluded that it can achieve its goal.

2. Related Work

During this research, no scientific article about ordering systems between retailers and their vendors could be found. On the contrary, only articles regarding local shelf stock management systems or web applications that allow ordering between end customers and retailers through their ecommerce websites were evident and thoroughly discussed in the academic community. Nonetheless, the latter two cases are fairly similar in many aspects with an ordering system between retailers and vendors. They portray exactly the same processes when browsing a product catalogue, adding products to a shopping cart and finally placing an order when it comes to buying those products. Such ordering systems allow, on one hand, customers to order their desired products and, on the other hand, retailers to manage their product catalogue and incoming orders. Also, IoT systems that use sensors and artificial intelligence software for warehouse management seem to be quite promising. The problem with these systems is that normally, technology based on sensors or smart software tends to be very costly and most retailers may not afford them.

Additionally, these systems manage to solve the stock out problem only in a local level. In other words, the platform cannot inform the retailer about the exhaustion of a product, unless this product already exists in the supplying warehouse. But in the case that there are no products available either in the store or in the warehouse whatsoever, the already existing systems do not seem to be of any use at all. Nonetheless, it is useful to study such systems and compare them with the proposed system.

Before diving into the IoT systems, it would be wise to explain what IoT stands for. It is an abbreviation of the phrase "Internet of Things" and, as Madakam, S., Ramaswamy,

R., and Tripathi, S. (2015) mention, it refers to everything that can connect to a network, from humans to devices that can make some calculations or execute processes and possibly exchange data with other such devices over the network. IoT devices vary from sensors to smartphones, personal computers and laptops and they can be interconnected either through the internet or any local network. They normally produce some data and send them over the network to another IoT device to process them or require some data to function.

Tejesh, B. S. S, and Neeraja, S. (2018) developed a system that helps retailers manage their stock located in their warehouse, with the use of IoT devices. More specifically, their system makes use of RFID tags, small pieces of plastic that emit constantly low frequency radio-waves. Those tags are used to track the products in the warehouse, so each one of them is placed on a product. Additional information about the product is written in those tags in a digital form. The RFID tags constantly emit their data over radio-waves and some receivers collect them, extract the data and send them to the main application of the warehouse management. Through the application, the retailer is able to list all the products in the warehouse and their remaining quantities, at any moment and in real-time. The system provides valuable information about stock management, but it does not allow the retailer to place orders and refill the warehouse. Instead, the retailer has to manually write down all the required products and also manually proceed to ordering all of them from his partner vendors. Thus, the stock out problem is not alleviated in a great level with such a solution.

On the other hand, there are web applications that provide a more user-friendly and efficient way of placing orders. One such system is *PrestaShop*, an e-commerce Content Management System (CMS) that allows developers to design, build and setup an e-commerce website or e-shop for a retailer. *PrestaShop* consists of two sub-systems, the front-office and the back-office. The front-office part of *PrestaShop* is the e-shop website itself. There the customers can browse through the product catalogue, search for products or apply filters to match their needs, add products to their shopping cart and finally place an order. Through the front-office a customer has also the option to track his orders, so as to find which progress stage they are at, by checking their status. The back-office sub-system, on the other hand, is the administration and management web application of the e-shop that concerns only the

retailer. Using this application, the retailer can manage the product catalogue by adding, updating or deleting products. The incoming orders may also be managed by viewing their details and changing their progress status. This includes a customer management tool for checking their information with the aim of delivering the correct products to the appropriate person. The architecture of this system follows a centralized approach, since both sub-systems belong to the same web application, running in the same server and all stored data can be found in one place.

There are many similarities and a couple of differences between the proposed system and PrestaShop. More specifically, the former includes two sub-systems, the retailer mobile application and the vendor web application. The retailer system resembles the front-office, where the retailer plays the role of the customer of a vendor. In this context, the retailer-customer:

- browses the product catalog offered by the vendor,
- searches and filters products,
- selects the desired products and adds them to a shopping cart,
- submits orders to the vendor via a standard ordering interface,
- monitors the status and/or tracks the submitted orders.

Similarly, the vendor system represents the PrestaShop back-office, where a vendor:

- lists and manages the catalogue of the offered products,
- manages the customers (who are all retailers),
- processes the incoming orders,
- updates the order statuses to inform the customers in the context of a convenient after-sales service.

Beyond the aforementioned similarities, there is a wide variety of elements that discriminate the proposed system from PrestaShop. The most important is that, in PrestaShop customers can register and create accounts themselves in the e-shop, whilst in the proposed system a vendor has to register each of their retailers, generating an API key and sharing it with each one of them. From the viewpoint of the retailer, all vendors must be registered in the application; this is achieved by obtaining one API key from each vendor. Another difference is that in a PrestaShop website,

multiple customers order from one and only one retailer, while in Ventail, multiple retailers may place orders to vendors at once. A third difference is that PrestaShop uses a centralized architecture, in contrast to the decentralized design of this application. This offers the advantage of efficiently working with these many-to-many relationships between retailers and vendors.

3. Problem Definition

It is widely known that a great deal of small and medium retailers do not use any kind of Information Technology or automation solutions in their business operation. Even today they stick to traditional, old-fashioned and inefficient practices. The most important part of their operation deals with stock management, since having available products to sell at any time, is what brings profit to a retailer. According to these traditional practices, retailers take inventory by browsing through every shelf in their warehouse and counting all of their products to finally write down on paper the amount of the available ones. This process is definitely time consuming and prone to errors since everything is done by the human hand. Another traditional stock management practice that constitutes an issue for improvement, is the way retailers order their supplies. In order to restock their warehouses, they have to go through their stock list to find all the products that are partially, or fully out-of-stock and possibly create a new list with the products and the amounts needed for each vendor. That is because each vendor that a retailer works with, offers specific products, so the retailer has to create multiple lists if there are any needed products from different vendors. The final step is to either send each order list to the respective vendor via email or call via telephone each vendor and place the order with the desired items orally.

Sticking to such practices, those small and medium retailers end up adding a lot of delay to their restock process. It takes a lot of time to place just one order through a telephone call, let alone orders to multiple vendors. Adding this wasted time to the time the vendor needs to deliver the products to the retailer, it leads to a huge amount of time until the retailer will be able to sell those products again. In other words, the retailer faces the risk of ending up in a stock out state.

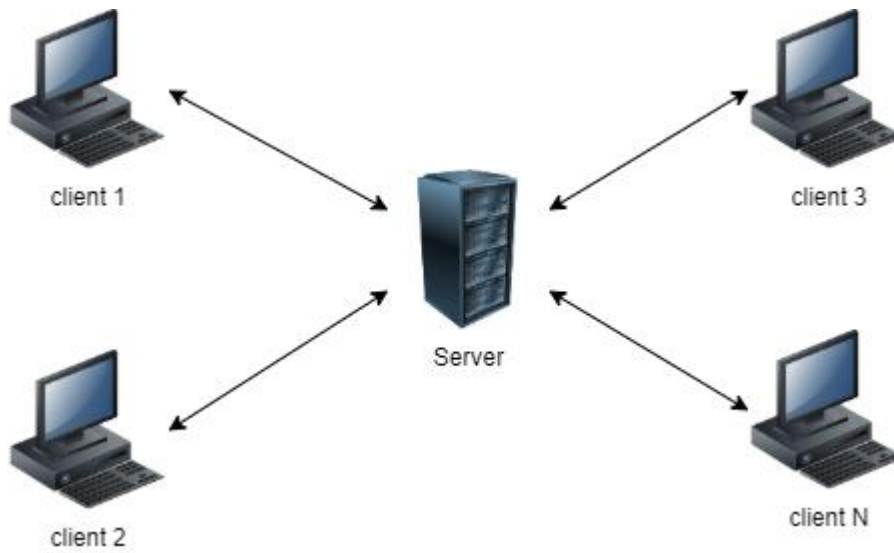
Nowadays, the aforementioned out-of-stock situation is the main problem that retailers have to avoid. It is not unusual that no matter how experienced they may be in their field, retailers still face the risk of not being able to properly and efficiently manage their stock. Frontoni et. al. (2016) mention that this out-of-stock situation occurs at the time that specific merchandise is not available for consumers to obtain resulting in the need to look for other substitutes.

The problem mainly originates from the retailers themselves, since they do not make use of appropriate ordering methodologies, as both researches of Corsten and Gruen (2003) and Fernie and Grant (2008) suggest. Every retailer should know and use modern and efficient technologies that can deal with this problem. But since technology rapidly advances, they should also keep track of the innovations that may occur, which would make this process even faster and more user-friendly.

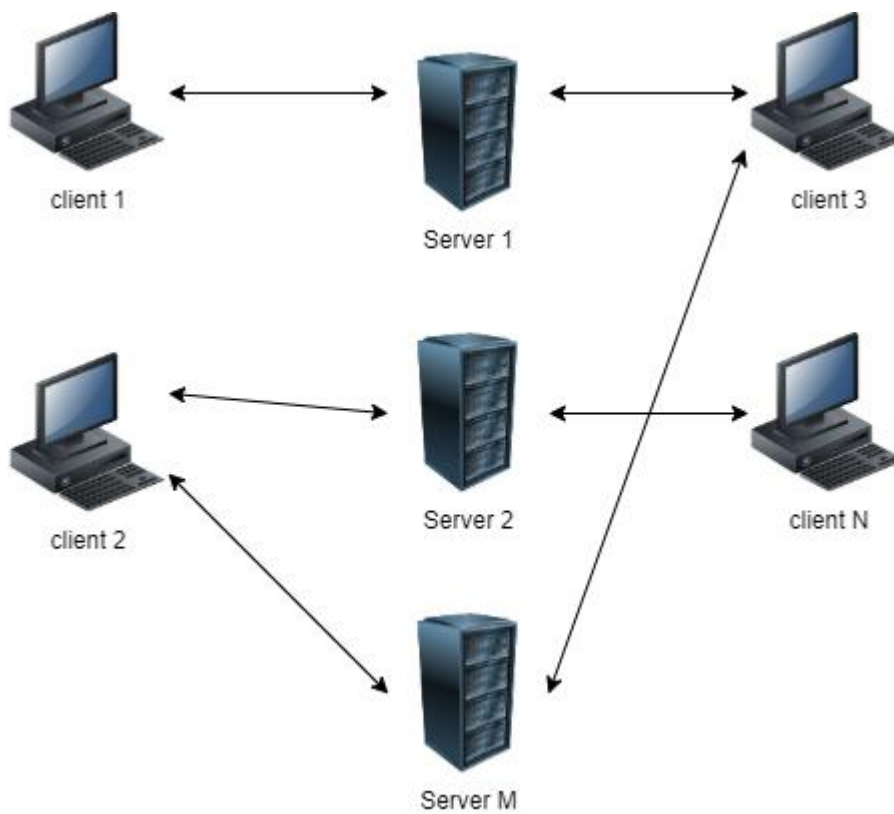
The impact of stock-outs for a retailer could be rather severe. According to Verhoef and Sloot (2006) the most acute ramifications of stock-outs for retailers is when customers proceed to not buy anything at all or, even worse, when they turn to a competitive retailer, where they will find the desired merchandise. Having dissatisfied customers is the first step towards closing-doing. And it becomes even worse when these dissatisfied customers choose to purchase what they need from a competitor, so that the adversary business thrives, while the business that has stocked-out steadily declines. Another research conducted by Pizzi and Scarpi (2013) clearly supports that customers can become dissatisfied with a retailer in the long term when stock-outs occur, as well. As follows, the income of the retailer with stock-outs will decrease, since a large percentage of the disappointed customers could turn to other retailers or even lose their interest to buy any products for a certain period of time. But some customers may become so dissatisfied that they will never turn to this business again in the future. Thus, a mismanagement of the stock can lead to significant delays, justified dissatisfaction on behalf of the customers, and eventually loss of income. This is the reason why it is of paramount importance for a retailer and the sustainability and growth of his company to be in the position to refill his stock exactly at the right time.

4. Solution to the Problem

A problem like this can be effectively tackled by using an appropriate software, and smart technologies. This dissertation presents the core design and development elements of a smart ordering system that provides retailers with a smart, fast and easy way of managing their supply orders across their vendors. The system itself consists of two subsystems, the vendor system and the retailer one. Both of them are connected through an Application Programming Interface (API) that belongs to the vendor system. Regarding the design architecture of the system, there are two possible ways to follow, centralized or decentralized architecture (Akritidis et.al. 2018). With a centralized architecture (Picture 1), the server-side processing or the business logic of the system, takes place in one central place, normally in one server machine which runs the server-side application of the system. Then, all client-side applications have to connect to that server to process any kind of business logic. On the other hand, in a decentralized architecture (Picture 2), the server-side application of the system is located in many different servers, each running as an individual, and independent application with its own database. In this case, the client-side applications may connect to multiple different servers, though one at a time, to execute their business logic.

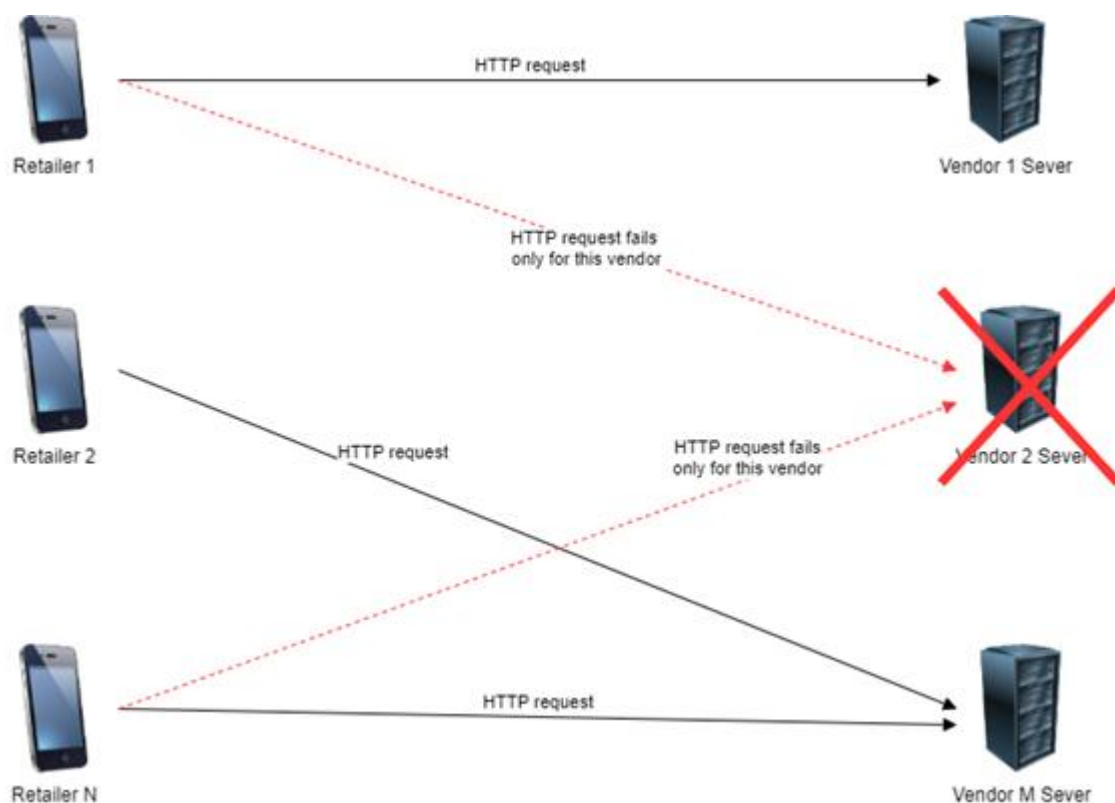


Picture 1: Centralized architecture



Picture 2: Decentralized architecture

Regarding the proposed system, a decentralized architecture was used, with the vendor system representing a server-side application running on the server of each vendor and the retailer system representing the client-side application. The reason this architecture was preferred over the centralized one is because its advantages are of more importance, as stated by Robin Jan Maly (2003). First and foremost, the decentralized design provides great availability since any single point of failure is absent. In contrast to the decentralized design, in the centralized architecture, if the server running the business logic fails for any reason, then none of the client-side applications will be able to perform its intended functionalities. On the contrary, the existence of multiple servers in the decentralized architecture guarantees that in the case of an isolated failure, the clients will still be able to perform their functionalities, on any other available server. In the current case, if the server of one of the vendors fails, then the retailers will still be able to use the system to place orders to all the other vendors (Picture 3).



Picture 3: Smart Ordering System - Decentralized

Another advantage is that it provides better communication speed, because the network bandwidth for the communication between the client-side and server-side application is distributed among multiple servers and thus increasing the overall network speed that the final user experiences. In other words, in a centralized system all clients send requests to one server over the internet, overloading the network and increasing the latency. On the opposite side, a decentralized system distributes the requests of clients to multiple different servers each time and decreases the load of the network, allowing faster connection over the internet. Finally, the decentralized design reduces the load of the server processing, leading to a faster response to the client. On a centralized system, all clients would send requests to one server with limited resources and it would take more time to process each request and send a response accordingly. Opposed to that, the decentralized architecture once again distributes the requests from all clients to multiple servers, so each server has to deal with fewer requests thus increasing the speed of the process. Relative to the proposed system, retailers can only connect to their partner vendors business logic and the vendor himself is the only one who can connect to the back office of the system, meaning the both the network bandwidth and the server load is going to be significantly low.

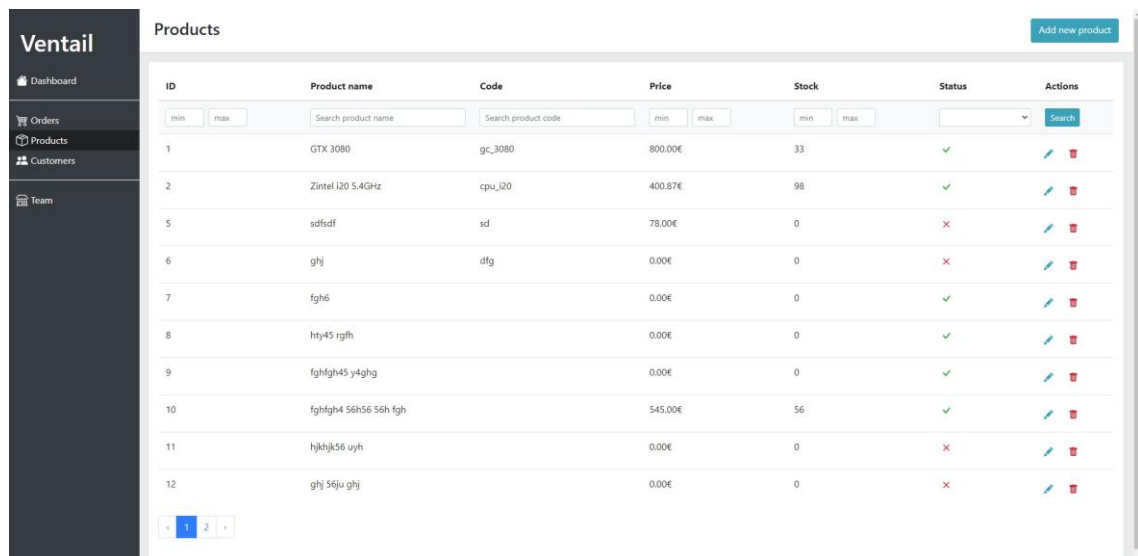
4.1 Vendor System





















This subsection describes in more details the system of the vendor. Concerning the technical details, this system is a web application developed with Laravel version 8, one of the best PHP frameworks. PHP is a server-side programming language that is used to develop websites and web applications (Sklar, 2016). In a typical setup, a PHP application is executed by a module (CGI, FastCGI or other) of an HTTP server that allows multiple clients to access this application at the same time. Therefore, the combination of PHP with an HTTP server allows the design of rich, multi-user applications. In addition, the PHP-powered web pages are dynamic which means that their content changes according to the submitted request of the client. In contrast, the static Web content is always the same, regardless of the parameters of the submitted

request. A dynamic Web page can connect and receive data from a local/remote database system, or a remote Web service, or an API.

The designed vendor system requires PHP version at least 7.3 and uses MySQL version 5 as the main database service. The Application Programming Interface is also developed with Laravel, as a sub-component of the vendor system, utilizing the efficient and smart routing capabilities of this powerful framework. More information about the API will be documented in the chapter 4.3 below.

Now let us continue with the functionalities that the system provides to the vendors. One important service is the ability to manage their product catalog. At first, they are presented with a list of products (Picture 4) that they have already registered in the system. This list shows some basic information about each product such as the product ID from the database record, the product name and code, its price the quantity of its stock and its status which has two possible values, enabled or disabled. Products with status set to enabled, are the only ones that will be shown to the retailer system and the ones with disabled status will be hidden. That way, a vendor can easily determine which products are active for sale and which are not, for example because they temporary cannot sell them for any reason.

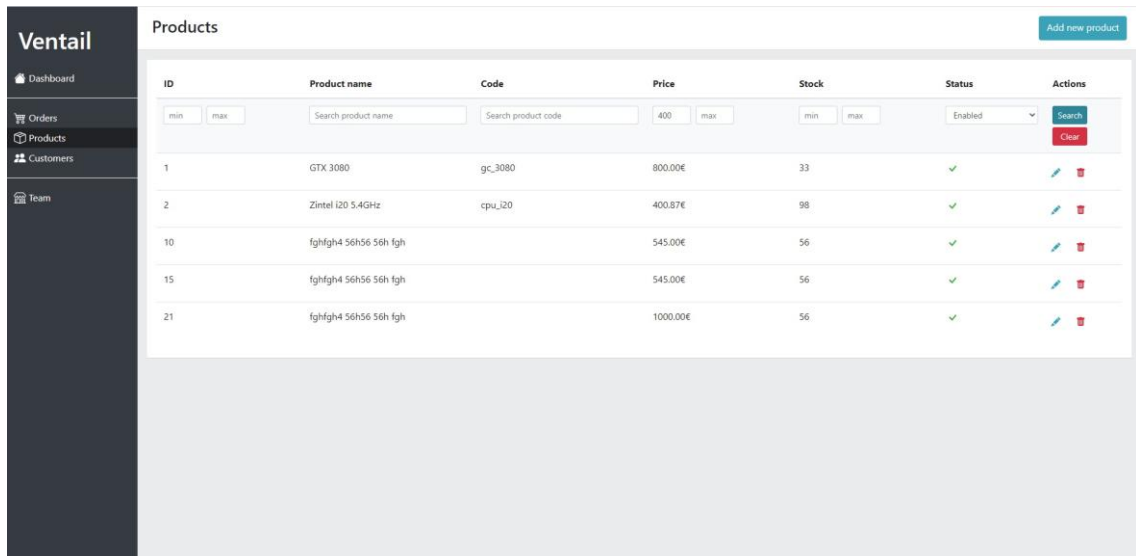


ID	Product name	Code	Price	Stock	Status	Actions
1	GTX 3080	gc_3080	800.00€	33	✓	 
2	Zintel i20 5.4GHz	cpu_i20	400.87€	98	✓	 
5	sdfsdf	sdf	78.00€	0	✗	 
6	ghj	dfg	0.00€	0	✗	 
7	fgh6		0.00€	0	✓	 
8	hty45 rgfh		0.00€	0	✓	 
9	fghfgh45 y4ghg		0.00€	0	✓	 
10	fghfgh4 56h56 56h fgh		545.00€	56	✓	 
11	hjhkh56 uyh		0.00€	0	✗	 
12	ghj 56ju ghj		0.00€	0	✗	 











Picture 4: Product list

Additionally, the product list provides a searching ability to the vendor. Among a set of filters, the vendor can choose any combination of them and press the “Search” button to get the desired products. The available filters are a range of product ID, the product

name, the product code, a range of price values, a range of stock quantities and the status of the product. If any products match those filters, they will be shown in the list after clicking the “Search” button (Picture 5). After a search attempt, the “Clear” button is displayed in the search bar, allowing the clearing of all search filters so that the entire, unfiltered product list is displayed.



The screenshot shows the 'Products' page in the Ventail system. The left sidebar contains navigation links for Dashboard, Orders, Products, Customers, and Team. The main content area displays a table of products with the following columns: ID, Product name, Code, Price, Stock, Status, and Actions. The table is filtered to show only products with a minimum price of 400 and a status of 'Enabled'. The search filters are visible at the top of the table, showing 'min' and 'max' for price and 'Enabled' for status. The 'Search' button is highlighted in blue, and the 'Clear' button is highlighted in red.

ID	Product name	Code	Price	Stock	Status	Actions
1	GTX 3080	gc_3080	800.00€	33	✓	 
2	Zintel i20 5.4GHz	cpu_i20	400.87€	98	✓	 
10	fgfhgh4 56h56 56h fgh		545.00€	56	✓	 
15	fgfhgh4 56h56 56h fgh		545.00€	56	✓	 
21	fgfhgh4 56h56 56h fgh		1000.00€	56	✓	 

Picture 5: Filtered product list with minimum price (400) and status (enabled).

Another feature of the product catalogue is the option to add a new product to it (Picture 6). The vendor can enter all the necessary information about the product, like its name, status, code, barcode, price, stock and description and save it permanently to the database.

The screenshot shows the 'New Product' form in the Ventail application. The interface includes a dark sidebar with navigation options: Dashboard, Orders, Products (selected), Customers, and Team. The main content area is titled 'Product name' and contains several input fields: 'Product name' (with a 'Disabled' toggle), 'Product Code', 'Barcode', 'Description' (a large text area), 'Stock', and 'Price' (with a currency selector set to '€'). A green 'Save' button is located at the bottom right of the form.

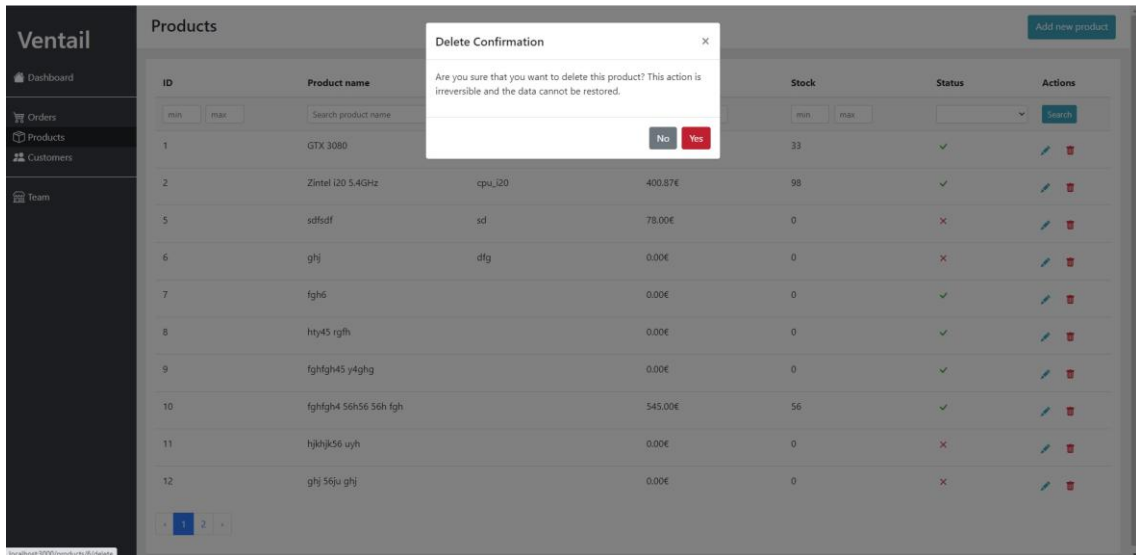
Picture 6: Screen of new product addition.

Furthermore, an existing product from the product list can be edited, by pressing the “Edit” button. The edit screen of the product (Picture 7) is similar to the one with the addition of a new product and all existing data of the product is loaded in the appropriate fields. Any of the data can be changed and saved permanently. Moreover, the product can be deleted by pressing the “Delete” button.

The screenshot shows the 'Product Edit' form in the Ventail application. The interface is similar to the 'New Product' form, but with pre-filled data. The 'Product name' field contains 'GTX 3080' and has an 'Enabled' toggle and a red 'Delete' button. The 'Product Code' field contains 'gc_3080', and the 'Barcode' field contains '1234342234334'. The 'Description' field contains the text 'Super duper graphics card of the 30 series. asd asd'. The 'Stock' field contains '33', and the 'Price' field contains '€ 800,00'. A green 'Save' button is located at the bottom right of the form.

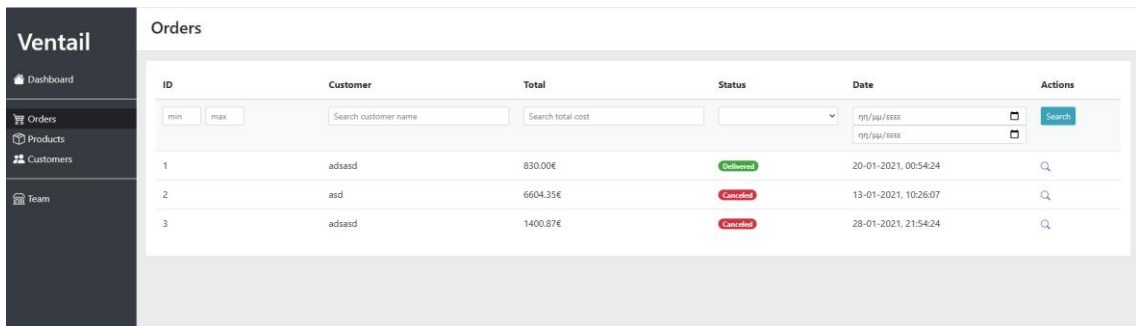
Picture 7: Screen of product edit.

Finally, the deletion of a product can be initiated from the product list, by clicking the “Deleted” button of the desired product and confirm the deletion from the popup message that appears (Picture 8).



Picture 8: Deletion confirmation message.

In the sequel, the most important feature of this system, the order management module is presented. A vendor can list all the orders (Picture 9) that were placed from all customers. The list contains basic information about each order, like the ID number from the database record, the name of the customer who placed the order, the total cost of the entire order, the current status and the date that it was placed.

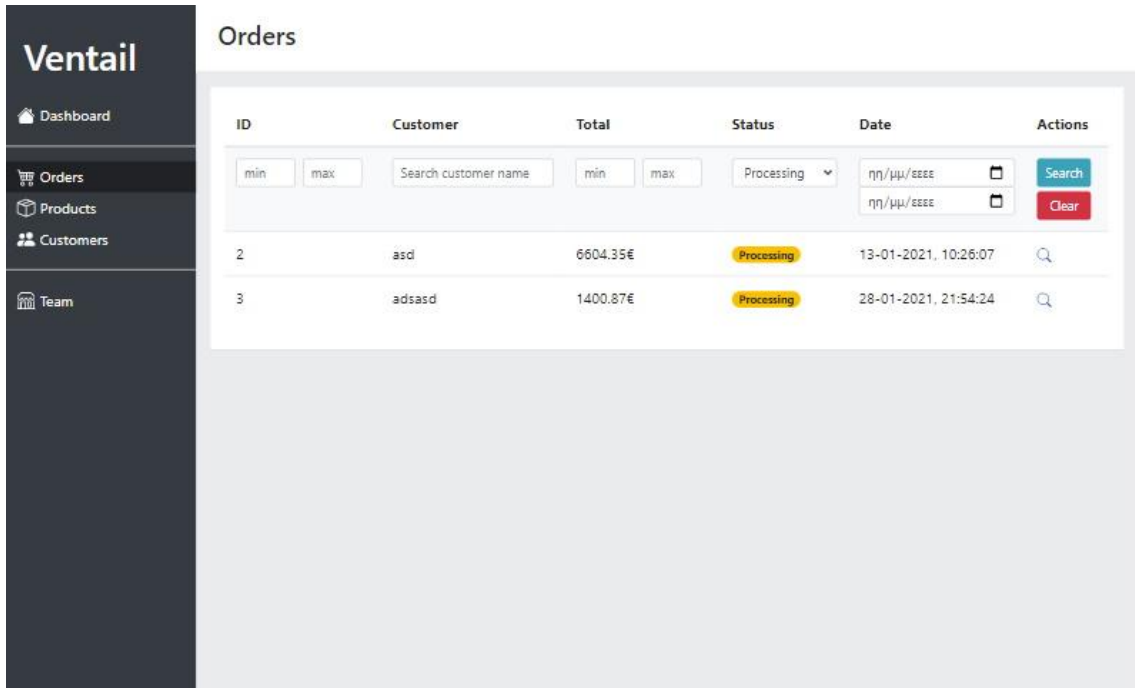


Picture 9: List of orders

The order status may be assigned three different values: “processing” when it is placed and created, “delivered” when the vendor finishes the order and sends it to the customer and “canceled” when the vendor has any reason to cancel a specific order.

The order management comes with a searching/filtering mechanism, similar to the one in the product catalogue. There are several filters matching the information fields provided by the order list and those are a range of order IDs, the name of the customer, a range of the order total, the status and a range of dates that the order was

placed. Following the same logic with the product catalogue, one or more filters can be combined and by pressing the “Search” button, if any orders match those criteria, they will be shown on the list (Picture 10). There is a “Clear” button to remove all search filters as well.

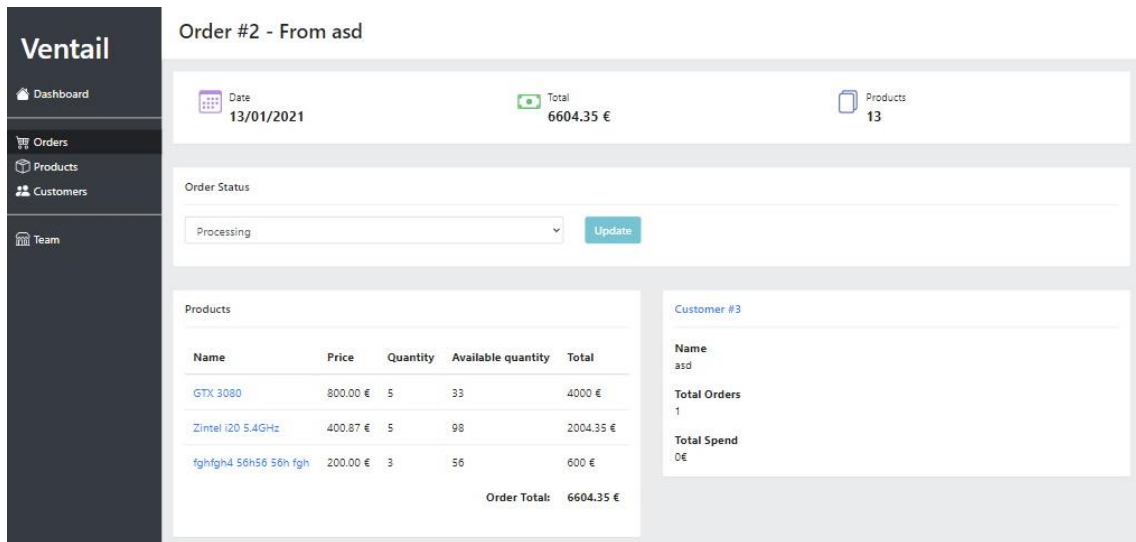


The screenshot shows the 'Orders' page in the Ventail system. On the left is a dark sidebar with the 'Ventail' logo and navigation links for Dashboard, Orders, Products, Customers, and Team. The main content area is titled 'Orders' and features a table with columns for ID, Customer, Total, Status, Date, and Actions. Above the table is a search and filter interface with input fields for 'min' and 'max' values, a search box for customer names, a status dropdown menu set to 'Processing', and date pickers. A 'Search' button and a 'Clear' button are also present. The table displays two orders, both with a 'Processing' status.

ID	Customer	Total	Status	Date	Actions
2	asd	6604.35€	Processing	13-01-2021, 10:26:07	
3	adsasd	1400.87€	Processing	28-01-2021, 21:54:24	

Picture 10: Filtered order list with status (processing)

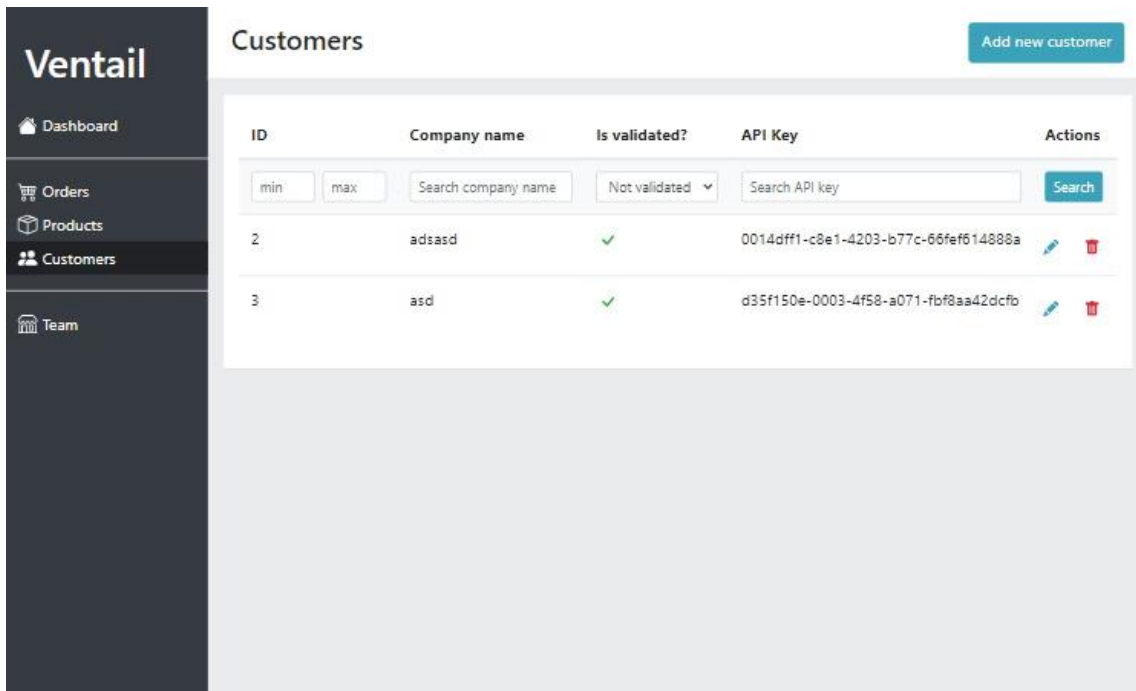
Moreover, a vendor can view in detail an order by clicking the “View” button of an order on the order list. At the very top of the order view screen (Picture 11), there is information about the name of the customer, the date that the order was placed, the order total and the total quantity of all products. Below this information, the order status can be found and can be updated at any time. Lastly, there are two sections, one showing the products that were ordered along with their price, quantities, stock and total cost of each product and the other one showing general information about the customer, such as the number of completed orders and the total amount of money earned from this customer.



Picture 11: The details view of an order.

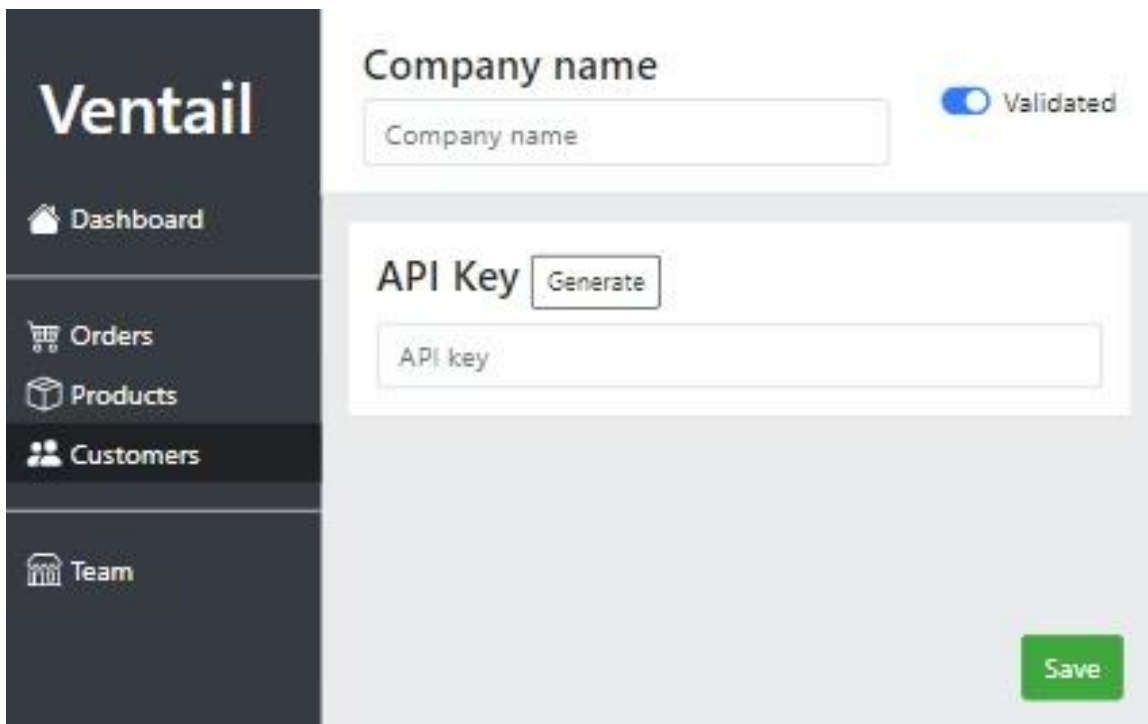
Another functionality of the system, which is equally important, is the customer management subsystem which allows a vendor to manually register the customers into the system and grant access to the product catalog and the ability to place orders. In fact, customers cannot register themselves through the retailer system, because the entire system follows a Business-to-Business (B2B) principle and each customer of a vendor is predetermined by a contract. So, to protect the vendor system against unauthorized access and since a vendor already has the details of the clients, he can easily register them into the system. The registration will generate a unique API key, for each customer, which is used for the connection between the two systems and it will be described in more detail in subsection 4.3 below.

There is also a customer list (Picture 12) in this section which sums up the essential data of each customer, like the database ID of the record, the name, the API key and the status. The status of a client may receive one of the following four values: enabled/disabled and validated/invalidated. Only validated customers can actually have interactions with the vendor system. The searching functionality could not be absent from customer management and it provides filters like a range of database IDs, the name, the status and the API key.



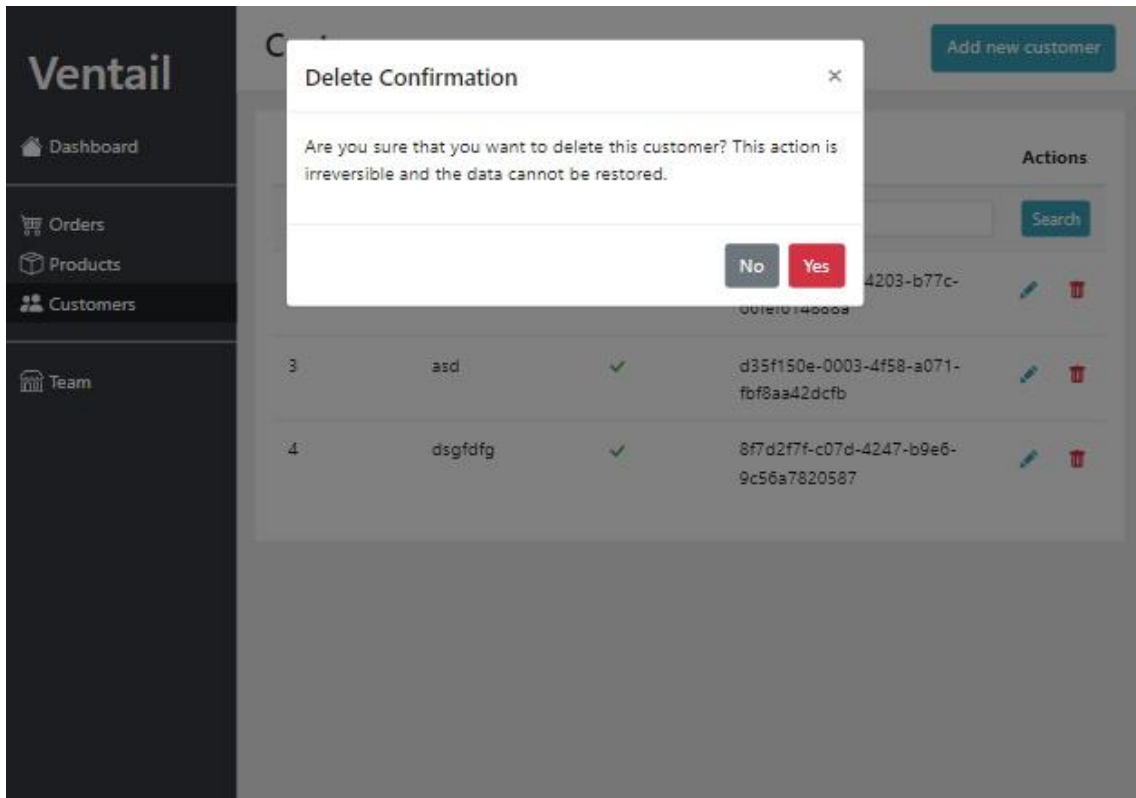
Picture 12: Customer List.

The registration of a customer is a quite simple process and the vendor has to press the “Add new customer” button to show the appropriate screen (Picture 13), input the data and save the record. There is also an update functionality, which follows the same logic and layout of the registration of a customer.



Picture 13: Customer Registration – Update Screen.

Lastly, there is also the option to delete a specific customer, provided that no correlated orders exist. For the deletion, a vendor has to click the “Delete” button of a specific customer on the list and confirm the popup message (Picture 14). If the customer has at least one order, the system will not allow the deletion and will show an error message accordingly.

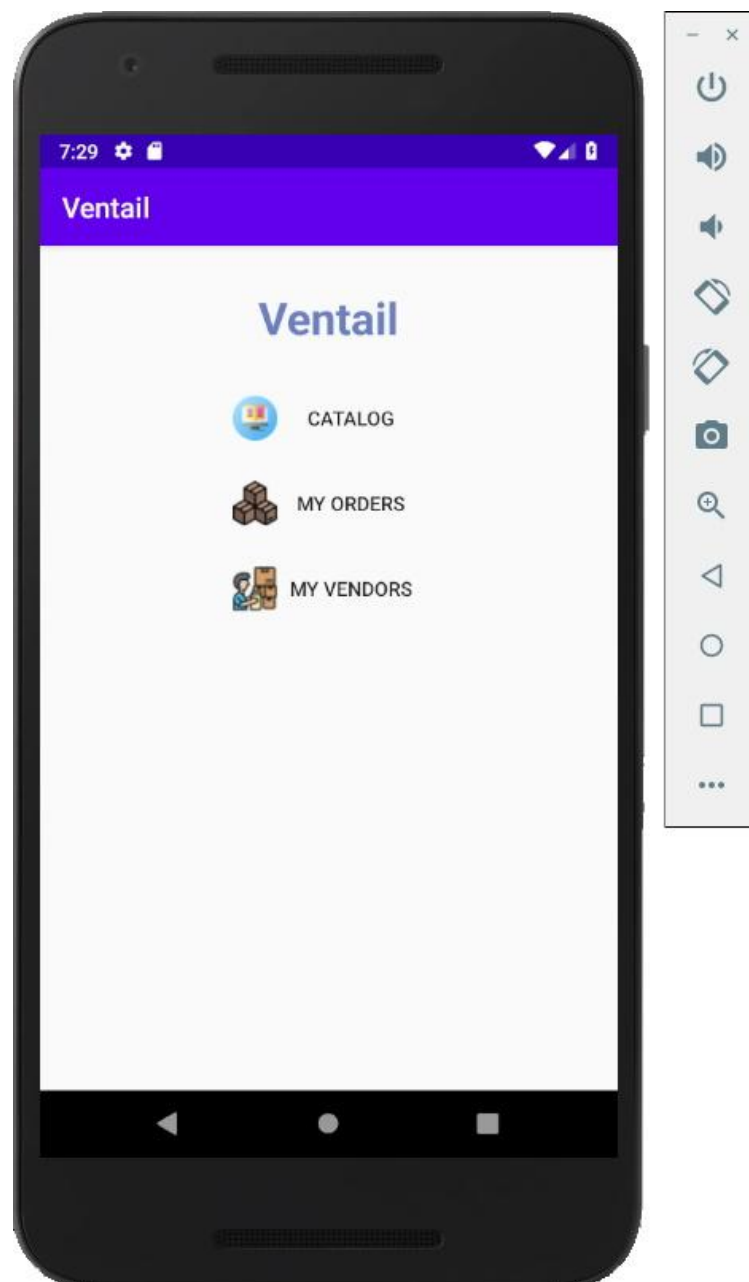


Picture 14: Customer deletion popup message.

Finally, the least significant functionality of the system is the user management module. By user, it is meant the user of the vendor system, the people that manages the application, which are the vendor and probably his employees. A list of all users is available and more users can be created or existing ones can be updated on their data. Also, an authenticated user has the ability to reset the password of another user, in case it is forgotten or lost and users can be deleted, except the first pre-installed user.

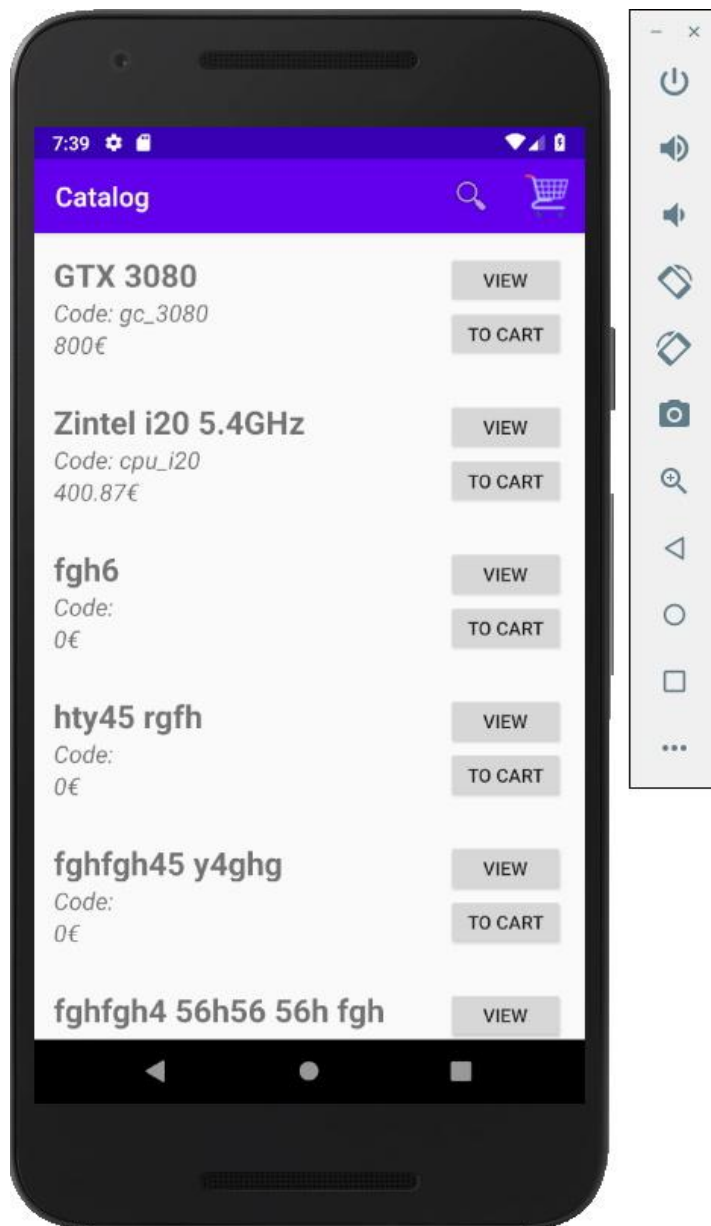
4.2 Retailer System

This chapter contains details about the sub-system of the retailer application. Starting with the technical ones, the retailer system is an Android mobile application, developed with Java 7 and minimum Android SDK or API Level 23. For data storage, it utilizes Android's native SQLite database service. Upon its startup, the application presents the user a menu with three options (Picture 15), the product catalogue, the order list and the vendor list.



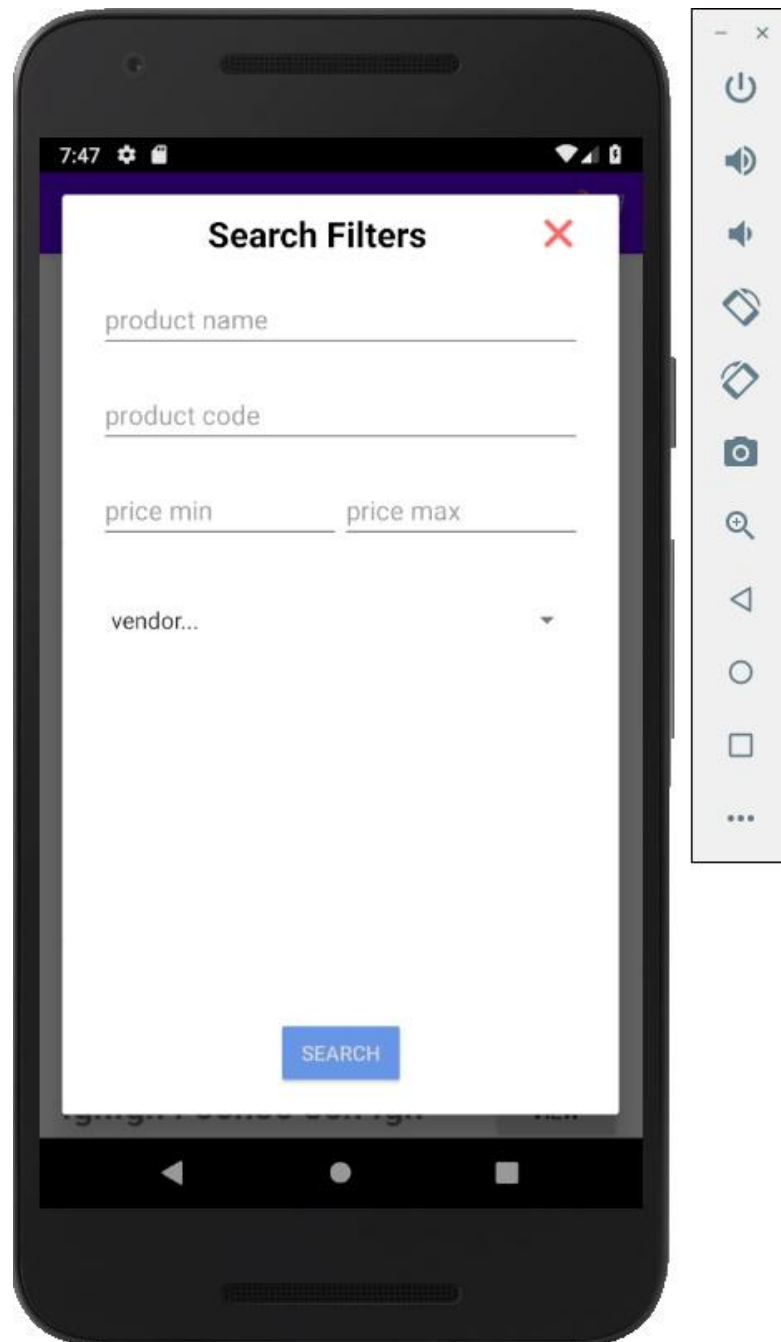
Picture 15: Retailer application - main screen.

Concerning the functionalities, first and foremost there is the product catalogue. The application implements an efficient endless scrolling list of products (Picture 16) from all the vendors that the retailer has a cooperation with. Not all products are loaded at once in the list, because having to download possibly a huge amount of product data from all the vendors, would take a long time and consume lots of resources of the mobile device. Instead, a small number of products is downloaded each time and when the list is about to finish with the next scroll downwards, a few more data will be downloaded.



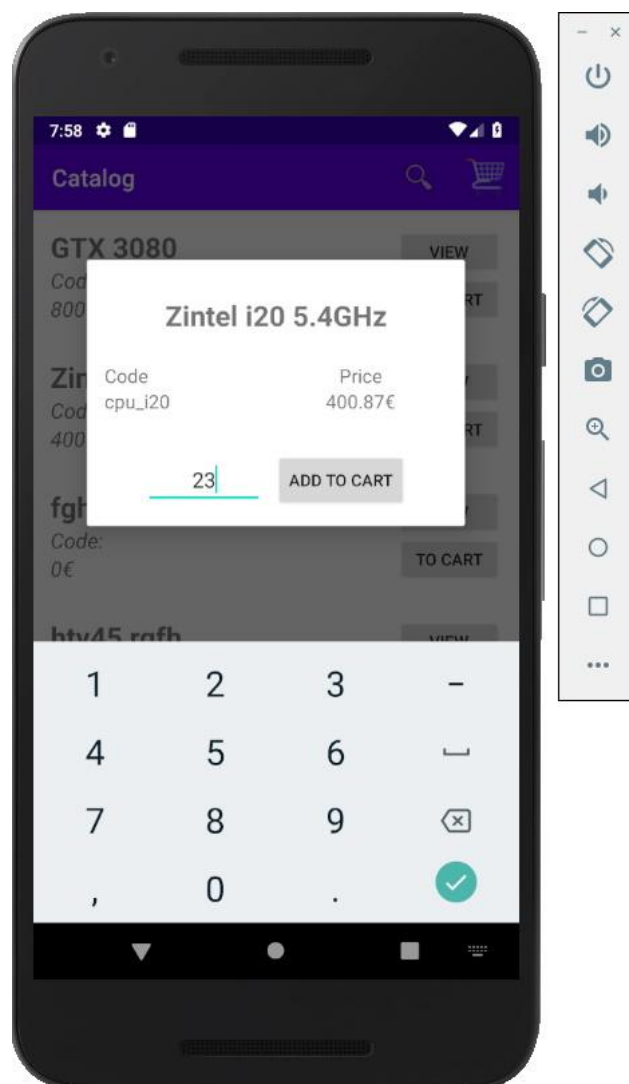
Picture 16: The endless scrolling list of products.

The user is also able to apply filters to find specific products. The filters include the selection of a specific vendor, the determination of a price range and the searching by title and/or code. They are all accessible via a specific popup dialog that appears immediately after the “Search” button has been pressed (Picture 17).



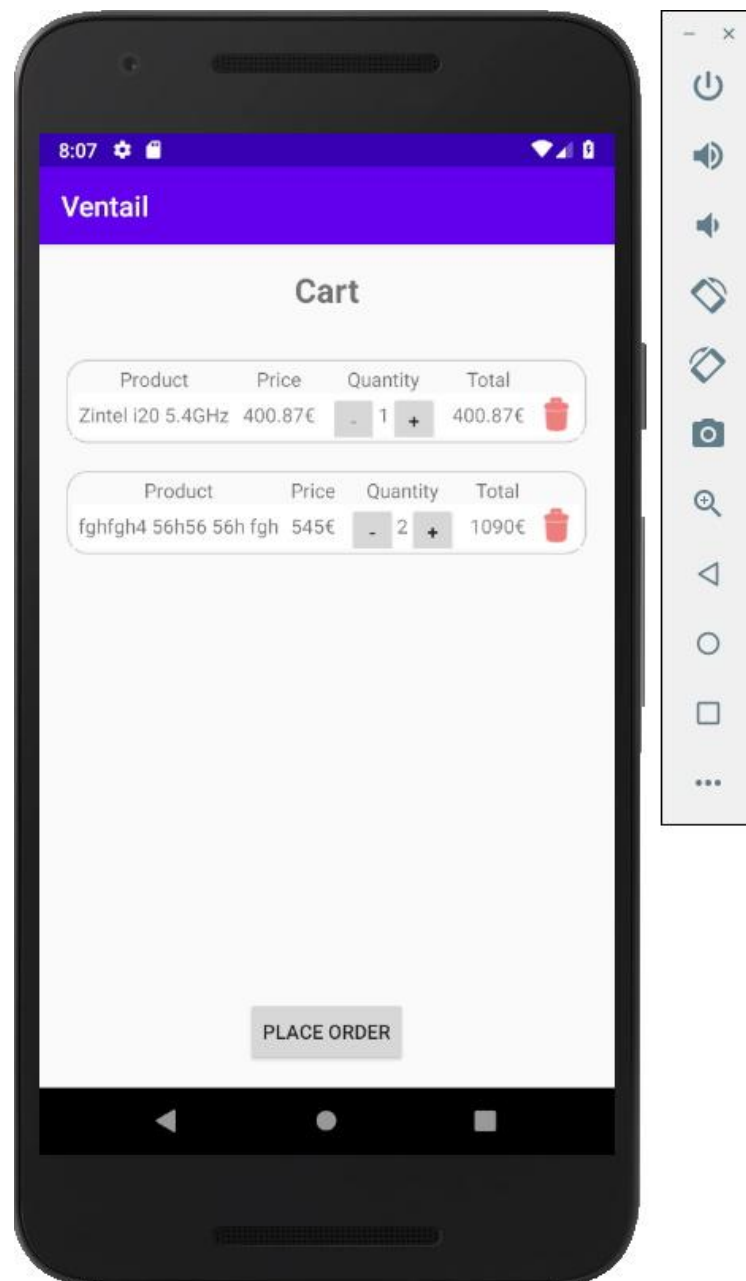
Picture 17: Search filters popup.

From the product catalog, there are two options for the retailer. The first one is “View Options” which shows a new screen with all the details of the product. The other one is the “Add to cart” option. The latter option is found inside the product details screen as well. By choosing the “Add to cart” button from the catalog, a popup dialog appears (Picture 18), showing the name, code and price of the product. It contains an input allowing the user to set the desired quantity for this product. Of course, there is a set of validity checks, that include a check if the quantity is a positive number and if it is not, then the confirmation button gets disabled and the product cannot be added to the cart.



Picture 18: Add to cart popup.

At any moment, while in the product catalog screen, the user can access the shopping cart (Picture 19) and view all the products that were added to it, change their quantities or even remove them completely from the cart. Additionally, some information about each product is displayed, such as the name of product, the price and the total cost accompanied by the price and the desired quantity.

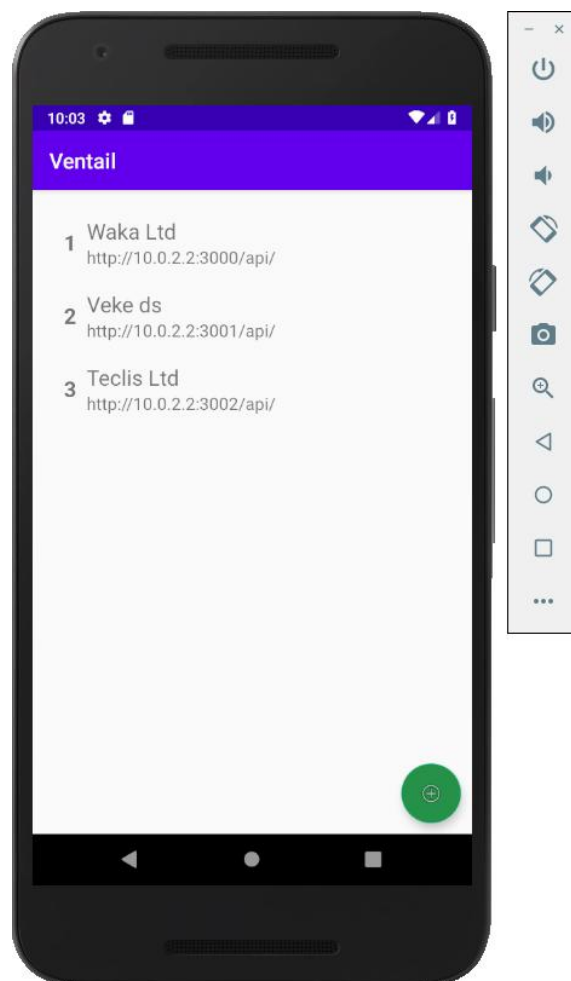


Picture 19: Shopping cart screen.

When all desired products for the resupply have been added to the cart (Picture 19), the retailer can press the “Place Order” button to send the order, without the need to make separate orders for each vendor. All products added to the cart could originate

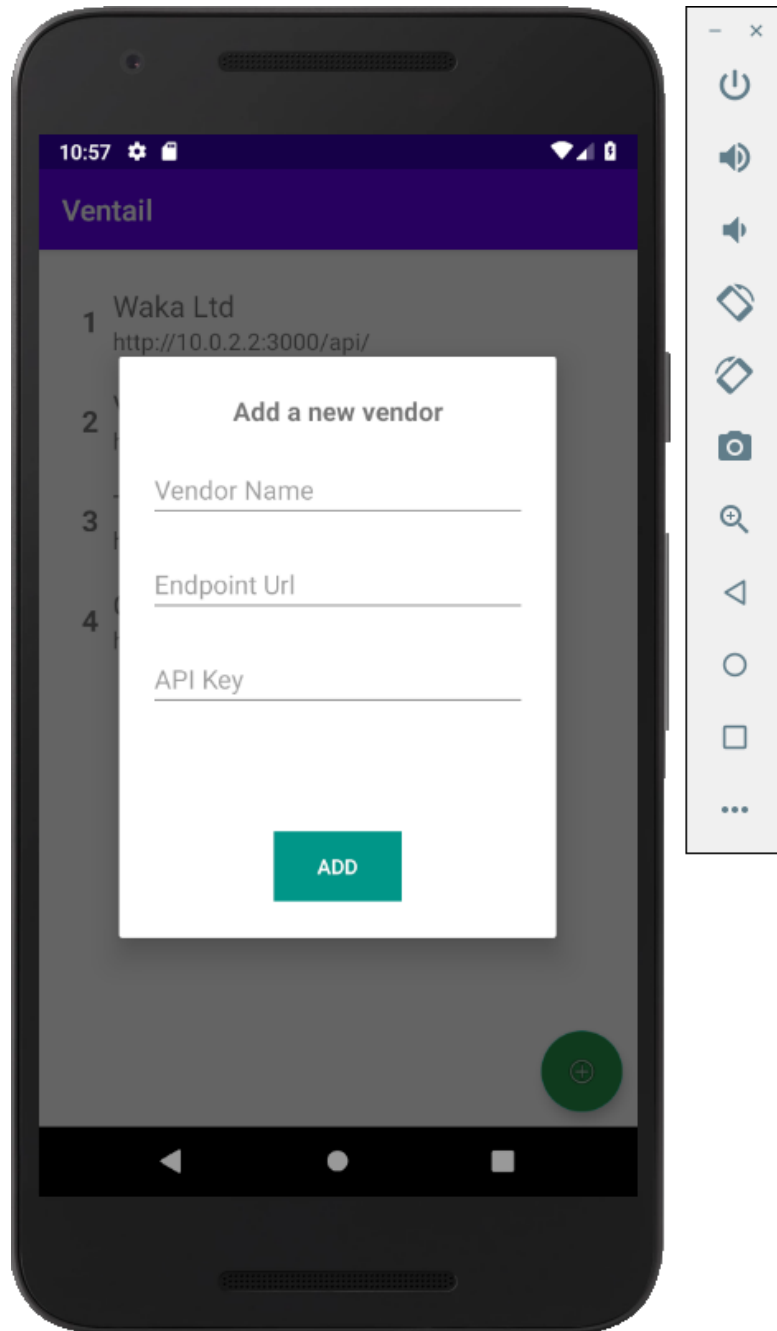
from many different vendors, yet the application implements a smart, automatic mechanism that creates sub-orders or partial orders for each vendor and groups the selected products into the appropriate sub-order. Then, each sub-order is sent to the appropriate vendor. This automatic mechanism provides speed and efficiency to the ordering process for the supplies.

One more feature that this system implements is vendor management. In this part of the system, a retailer is able to register all the collaborating vendors to the application. When the system downloads the product catalogue, it uses the list of registered vendors to find and fetch their respective product catalogues. Initially by pressing the “My Vendors” button in the main menu, the user is presented with a list of all registered vendors (Picture 20). The list contains some basic information about each vendor like the name and the address of the server that runs the vendor application.



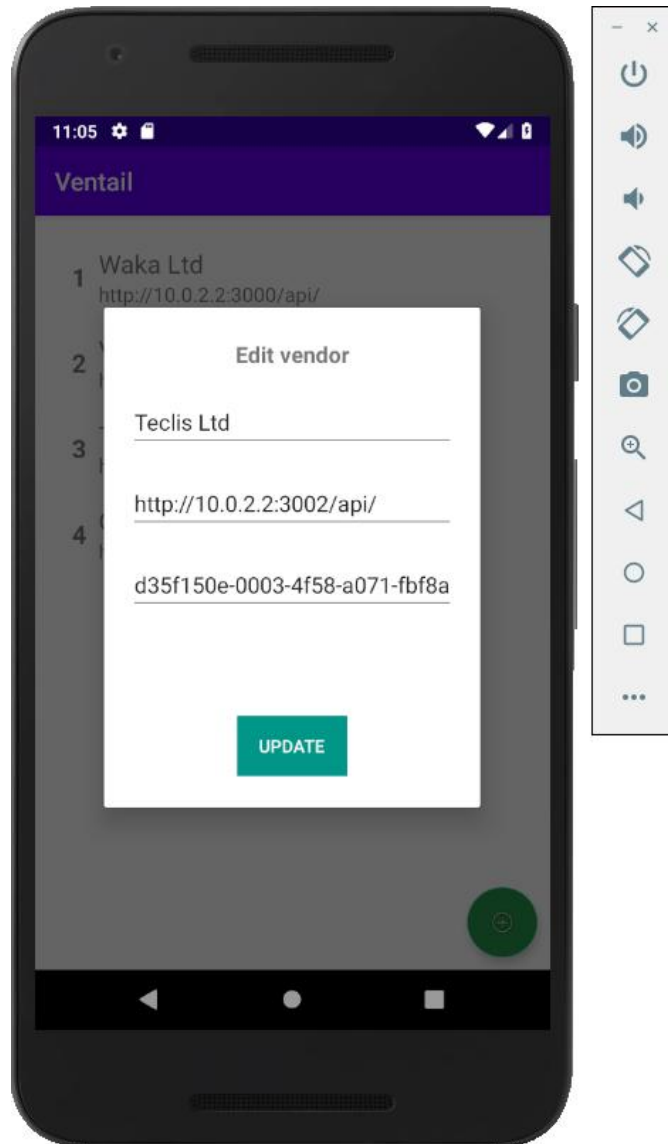
Picture 20: Vendors List

For the vendor registration, the user enters in the popup dialog (Picture 21) a descriptive name, the address of the server that runs the vendor system and the API key. The last two pieces of information are provided by the vendor. More details about them will be provided in Subsection 4.3.



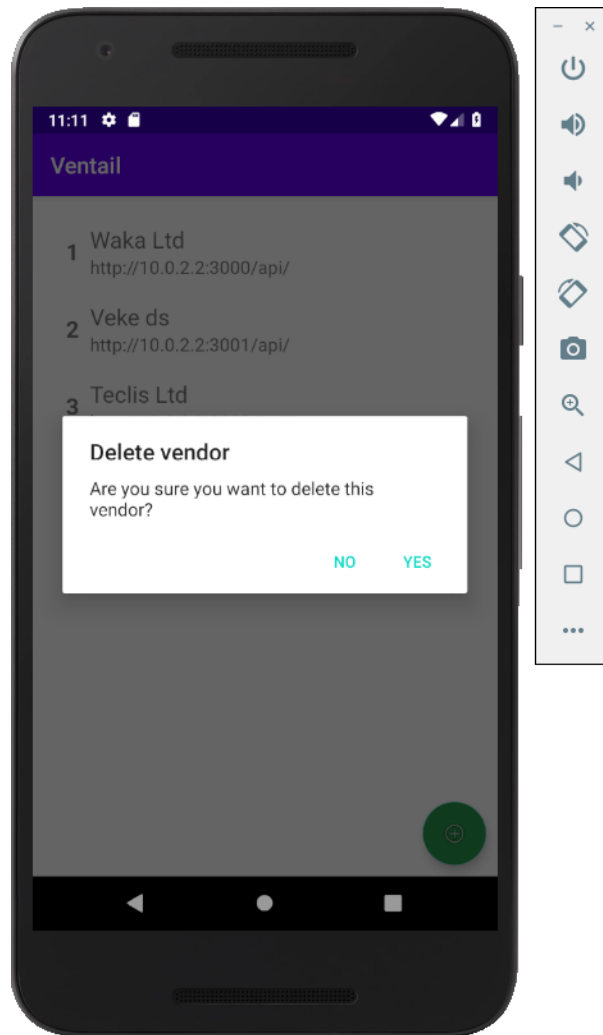
Picture 21: Vendors add popup dialog.

Additionally, the system allows a retailer, to edit and update the information of a specific vendor. By pressing the desired vendor item in the list, a popup dialog (Picture 22), like the one for the vendor addition, opens and all of its fields are filled with the saved data. Then, the user can change those values and save the changes permanently to the database, by pressing the “Update” button.



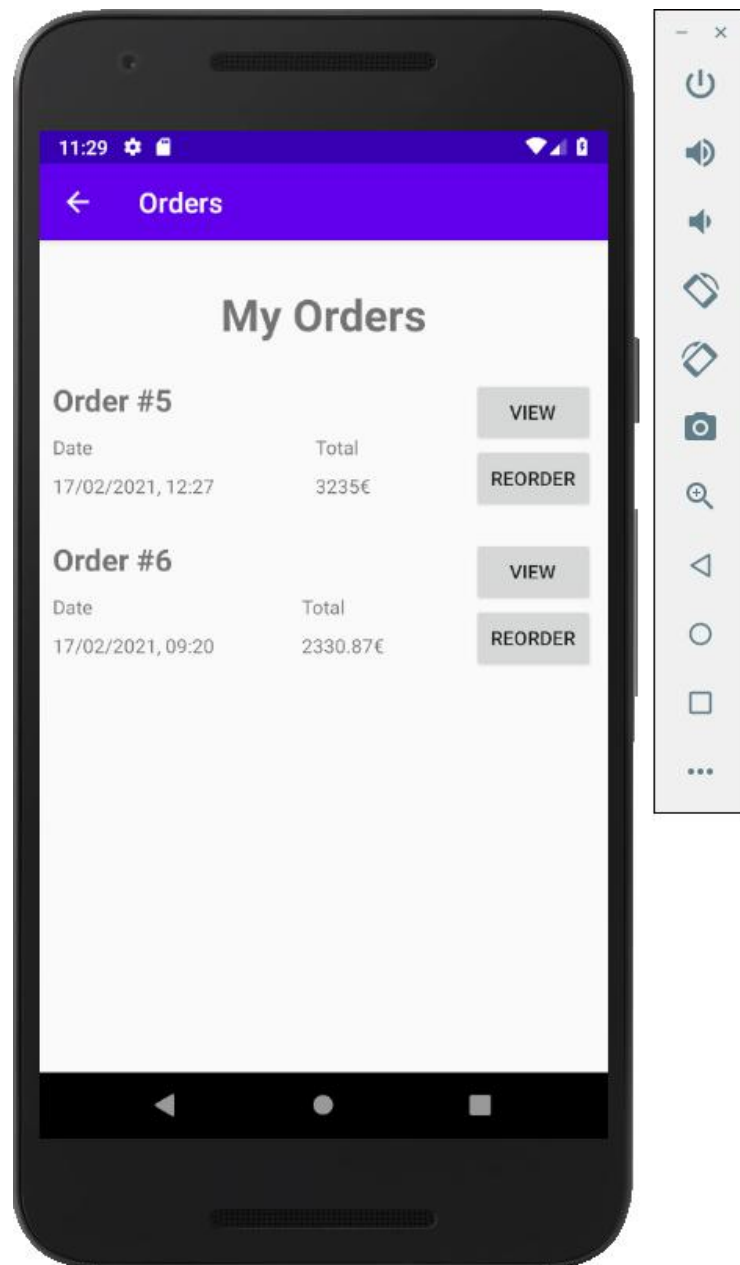
Picture 22: Vendor update popup dialog.

Finally, the deletion option is provided and the user needs to long press the list item of the desired vendor and then confirm the popup message (Picture 23) that appears. After confirming it, the vendor’s data will be deleted permanently from the database.



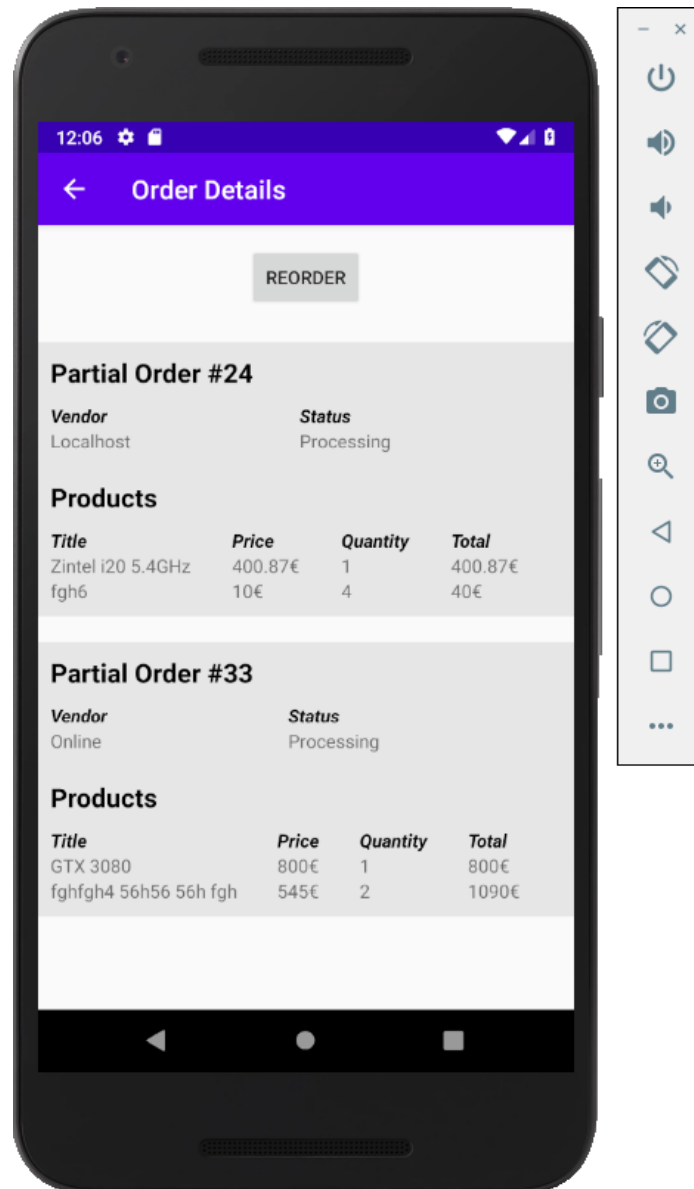
Picture 23: Vendor delete popup confirmation.

Last but not least, an equally important feature is the order list, where the retailer is presented with a list of all submitted orders (Picture 24). All information about these orders is stored locally in a database in the device. Those details have to do with the date that the order was placed and the total cost. Two additional options are provided for each order: the first one is a form that presents all the order details, whereas the second one is a “Reorder” button that creates a new order with the same products and submits it to the vendor.



Picture 24: Orders List

In the order details screen (Picture 25), the user may access a detailed list of all partial orders that were placed to each vendor through the selected order, as well as check all the products that were ordered along with their price and quantities. Another useful piece of information is the partial order status which shows the progress of the partial order from the vendor's side. The status is not stored locally in the device, because it can change at any time by the vendor, but it is rather downloaded every time this screen is shown. In the case of a connection failure with the vendor, the status is displayed with a dash "-".



Picture 25: Order's details list

Finally, with the reorder option, either from the order list or the order's details screen, the application downloads the products from the partial orders from each respective vendor and adds those products to the cart along with the quantities that they had in the selected order. It is necessary to re-download the products to clear any inconsistencies between the products, because a vendor might have changed the title or the price of a product that was recorded in a past order or he might have even deleted it. Thus, the products from the order are downloaded and added to the cart with the latest data and the retailer can verify them before proceeding to place the order.

4.3 Connection of the two Systems

As mentioned before, the two sub-systems are connected to each other through an Application Programming Interface (API) which is a sub-module of the vendor system. Essentially, the retailer application uses this API to get or send data through HTTP requests. This connection is unidirectional, from the retailer to the vendor, since the retailer application is the one that needs to either download data from the vendor or send data to the vendor, for an order for example. All the data the vendor needs to access is located in the same local server where the vendor application is running.

These HTTP requests cannot be performed simply by anyone, since there is a security layer that allows only authorized users to send or receive data through their connection. This security layer is accomplished from the customer registration, (mentioned at chapter 4.1), and the vendor registration, (mentioned at chapter 4.2). First, the vendor registers a customer and a unique API key is generated for this customer. Then, the vendor has to give this key along with the IP address or the domain of the server where his application is running. Finally, when registering a vendor, the retailer has to input these data and the retailer application will automatically include them for every HTTP request it sends to the specified vendor. This way, every retailer will have permission to send requests only to their vendors and not to any other vendor and also any malicious user outside of the system will not be able to send any requests at all, without having a valid API key even if they manage to find the IP address or domain of the server of a vendor.

5. System Evaluation

The evaluation of the entire system was realized with the participation of eight people in total. Three of them played the role of the vendor and five of them the role of the retailer. The vendors installed the vendor application into their hosting provider, while the retailer installed the retailer one into their smartphones. Each vendor registered a random number of retailers into their system as customers and the respective customers or retailers registered those vendors into their mobile application. All of them examined and used the two systems thoroughly for a week.

In terms of effectiveness, both the vendors and the retailers stated that the systems worked as expected and delivered what they were supposed to deliver. The vendors were able to easily manage their business-to-business product catalogue and the orders from their customers. On the other hand, the retailers mentioned that the application allowed them to efficiently resupply with the help of the endless scrolling list filled with products from all vendors and the process is consuming almost no time at all. More specifically, by using conventional methods they would need on average two hours to write down all the desired products for re-order, while with the retailer system they only need around five minutes. They can monitor the progress status of their orders and place again a previously stored order, which is very important for them since most of the times they tend to order the same supplies in the same quantities. Overall, they pointed out that before using Ventail, they had to spend up to three whole days to complete the resupply process by contacting each vendor, one after the other, wasting time until they understand each other. Though with the system it only takes one day and that is not only due to the fact that the retailer system reduces greatly the time wasted on the ordering process, but also the vendor system allows vendors to efficiently manage their orders

Regarding the communication speed, the results cannot be very objective, since speed mostly depends on variable factors, like the internet connection provided from an Internet Service Provider, the processing power capabilities of a vendor server or a client device that will render the application. Nevertheless, all three vendors pointed out that they experienced satisfactory speeds and each page was loading really fast, almost instantly. The fact that PHP was used as the building block of this system as well as the efficient pagination of all data led to high server-side processing speeds and lightweight HTML pages that can be rendered by any browser quickly. On the other side, three out of five retailers mentioned that the running speed of the application was good, while the other two stated that sometimes there were some delays. Those delays had to do with the product catalogue and the downloading process, so it is assumed that the problem lies in their internet connection. Either their Internet Service Provider has a low bandwidth or somewhere through the routing there were some unexpected delays, especially since with the Covid-19 quarantine the internet traffic is too high.

The final aspect of the entire system that was evaluated was the power consumption. This is possibly the most important one in the field of IoT devices, since IoT devices run on low-capacity batteries. This explains why the system was designed and developed to maximize power efficiency and resource management. Therefore, the goal was to minimize the battery consumption of the IoT devices that would use it. Of course, there is no point in measuring the vendor system for power consumption because it is a server-side processing application. In other words, it is executed by a server that is constantly plugged in to a stable power supply.

So, four out of five users were satisfied with the power efficiency of the retailer system and they mentioned that they did not have to charge extra times their smartphones while using the application for an entire week. This is due to the fact that the application is not using almost any power-hungry resources of the smartphone, like camera, GPS, or any other sensors. It only makes some HTTP requests to vendor systems, but only when needed, like when the user launches the product catalogue or places an order. The one user that had some noticeable battery drainage, probably had an older device with a lower battery capacity or even a damaged battery.

6. Conclusions

To summarize, the stock-out problem that retailers have to face is probably the most crucial one since it can lead to a severe loss of customers and therefore profits. The adoption of traditional ways of resupplying definitely renders the situation even worse. Therefore, it would be wise to incorporate information technologies and technologies related with Internet of Things into the supply process. This will boost the ordering capabilities and improve the time of the resupply process. That is exactly the goal of the proposed system, namely to provide retailers with an effective and time efficient way of ordering their supplies from many vendors at the same time and having the option to place again previously submitted orders, with the aim of reducing the wasted time. The aforementioned goal is achieved, according to the system evaluation, and the system provides not only an effective solution to the problem, but also a time and power efficient solution. There is definitely room for improvements and additions to the system. One of them would be the option for the vendor system to connect to the Enterprise Resource Planning (ERP) application of the vendor or the vendor's electronic store and automatically insert all products into the system. Another interesting point of improvement would be the implementation of the ability for the retailer system to connect to the ERP or electronic store of the retailer and automatically order exhausted or nearly exhausted products.

Bibliography

- Akritidis, L. Bozani, P., and Papadopoulos D. (2018). Efficient Urban Transportation(s) with IoT Devices and Robust Workers Allocation, *In Proceedings of 2018 South Eastern European Conference on Design Automation, Computer Engineering, Computer Networks and Social Media Conference (SEEDA-CECNSM)*, pp. 1-6.
- Corsten, D., & Gruen, T. (2003). Desperately seeking shelf availability: an examination of the extent, the causes, and the efforts to address retail out-of-stocks. *International Journal of Retail & Distribution Management*, 31(12), 605-617. doi:10.1108/09590550310507731
- Fernie, J., & Grant, D. (2008). On-shelf availability: the case of a UK grocery retailer. *The International Journal of Logistics Management*, 19(3), 293-308. doi:10.1108/09574090810919170
- Frontoni, E., Mancini, A., Zingaretti, P., Contigiani, M., Bello, L., & Placidi, V. (2016). Design and test of a real-time shelf out-of-stock detector system. *Microsystem Technologies*, 24(3), 1369-1377. doi:10.1007/s00542-016-3003-3
- Madakam, S., Ramaswamy, R., & Tripathi, S. (2015). Internet of Things (IoT): A Literature Review. *Journal of Computer and Communications*, 03(05), 164-173. doi:10.4236/jcc.2015.35021
- Maly, R. (2003, March). *Comparison of Centralized (Client-Server) and Decentralized (Peer-to-Peer) Networking*. Ανάκτηση December 13, 2020, από <https://pub.tik.ee.ethz.ch/students/2002-2003-Wi/SA-2003-16.pdf>
- Patel, K., & Patel, S. (2016). Internet of Things-IOT: Definition, Characteristics, Architecture, Enabling Technologies, Application & Future Challenges. *International Journal of Engineering Science and Computing*.
- Pizzi, G., & Scarpi, D. (2013). When Out-of-Stock Products DO Backfire: Managing Disclosure Time and Justification Wording. *Journal of Retailing*, 89(3), 352-359. doi:10.1016/j.jretai.2012.12.003
- Sklar, D. (2016). *Learning PHP: A Gentle Introduction to the Web's Most Popular Language*. O'Reilly Media.

- Tejesh, B., & Neeraja, S. (2018). Warehouse inventory management system using IoT and open source framework. *Alexandria Engineering Journal*, 57(4), 3817-3823.
doi:10.1016/j.aej.2018.02.003
- Verhoef, P., & Sloot, L. (2006). Out-of-Stock: Reactions, Antecedents, Management Solutions, and a Future Perspective. *Retailing in the 21st Century*, 239-253.
doi:10.1007/3-540-28433-8_16

Appendix A – Vendor System Source Code

Middleware Classes

In Laravel Framework, middleware classes provide a function that executes every time before or after the controller class that handles the main request. The vendor system uses two middleware classes for authentication purposes.

Middleware - Authenticate

```
1 <?php
2
3 namespace App\Http\Middleware;
4
5 use Illuminate\Auth\Middleware\Authenticate as Middleware;
6
7 class Authenticate extends Middleware
8 {
9     protected function redirectTo($request)
10    {
11        return route('login');
12    }
13 }
14
```

This middleware is used to block the access to the system to any unauthenticated request and redirects to the login page. It extends Laravel's authentication system and overrides the redirectTo function, which is called before any processing of an incoming request occurs and returns a route to the controller that renders the login page, if there is no logged in user.

Middleware - ApiAuthenticate

```
1 <?php
2
3 namespace App\Http\Middleware;
4
5 use Closure;
6 use App\Models\Customers;
7
8 class ApiAuthenticate
9 {
10     public function handle($request, Closure $next)
11     {
12         if ( !$request->key ) {
13             return abort(403);
14         }
15
16         if (Customers::where('api_key', $request->key)->where('validated', 1)->
17             doesntExist()) {
18             return abort(401);
19         }
20
21         return $next($request);
22     }
23 }
```

Similarly with the previous one, this middleware blocks any unauthorized access to the system through the API that connects the retailer to the vendor system. First of all, it is checked if the request includes a parameter for the API key, if not then a 403 HTTP error is returned. Then, if the key parameter exists, it is checked whether or not there is a customer record with this API key in the database. If there is, then the request is processed normally, else a 401 HTTP error is returned.

Controller Classes

Controller classes in Laravel are responsible for processing incoming HTTP requests and either render a webpage, access the database and retrieve some data, modify, insert or delete records from the database, process files and anything in general that has to do with server-side processing.

AuthController

```
1 <?php
2
3 namespace App\Http\Controllers;
4
5 use App\Http\Controllers\Controller;
6 use Illuminate\Http\Request;
7 use Illuminate\Support\Facades\Auth;
8
9 class AuthController extends Controller
10 {
11
12     public function login_page(Request $request){
13         if(!Auth::check())
14             return view('login');
15         else
16             return redirect()->intended(route('index'));
17     }
18
19     public function do_login(Request $request){
20
21         $credentials = $request->only('username', 'password');
22
23         if (Auth::attempt($credentials)) {
24             return redirect()->intended(route('index'));
25         }
26         else{
27             $flashdata = ['type'=>'danger', 'text'=>'Wrong username or password. Try again.'];
28             return back()->with('message', $flashdata);
29         }
30     }
31
32     public function do_logout(Request $request){
33         Auth::logout();
34         return redirect()->intended('login');
35     }
36 }
```

This controller handles anything that has to do with the user authentication of the application.

- The `login_page` function is responsible to render the login page, if there is no logged in user, or in the other case it redirects to the home page.
- The `do_login` function handles the HTTP post request that is sent when the user presses the login button in the login form. It checks whether the credentials are correct to allow access or return an error message.
- The `do_logout` function releases the session variable that keeps track whether there is a logged in user and redirects to the login page, in other words it logs out the user.

HomeController

```
1  <?php
2
3  namespace App\Http\Controllers;
4
5  use App\Http\Controllers\Controller;
6  use App\Models\OrderStatus;
7  use App\Models\Products;
8  use App\Models\Customers;
9
10 class HomeController extends Controller
11 {
12     /**
13      * Index of the Home page.
14      *
15      * @return \Illuminate\View\View
16      */
17     public function index()
18     {
19         $status = OrderStatus::all();
20         $products = Products::count();
21         $customers = Customers::count();
22
23         $cust = Customers::all();
24         $earnings = 0;
25         foreach ($cust as $c) {
26             $earnings += $c->total_spend;
27         }
28
29         return view('home_index', [
30             'status'=>$status,
31             'products'=>$products,
32             'customers'=>$customers,
33             'earnings'=>$earnings
34         ]);
35     }
36 }
```

This controller is responsible for rendering the dashboard or home page of the vendor application. It calculates the total number of products, customers and orders, grouped by order status, and the total earning and passes these data to the view file which will be rendered on screen.

ProductsController

```
1 <?php
2
3 namespace App\Http\Controllers;
4
5 use App\Http\Controllers\Controller;
6
7 use App\Models\Products;
8 use Illuminate\Http\Request;
9
10 class ProductsController extends Controller
11 {
12     private $page_title = 'Products';
13
14     /**
15      * Index of the Home page.
16      *
17      * @return \Illuminate\View\View
18      */
19     public function index(Request $request)
20     {
21         $query = Products::on();
22         $query = $this->parseFilters($request, $query);
23         $products = $query->paginate(10);
24
25         return view('products.index', [
26             'page_title'=>$this->page_title,
27             'products'=>$products
28         ]);
29     }
30
31     public function create()
32     {
33         return view('products.create', ['action'=>route('products_submit_create')]);
34     }
35
36     public function submit_create(Request $request){
37         $data = $results = array_filter($request->input(), 'strlen');
38         unset($data['_token']);
39
40         $product = Products::create($data);
41         $product->save();
42
43         return redirect()->route('products');
44     }
45
46     public function edit(Request $request, $id)
47     {
48         $product = Products::find($id);
49         return view('products.edit', [
50             'product'=>$product,
51             'action'=>route('products_submit_edit', ['id'=>$id])
52         ]);
53     }
54
55     public function submit_edit(Request $request, $id){
56         $data = $results = $request->input();
57         unset($data['_token']);
58
59         Products::find($id)->update($data);
60
61         echo '<script>window.history.back()</script>';
62         exit;
63     }
64 }
```

```

64
65 public function delete(Request $request, $id){
66     $product = Products::withCount('orders')->find($id);
67     $flashdata = ['type'=>'success', 'text'=>'Product has been deleted successfully.'];
68
69     if(!$product)
70         $flashdata = ['type'=>'danger', 'text'=>'There is not such a product.'];
71     else{
72
73         if($product->orders_count > 0){
74             $flashdata = ['type'=>'danger', 'text'=>'There is one or more orders associated with this product.'];
75         }
76         else{
77             $product->delete();
78         }
79     }
80
81     return back()->with('message', $flashdata);
82 }
83
84 private function parseFilters(Request $request, $query){
85     if($request->filled('id.min')){
86         $query->where('id', '>=', $request->input('id.min'));
87     }
88     if($request->filled('id.max')){
89         $query->where('id', '<=', $request->input('id.max'));
90     }
91
92     if($request->filled('title')){
93         $query->where('title', 'like', '%'.$request->input('title').'%');
94     }
95
96     if($request->filled('product_code')){
97         $query->where('product_code', 'like', $request->input('product_code').'%');
98     }
99
100    if($request->filled('price.min')){
101        $query->where('price', '>=', $request->input('price.min'));
102    }
103    if($request->filled('price.max')){
104        $query->where('price', '<=', $request->input('price.max'));
105    }
106
107    if($request->filled('stock.min')){
108        $query->where('stock', '>=', $request->input('stock.min'));
109    }
110    if($request->filled('stock.max')){
111        $query->where('stock', '<=', $request->input('stock.max'));
112    }
113
114    if($request->filled('enabled')){
115        $query->where('enabled', '=', $request->input('enabled'));
116    }
117
118    if($request->filled('limit')){
119        $query->limit($request->input('limit'));
120    }
121
122    if($request->filled('offset')){
123        $query->offset($request->input('offset'));
124    }
125    return $query;
126 }
127 }

```

ProductsController deals with the product management requests of the system.

- The index function handles the product list rendering. The products are collected from the database, with possible user requested filters applied on the query, and the results are paginated through Laravel's pagination functionality.
- The create function simply renders the page for creating a new product.

- The submit_create function handles the form submission to create a new product. A new product record is inserted in the database with the data extracted from the request.
- The edit function renders the edit page for a specific product. A product id is passed as a parameter from the request url and it is used to retrieve the record from the database. Then the record is passed in the view file which renders the edit page filled with the product information.
- The submit_edit function handles the form submission request to update the data of a specific product. The record of the product in the database is updated with the data extracted from the request.
- The delete function removes a product record from the database. A product id is passed as a parameter from the request url and it is used to delete the specified product record.
- The parseFilters function is executed inside the index function and it is responsible for applying the user requested filters on the product listing query. For each possible filter, it is checked if it is present in the request and if so, the appropriate SQL clause is appended to the query which will be executed later in the index function.

CustomersController

```
1 <?php
2
3 namespace App\Http\Controllers;
4
5 use App\Http\Controllers\Controller;
6
7 use App\Models\Customers;
8 use Illuminate\Http\Request;
9
10 class CustomersController extends Controller
11 {
12     private $page_title = 'Customers';
13
14     /**
15      * Index of the Home page.
16      *
17      * @return \Illuminate\View\View
18      */
19     public function index(Request $request)
20     {
21         $query = Customers::on();
22         $query = $this->parseFilters($request, $query);
23         $customers = $query->paginate(10);
24
25         return view('customers.index', [
26             'page_title'=>$this->page_title,
27             'customers'=>$customers
28         ]);
29     }
30
31     public function create()
32     {
33         return view('customers.create', ['action'=>route('customers_submit_create')]);
34     }
35
36     public function submit_create(Request $request){
37         $data = $results = array_filter($request->input(), 'strlen');
38         unset($data['_token']);
39
40         $customer = Customers::create($data);
41         $customer->save();
42
43         return redirect()->route('customers');
44     }
45 }
```



```

46 public function edit(Request $request, $id)
47 {
48     $customer = Customers::find($id);
49     return view('customers.edit', [
50         'customer'=>$customer,
51         'action'=>route('customers_submit_edit', ['id'=>$id])
52     ]);
53 }
54
55 public function submit_edit(Request $request, $id){
56     $data = $results = $request->input();
57     unset($data['_token']);
58
59     Customers::find($id)->update($data);
60
61     echo '<script>window.history.back()</script>';
62     exit;
63 }
64
65 public function delete(Request $request, $id){
66     $customer = Customers::withCount('orders')->find($id);
67     $flashdata = ['type'=>'success', 'text'=>'Customer has been deleted successfully.'];
68
69     if(!$customer)
70         $flashdata = ['type'=>'danger', 'text'=>'Customer not found.'];
71     else{
72         if($customer->orders_count > 0){
73             $flashdata = ['type'=>'danger', 'text'=>'There is one or more orders associated with this customer.'];
74         }
75         else{
76             $customer->delete();
77         }
78     }
79
80     return back()->with('message', $flashdata);
81 }
82
83 private function parseFilters(Request $request, $query){
84     if($request->filled('id.min')){
85         $query->where('id', '>=', $request->input('id.min'));
86     }
87     if($request->filled('id.max')){
88         $query->where('id', '<=', $request->input('id.max'));
89     }
90
91     if($request->filled('company_name')){
92         $query->where('company_name', 'like', '%'.$request->input('company_name').'%');
93     }
94
95     if($request->filled('validated')){
96         $query->where('validated', '=', $request->input('validated'));
97     }
98
99     if($request->filled('api_key')){
100         $query->where('api_key', 'like', $request->input('api_key').'%');
101     }
102
103     return $query;
104 }
105 }

```

CustomersController deals with the customer management of Ventail system.

- The index function handles the customer list rendering. Customer records are collected from the database, with possible user requested filters applied on the query, and the results are paginated through Laravel's pagination functionality.
- The create function simply renders the page that allows a user to register a new customer.
- The submit_create function handles the form submission request to register a new customer. A new customer record is inserted in the database with the data extracted from the request.

- The edit function renders the edit page for a specific customer. A customer id is passed as a parameter from the request url and it is used to retrieve the record from the database. Then the record is passed in the view file which renders the edit page filled with the customer information.
- The submit_edit function handles the form submission request to update the data of a specific customer. The record of the customer in the database is updated with the data extracted from the request.
- The delete function removes a customer record from the database. A customer id is passed as a parameter from the request url and it is used to delete the specified record.
- The parseFilters function is executed inside the index function and it is responsible for applying the user requested filters on the customer listing query. For each possible filter, it is checked if it is present in the request and if so, the appropriate SQL clause is appended to the query which will be executed later in the index function.

OrdersController

```
1 <?php
2
3 namespace App\Http\Controllers;
4
5 use App\Http\Controllers\Controller;
6
7 use App\Models\Orders;
8 use App\Models\OrderStatus;
9 use Illuminate\Http\Request;
10
11 class OrdersController extends Controller
12 {
13     private $page_title = 'Orders';
14
15     /**
16      * Index of the Home page.
17      *
18      * @return \Illuminate\View\View
19      */
20     public function index(Request $request)
21     {
22         $query = Orders::withSum('products as total_cost', \DB::raw('price * order_item.quantity'));
23         $this->parseFilters($request, $query);
24         $orders = $query->paginate(10)->withQueryString();
25
26         return view('orders.index', [
27             'page_title'=>$this->page_title,
28             'orders'=>$orders
29         ]);
30     }
31
32     public function view(Request $request, $id)
33     {
34         $order = Orders::withSum('products as total_cost', \DB::raw('price * order_item.quantity'))->find($id);
35         $statuses = OrderStatus::all();
36
37         return view('orders.view', [
38             'order'=>$order,
39             'statuses'=>$statuses,
40             'action'=>route('orders_submit_status', ['id'=>$id])
41         ]);
42     }
43
44     public function submit_status(Request $request, $id)
45     {
46         $flashdata = ['type'=>'success', 'text'=>'Order status update successful!'];
47
48         //check if status request parameter exists
49         if(!$request->filled('status')){
50             $flashdata = ['type'=>'danger', 'text'=>'Please select an order status.'];
51         }
52         else {
53             //prevent new status being equal with existing one
54             $order = Orders::where('status','<>', $request->status)->find($id);
55             if(!$order){
56                 $flashdata = ['type'=>'danger', 'text'=>'Please select a different order status.'];
57             }
58             else {
59                 switch ($request->status) {
60                     case 1:
61                         if( !$this->reverseOrder($order, 1) )
62                             $flashdata = ['type'=>'danger', 'text'=>'A server error occurred while updating order status.'];
63                         break;
64                     case 2:
65                         if($order->can_be_completed){
66                             if( !$this->completeOrder($order) )
67                                 $flashdata = ['type'=>'danger', 'text'=>'A server error occurred while updating order status.'];
68                         }
69                         else{
70                             $flashdata = ['type'=>'danger', 'text'=>'Insufficient stock for one or more products.'];
71                         }
72                         break;
73                     case 3:
74                         if( !$this->reverseOrder($order, 3) )
75                             $flashdata = ['type'=>'danger', 'text'=>'A server error occurred while updating order status.'];
76                         break;
77                     default:
78                         $flashdata = ['type'=>'danger', 'text'=>'Please select a valid order status.'];
79                 }
80             }
81         }
82     }
83
84     return back()->with('message', $flashdata);
85 }
```

```

87 private function completeOrder($order){
88     $success = false;
89     try {
90         \DB::beginTransaction();
91         foreach ($order->products as $product) {
92             $product->stock = $product->stock - $product->order_item->quantity;
93             $product->save();
94         }
95
96         $order->status = 2;
97         $order->save();
98
99         \DB::commit();
100        $success = true;
101    } catch (Exception $e) {
102        \DB::rollBack();
103    }
104    return $success;
105 }
106
107 private function reverseOrder($order, $newStatus){
108     $success = false;
109     try {
110         \DB::beginTransaction();
111
112         //must renew the stock of the products
113         if( $order->status == 2 ){
114             foreach ($order->products as $product) {
115                 $product->stock = $product->stock + $product->order_item->quantity;
116                 $product->save();
117             }
118         }
119
120         $order->status = $newStatus;
121         $order->save();
122
123         \DB::commit();
124         $success = true;
125     } catch (Exception $e) {
126         \DB::rollBack();
127     }
128
129     return $success;
130 }

```

```

132 private function parseFilters(Request $request, $query){
133
134     if($request->filled('id.min')){
135         $query->where('id', '>=', $request->input('id.min'));
136     }
137     if($request->filled('id.max')){
138         $query->where('id', '<=', $request->input('id.max'));
139     }
140
141     if($request->filled('customer_company_name')){
142         $query->whereHas('customer', function($subquery) use ($request) {
143             return $subquery->where('company_name', 'like', '%'.$request->input('customer_company_name').'%');
144         });
145     }
146
147     if($request->filled('date.min')){
148         $date = new \DateTime($request->input('date.min'));
149         $date->setTime(0,0,0);
150         $query->where('date', '>=', $date);
151     }
152
153     if($request->filled('date.max')){
154         $date = new \DateTime($request->input('date.max'));
155         $date->setTime(23,59,59);
156         $query->where('date', '<=', $date);
157     }
158
159     if($request->filled('order_status')){
160         $query->where('status', $request->input('order_status'));
161     }
162
163
164
165
166     if($request->filled('total_cost.min')){
167         $query->havingRaw('total_cost >= ?', [$request->input('total_cost.min')]);
168     }
169
170     if($request->filled('total_cost.max')){
171         $query->havingRaw('total_cost <= ?', [$request->input('total_cost.max')]);
172     }
173
174     return $query;
175 }
176 }

```

This controller is responsible for handling any request that has to do with the order management of the system.

- The index function handles the order list rendering. Order records are collected from the database, with possible user requested filters applied on the query, and the results are paginated through Laravel's pagination functionality. The results contain also the total cost of each order.
- The view function renders the view page for a specific order. An order id is passed as a parameter from the request url and it is used to retrieve the record from the database. Then the record is passed in the view file which renders the view page filled with the order information. Additionally, the order status data are retrieved and passed into the view file in order to render the appropriate status name based on the status id of the order.
- The submit_status function handles the form submission request from the user to change and update the status of a specified order. A few checks have to be made before the database record gets updated. If the current order status is Delivered

and it is requested to change to either Processing or Canceled, the quantity of each product from the order has to be added in the current stock of the respective product. On the other hand, if the current order status is not Delivered and it is requested to change to Delivered, first it has to be checked if the order can be completed and then the stock has to be decreased by the amount that is presented in the order. If the forementioned restrictions are satisfied, the order status will be updated successfully. If not, an error message will be returned.

- The `completeOrder` function changes the status of a specific order to Delivered. The stock of all products from this order is decreased by the amount of their order quantity. Finally, the order record is updated with the new status.
- The `reverseOrder` function updates the status of a specified order. Also, it is checked if the current status is already Delivered to reverse the stock of each product presented in the order.
- The `parseFilters` function is executed inside the `index` function and it is responsible for applying the user requested filters on the order listing query. For each possible filter, it is checked if it is present in the request and if so, the appropriate SQL clause is appended to the query which will be executed later in the `index` function.

UserController

```
1  <?php
2
3  namespace App\Http\Controllers;
4
5  use App\Http\Controllers\Controller;
6
7  use App\Models\User;
8  use Illuminate\Http\Request;
9
10 class UsersController extends Controller
11 {
12     private $page_title = 'Users';
13
14     /**
15      * Index of the Home page.
16      *
17      * @return \Illuminate\View\View
18      */
19     public function index(Request $request)
20     {
21         $query = User::on();
22         $query = $this->parseFilters($request, $query);
23
24         $query->where('id', '<>', 1); //except default admin
25
26         $users = $query->paginate(10);
27
28         return view('users.index', [
29             'page_title' => $this->page_title,
30             'users' => $users
31         ]);
32     }
33
34     public function create()
35     {
36         return view('users.create', ['action' => route('users_submit_create')]);
37     }
38
39     public function submit_create(Request $request){
40         $data = array_filter($request->all(), 'strlen');
41         unset($data['_token']);
42
43         if( isset($data['username']) ){
44             $user = User::where("username", $data['username'])->first();
45             if( $user ){
46                 $flashdata = [
47                     'type' => 'danger',
48                     'text' => 'User with username "'.$data['username'].'" already exists. Please select another one.'
49                 ];
50                 return back()->with('message', $flashdata);
51             }
52         }
53
54         $data['password'] = Hash::make($data['password']);
55         $user = User::create($data);
56         $user->save();
57
58         return redirect()->route('users');
59     }
60
61     public function edit(Request $request, $id)
62     {
63         $user = User::find($id);
64         return view('users.edit', [
65             'user' => $user,
66             'action' => route('users_submit_edit', ['id' => $id])
67         ]);
68     }
69 }
```

```

70 public function submit_edit(Request $request, $id){
71     $data = array_filter($request->all(), 'strlen');
72     unset($data['_token']);
73
74     $user = User::find($id);
75     if( isset($data['username']) && $data['username'] != $user->username ){
76         $userExists = User::where("username", $data['username']->first());
77         if( $userExists ){
78             $flashdata = [
79                 'type'=>'danger',
80                 'text'=>'User with username "'.$data['username'].'" already exists. Please select another one.'
81             ];
82             return back()->with('message', $flashdata);
83         }
84     }
85
86     if( isset($data['password']) && !empty($data['password'])){
87         $data['password'] = \Hash::make($data['password']);
88     }
89
90     $user->update($data);
91
92     return back();
93 }
94
95 public function delete(Request $request, $id){
96     $user = User::find($id);
97     $flashdata = ['type'=>'success', 'text'=>'User has been deleted successfully.'];
98
99     if(!$user)
100         $flashdata = ['type'=>'danger', 'text'=>'User not found.'];
101     else{
102         $user->delete();
103         if($user->id == \Auth::id()){
104             return redirect()->route('login');
105         }
106     }
107
108     return back()->with('message', $flashdata);
109 }
110
111 private function parseFilters(Request $request, $query){
112     if($request->filled('id.min')){
113         $query->where('id', '>=', $request->input('id.min'));
114     }
115     if($request->filled('id.max')){
116         $query->where('id', '<=', $request->input('id.max'));
117     }
118
119     if($request->filled('company_name')){
120         $query->where('company_name', 'like', '%'.$request->input('company_name').'%');
121     }
122
123     if($request->filled('validated')){
124         $query->where('validated', '=', $request->input('validated'));
125     }
126
127     if($request->filled('api_key')){
128         $query->where('api_key', 'like', $request->input('api_key').'%');
129     }
130
131     return $query;
132 }
133 }

```

UserController deals with the user management of the vendor system.

- The index function handles the user list rendering. User records are collected from the database, with possible requested filters applied on the query, and the results are paginated through Laravel's pagination functionality.
- The create function simply renders the page that allows the creation of a user for the system.

- The submit_create function handles the form submission request to create a new user. A new user record is inserted in the database with the data extracted from the request if the username presented in the request does not exist in the database. Also, the password gets hashed and the hash is stored in the database instead of the clear-text password.
- The edit function renders the edit page for a specific user. A user id is passed as a parameter from the request url and it is used to retrieve the record from the database. Then the record is passed in the view file which renders the edit page filled with the user information.
- The submit_edit function handles the form submission request to update the data of a specific user. The record of the user in the database is updated with the data extracted from the request, only if the provided username is the same as before or different from any other record. Furthermore, if a password is provided, it will be hashed and the hash will replace the old one in the database.
- The delete function removes a user record from the database. A user id is passed as a parameter from the request url and it is used to delete the specified record.
- The parseFilters function is executed inside the index function and it is responsible for applying any requested filter on the user listing query. For each possible filter, it is checked if it is present in the request and if so, the appropriate SQL clause is appended to the query which will be executed later in the index function.

ApiController

```
1 <?php
2
3 namespace App\Http\Controllers;
4
5 use App\Http\Controllers\Controller;
6 use Illuminate\Support\Facades\DB;
7 use App\Models\Products;
8 use App\Models\Customers;
9 use App\Models\Orders;
10 use App\Models\OrderItem;
11 use Illuminate\Http\Request;
12
13 use Illuminate\Support\Facades\Log;
14
15 class ApiController extends Controller
16 {
17     private $console;
18
19     public function __construct(){
20         $this->console = new \Symfony\Component\Console\Output\ConsoleOutput();
21     }
22
23     public function map_request(Request $request){
24         switch ($request->action) {
25             case 'catalog':
26                 return $this->products($request);
27             case 'place_order':
28                 return $this->place_order($request);
29             case 'order_status':
30                 return $this->get_order_status($request);
31             case 'products':
32                 return $this->get_products($request);
33             default:
34                 return response("Unknown action",422);
35         }
36     }
37
38     private function products(Request $request)
39     {
40         $query = Products::on();
41         $query = $this->parseFilters($request, $query);
42         $products = $query->get();
43
44         return response()->json($products);
45     }
46 }
```

```

47 private function place_order(Request $request){
48     $customer = Customers::where('api_key', $request->key)->first();
49     $products = $request->products;
50
51     abort_if( !$products // !is_array($products) // count($products) <= 0 , 422 );
52
53     try {
54         DB::beginTransaction();
55         $order = new Orders(['customer_id'=>$customer->id, 'status'=>1]);
56         $order->save();
57
58         foreach ($products as $p) {
59             $product = Products::find($p['product_id']);
60             if($product){
61                 $order->products()->save($product, [
62                     'quantity'=>$p['quantity'],
63                     'product_title'=>$product->title,
64                     'product_price'=>$product->price
65                 ]);
66             }
67         }
68
69         DB::commit();
70         return response($order->id ,200);
71     } catch (\Exception $ex) {
72         DB::rollback();
73         return response()->json(['error' => $ex->getMessage()], 500);
74     }
75 }
76
77 private function get_order_status(Request $request){
78     $customer = Customers::where('api_key', $request->key)->first();
79
80     abort_if( !$request->filled('order_id') , 422 );
81     $order = $customer->orders()->find($request->order_id);
82     abort_if( !$order , 404 );
83     return response($order->order_status->name,200);
84 }
85

```

```

86 private function get_products(Request $request)
87 {
88     abort_if( !$request->filled('ids') , 422 );
89
90     $ids = explode(",", $request->ids);
91     abort_if( !$ids // !is_array($ids) // count($ids) <= 0 , 422 );
92
93     $products = Products::whereIn("id", $ids)
94                 ->where('enabled', '=', 1)
95                 ->get();
96
97     return response()->json($products);
98 }
99
100 private function parseFilters(Request $request, $query){
101     if($request->filled('id.min')){
102         $query->where('id', '>=', $request->input('id.min'));
103     }
104     if($request->filled('id.max')){
105         $query->where('id', '<=', $request->input('id.max'));
106     }
107
108     if($request->filled('title')){
109         $query->where('title', 'like', '%'.$request->input('title').'%');
110     }
111
112     if($request->filled('product_code')){
113         $query->where('product_code', 'like', $request->input('product_code').'%');
114     }
115
116     if($request->filled('price_min')){
117         $query->where('price', '>=', $request->input('price_min'));
118     }
119     if($request->filled('price_max')){
120         $query->where('price', '<=', $request->input('price_max'));
121     }
122
123     $query->where('enabled', '=', 1);
124
125     if($request->filled('limit')){
126         $query->limit($request->input('limit'));
127     }
128
129     if($request->filled('offset')){
130         $query->offset($request->input('offset'));
131     }
132     return $query;
133 }
134 }

```

ApiController handles any request from the retailer system.

- The map_request function maps any incoming request to the appropriate private function of the ApiController that is capable of handling the request. In order for the mapping to take place, a parameter named action must be present in the request. If it is not present or it is invalid, a 422 HTTP error is returned.
- The products function handles the catalogue listing request from the retailer application. Any filter that is sent through the request is applied to the query and the result is return in json format.
- The place_order function creates an order sent by a retailer application. First of all, it retrieves the customer record based on the API key provided by a request

parameter, called key. Then, it checks if the request contains a non-empty array named products and if it does not, it returns a 422 HTTP error. Else, it creates a new order record in the database with the customer's id. Finally, for each product in the array, a new order_item record is stored in the database for the forementioned order.

- The `get_order_status` function returns the status of a specified order and customer. The function retrieves the customer record based on the API key provided by a request parameter, called key. If the request parameter `order_id` is not present, a 422 HTTP error is returned. Else, the order record is retrieved from the database based on the `order_id` and the customer id and its status is returned.
- The `get_products` function returns the data of products based on an array of product ids sent with the request. If the array is not present, a 422 HTTP error is returned. Else, all products with the specified ids that are enabled are retrieved from the database and returned in json format.
- The `parseFilters` function is executed inside the `products` function and it is responsible for applying any requested filter on the catalogue listing query. For each possible filter, it is checked if it is present in the request and if so, the appropriate SQL clause is appended to the query which will be executed later in the index function. It also appends a where clause to retrieve only products that are set as active.

Model Classes

Model classes in Laravel represent a database table and their fields represent the table columns. Laravel uses an Object-Relational Mapper (ORM), a plugin that makes database interaction very easy and less time consuming. Almost all queries are generated automatically and can be called from a model instance by auto-generated functions. Relations between models can also be defined in a model class and Laravel will automatically include them in the queries. Finally, computed attributes can be defined which are custom attributes that do not exist in the database and are calculated per model instance on the runtime from PHP.

Customers Model

```
1 <?php
2
3 namespace App\Models;
4
5 use Illuminate\Database\Eloquent\Model;
6
7 class Customers extends Model
8 {
9     protected $primaryKey = 'id';
10    public $timestamps = false;
11    protected $appends = ['total_spend'];
12    protected $guarded = [];
13
14    public function orders()
15    {
16        return $this->hasMany(Orders::class, 'customer_id');
17    }
18
19    public function getTotalSpendAttribute()
20    {
21        $orders = $this->orders()
22                ->where('status',2)
23                ->withSum('products as total_cost', \DB::raw('price * order_item.quantity'))
24                ->get();
25
26
27        $sum = 0;
28        foreach ($orders as $order) {
29            $sum += $order->total_cost;
30        }
31        return $sum;
32    }
33 }
34
```

This model represents the customers table in the database. A many-to-many relationship with the orders table is defined. Additionally, the total_spend computed attribute is defined that contains the total amount of money that the customer spent on orders.

Products Model

```
1 <?php
2
3 namespace App\Models;
4
5 use Illuminate\Database\Eloquent\Model;
6
7 class Products extends Model
8 {
9     protected $primaryKey = 'id';
10    protected $guarded = [];
11
12    public $timestamps = false;
13
14    public function orders()
15    {
16        return $this->belongsToMany(Orders::class, 'order_item', 'product_id', 'order_id')
17                ->withPivot('quantity');
18    }
19 }
20
```

This model represents the products table in the database. A many-to-many relationship with the orders table is defined. A pivot is used as an intermediate table that breaks the initial many-to-many relationship to one-to-many relationships.

Orders Model

```
1 <?php
2
3 namespace App\Models;
4
5 use Illuminate\Database\Eloquent\Model;
6
7 class Orders extends Model
8 {
9     protected $primaryKey = 'id';
10    protected $guarded = [];
11    protected $appends = ['can_be_completed'];
12
13    public $timestamps = false;
14
15    public function products()
16    {
17        return $this->belongsToMany(Products::class, 'order_item', 'order_id', 'product_id')
18            ->withPivot('quantity', 'product_title', 'product_price')
19            ->as('order_item');
20    }
21
22    public function customer()
23    {
24        return $this->belongsTo(Customers::class, 'customer_id');
25    }
26
27    public function order_status()
28    {
29        return $this->belongsTo(OrderStatus::class, 'status');
30    }
31
32    public function getCanBeCompletedAttribute()
33    {
34        return $this->products()->min(DB::raw('products.stock >= order_item.quantity')) == 1;
35    }
36 }
37
```

This model represents the orders table in the database. A many-to-many relationship with the products table is defined. A pivot is used as an intermediate table that breaks the initial many-to-many relationship to one-to-many relationships. Two more relationships are defined, one belongs-to with the customers model and another one with the order_status model. Finally, the can_be_completed computed attribute is defined that contains a boolean value of whether the specific order can be changed to delivered status or not. It calculates that by checking if the stock of the order's products can satisfy the respective order quantities.

OrderItem Model

```
1 <?php
2
3 namespace App\Models;
4
5 use Illuminate\Database\Eloquent\Model;
6
7 class OrderItem extends Model
8 {
9     protected $table = 'order_item';
10    protected $primaryKey = ['product_id', 'order_id'];
11    protected $appends = ['total_cost'];
12    public $incrementing = false;
13    public $timestamps = false;
14
15
16    protected $guarded = [];
17
18    public function product()
19    {
20        return $this->belongsTo(Products::class, 'product_id');
21    }
22
23    public function order()
24    {
25        return $this->belongsTo(Orders::class, 'order_id');
26    }
27
28    public function getTotalCostAttribute()
29    {
30        return $this->product()->first()->price * $this->quantity;
31    }
32 }
```

This model is the pivot or the intermediate table that connects the orders and products table and breaks their many-to-many relationship. Two belongs-to relationships are defined one for the orders table and one for the products table. Moreover, the total_cost computed attribute is defined which contains the total cost of the specific order item and it is calculated by multiplying the order quantity and the product's price.

Appendix B – Retailer System Source Code

Activities

Main Activity

```
1 package com.georgemichailidis.ventail;
2
3 import ...
12
13 public class MainActivity extends AppCompatActivity {
14
15     @Override
16     protected void onCreate(Bundle savedInstanceState) {
17         super.onCreate(savedInstanceState);
18         setContentView(R.layout.activity_main);
19         setupGUI();
20         DatabaseHandler.initialize(context: this);
21     }
22
23     private void setupGUI(){
24         final Button btnCatalog = findViewById(R.id.btnCatalog);
25         final Button btnOrders = findViewById(R.id.btnOrders);
26         final Button btnVendors = findViewById(R.id.btnVendors);
27
28         View.OnClickListener listener = (v) -> {
29             if(v.equals(btnCatalog)){
30                 openActivity(CatalogActivity.class);
31             }
32             else if(v.equals(btnOrders)){
33                 openActivity(OrdersActivity.class);
34             }
35             else if(v.equals(btnVendors)){
36                 openActivity(VendorsActivity.class);
37             }
38         };
39
40         btnCatalog.setOnClickListener(listener);
41         btnOrders.setOnClickListener(listener);
42         btnVendors.setOnClickListener(listener);
43     }
44
45     private void openActivity(Class activityClass){
46         Intent i = new Intent(packageContext: MainActivity.this, activityClass);
47         startActivity(i);
48     }
49 }
50
51
52
53
```

This activity is the starting point of the retailer application. It creates a screen that contains a menu with three buttons, one for the product catalogue screen, one for the orders list screen and one for the vendor management screen. Also, it initializes the main, global database handler helper class.

Catalog Activity

```
1 package com.georgemichailidis.ventail;
2
3 import ...
26
27 public class CatalogActivity extends AppCompatActivity {
28
29     private int fetchLimit;
30     private ProductFetchTask.AsyncTaskListener fetchCompletedListener;
31
32     private View loader;
33
34     private ProductAdapter adapter;
35     private RecyclerView rv_products;
36     private EndlessRecyclerViewScrollListener scrollListener;
37     private boolean isScrollFetch = false;
38
39     public CatalogActivity(){
40         VendorCoordinator.initialize(Vendor.all(Vendor.class));
41         fetchLimit = 10;
42     }
43
44     @Override
45     public void onBackPressed() {
46         Intent upIntent = NavUtils.getParentActivityIntent( sourceActivity: this);
47         startActivity(upIntent);
48         finish();
49     }
50
51     @Override
52     protected void onCreate(Bundle savedInstanceState) {
53         super.onCreate(savedInstanceState);
54         setContentView(R.layout.activity_catalog);
55
56         FiltersHelper.clear();
57
58         loader = findViewById(R.id.Loader);
59
60         rv_products = (RecyclerView) findViewById(R.id.rv_products);
61
62         LinearLayoutManager layoutManager = new LinearLayoutManager( context: this);
63         rv_products.setLayoutManager(layoutManager);
64
65         adapter = new ProductAdapter();
66         rv_products.setAdapter(adapter);
67
68         fetchCompletedListener = new ProductFetchTask.AsyncTaskListener() {
69             @Override
70             public void onTaskCompleted(ArrayList<Product> products) {
71                 adapter.addAll(products);
72                 adapter.notifyDataSetChanged();
73                 loader.setVisibility(View.GONE);
74
75                 if( isScrollFetch ) {
76                     rv_products.addOnScrollListener(scrollListener);
77                     isScrollFetch = false;
78                 }
79             }
80         }
```

```

81         @Override
82         public void onTaskFailed(Exception e) {
83             loader.setVisibility(View.GONE);
84             Toast.makeText( context: CatalogActivity.this, text: "Vendor server not responding.", Toast.LENGTH_LONG).show();
85         }
86     };
87
88     scrolllistener = new EndlessRecyclerViewScrollListener(linearLayoutManager){
89         @Override
90         public void onLoadMore(int page, int totalItemsCount, RecyclerView view) {
91             fetchProductsOnScroll();
92         }
93     };
94     rv_products.addOnScrollListener(scrolllistener);
95     loadProducts();
96 }
97
98 @Override
99 public boolean onCreateOptionsMenu(Menu menu) {
100     menu.clear();
101     getMenuInflater().inflate(R.menu.catalog, menu);
102
103     MenuItem item = menu.findItem(R.id.menu_item_cart);
104     item.setActionView(R.layout.cart_badge);
105     FrameLayout notifCount = (FrameLayout) item.getActionView();
106     notifCount.setOnClickListener((v) -> { openCart(); });
107
108     TextView tv = (TextView) notifCount.findViewById(R.id.cart_badge);
109     CartHelper.initialize(tv);
110
111     return true;
112 }
113
114 @Override
115 public boolean onOptionsItemSelected(MenuItem item) {
116     switch (item.getItemId()) {
117         case R.id.menu_item_filters:
118             FiltersDialog dialog = new FiltersDialog( context: this);
119             dialog.setOnSearchListener(() -> {
120                 adapter.removeAll();
121                 scrolllistener.resetState();
122                 CatalogActivity.this.loadProducts();
123             });
124             return true;
125         default:
126             return super.onOptionsItemSelected(item);
127     }
128 }
129
130 private void loadProducts(){
131     if(!VendorCoordinator.hasNext())
132         return;
133
134     loader.setVisibility(View.VISIBLE); //start loading animation
135
136     //start async task to fetch products
137     ProductFetchTask task = new ProductFetchTask(fetchLimit);
138     task.setOnTaskCompleteListener(fetchCompletedListener);
139     task.execute();
140 }
141
142 private void fetchProductsOnScroll(){
143     isScrollFetch = true;
144     rv_products.removeOnScrollListener(scrolllistener);
145     loadProducts();
146 }
147
148 private void openCart(){
149     Intent intent = new Intent( packageContext: CatalogActivity.this, CartActivity.class);
150     startActivity(intent);
151 }
152
153 }

```

This activity presents the catalogue of products from all vendors in an infinite-scrolling list. Products are downloaded with the help of an ProductFetchTask object.

Cart Activity

```
1 package com.georgemichailidis.ventail;
2
3 import ...
30
31 public class CartActivity extends AppCompatActivity {
32
33     private CartItemAdapter adapter;
34
35     @Override
36     public void onBackPressed() {
37         Intent upIntent = NavUtils.getParentActivityIntent( sourceActivity: this);
38         startActivity(upIntent);
39         finish();
40     }
41
42     @Override
43     protected void onCreate(Bundle savedInstanceState) {
44         super.onCreate(savedInstanceState);
45         setContentView(R.layout.activity_cart);
46
47         final RecyclerView rv_cart_items = findViewById(R.id.rv_cart_items);
48         final Button cart_btn_order = findViewById(R.id.cart_btn_order);
49         final TextView tv_empty_cart = findViewById(R.id.tv_empty_cart);
50         final FrameLayout place_order_loader_layout = findViewById(R.id.place_order_loader_layout);
51
52         if(CartHelper.hasItems()){
53             tv_empty_cart.setVisibility(View.GONE);
54             cart_btn_order.setVisibility(View.VISIBLE);
55             rv_cart_items.setVisibility(View.VISIBLE);
56
57             LinearLayoutManager layoutManager = new LinearLayoutManager( context: this);
58             rv_cart_items.setLayoutManager(layoutManager);
59
60             adapter = new CartItemAdapter( CartHelper.getItems() );
61             adapter.setCartItemListener(() -> {
62                 tv_empty_cart.setVisibility(View.VISIBLE);
63                 cart_btn_order.setVisibility(View.GONE);
64                 rv_cart_items.setVisibility(View.GONE);
65             });
66             rv_cart_items.setAdapter(adapter);
67
68             cart_btn_order.setOnClickListener((v) -> {
69                 if(!CartHelper.hasItems()) {
70                     Toast.makeText(
71                         context: CartActivity.this,
72                         text: "Empty Cart",
73                         Toast.LENGTH_LONG
74                     ).show();
75                     return;
76                 }
77
78                 HashMap<Long, HashMap<Long, OrderItem>> items = CartHelper.getMappedItems();
79                 ArrayList<PartialOrder> pOrders = new ArrayList<>();
80
81                 for (Map.Entry<Long, HashMap<Long, OrderItem>> entries: items.entrySet()) {
82                     ArrayList<OrderItem> orderItems = new ArrayList<>(entries.getValue().values());
83                     if(orderItems.size() > 0)
84                         pOrders.add( new PartialOrder(entries.getKey(), orderItems) );
85                 }
86
87                 if( pOrders.size() > 0 ){
88                     place_order_loader_layout.setVisibility(View.VISIBLE);
89                     PlaceOrderTask task = new PlaceOrderTask(pOrders);
90
91
92
93
94
95
```

```

96 task.setOnTaskCompleteListener(new PlaceOrderTask.AsyncTaskListener() {
97     @Override
98     public void onTaskCompleted(ArrayList<PartialOrder> orders) {
99         place_order_loader_layout.setVisibility(View.GONE);
100         if(orders.size() <= 0)
101             Toast.makeText(
102                 context: CartActivity.this,
103                 text: "Failed to communicate with all vendors. Order could not be placed.",
104                 Toast.LENGTH_LONG
105             ).show();
106         else
107             saveOrder(orders);
108     }
109
110     @Override
111     public void onTaskFailed(Exception e) {
112         Toast.makeText(
113             context: CartActivity.this,
114             text: "An error occurred. Please try again later.",
115             Toast.LENGTH_LONG
116         ).show();
117         place_order_loader_layout.setVisibility(View.GONE);
118     }
119 });
120
121 task.execute();
122 }
123 });
124 }
125 }
126 }
127
128 private void saveOrder(ArrayList<PartialOrder> partialOrders){
129     DatabaseHandler.instance.beginTransaction();
130     try {
131         Order order = new Order(partialOrders);
132         if( order.create() ){
133             DatabaseHandler.instance.commit();
134         }
135         else{
136             DatabaseHandler.instance.rollback();
137         }
138
139         CartHelper.clear();
140         finish();
141     }catch (Exception e){
142         e.printStackTrace();
143         DatabaseHandler.instance.rollback();
144     }
145 }
146 }

```

Cart activity can be launched either by Catalog activity or during a reorder attempt from OrderDetail activity. The activity displays all items that are currently in the cart and their quantities can be modified further or they can be removed from cart completely.

Vendors Activity

```
1 package com.georgemichailidis.ventail;
2
3 import ...
4
5
6
7
8
9
10
11
12
13
14
15
16 public class VendorsActivity extends AppCompatActivity {
17
18     private RecyclerView rvVendorsList;
19     private VendorAdapter adapter;
20
21     @Override
22     protected void onCreate(Bundle savedInstanceState) {
23         super.onCreate(savedInstanceState);
24         setContentView(R.layout.activity_vendors);
25
26         TextView msgEmpty = findViewById(R.id.msgEmpty);
27         FloatingActionButton btnAddVendor = findViewById(R.id.btnAddVendor);
28
29         rvVendorsList = findViewById(R.id.rvVendorsList);
30         LinearLayoutManager layoutManager = new LinearLayoutManager( context: this);
31         rvVendorsList.setLayoutManager(layoutManager);
32
33         adapter = new VendorAdapter(Vendor.all(Vendor.class));
34
35         rvVendorsList.setAdapter(adapter);
36
37         if(adapter.getItemCount() > 0){
38             msgEmpty.setVisibility(View.GONE);
39             rvVendorsList.setVisibility(View.VISIBLE);
40         }
41
42         btnAddVendor.setOnClickListener((v) -> {
43
44             VendorDialog dialog = new VendorDialog( context: VendorsActivity.this);
45             dialog.setOnCreateListener((newVendor) -> {
46                 adapter.add(newVendor);
47             });
48             dialog.show();
49         });
50
51
52
53
54     };
55 }
56
57 }
```

This activity is responsible of presenting the vendor management feature of the retailer system. Vendors are loaded from the database to a RecyclerView.

Orders Activity

```
1 package com.georgemichailidis.ventail;
2
3 import ...
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18 public class OrdersActivity extends AppCompatActivity {
19
20     @Override
21     protected void onCreate(Bundle savedInstanceState) {
22         super.onCreate(savedInstanceState);
23         setContentView(R.layout.activity_orders);
24
25         ArrayList<Order> orders = Order.all(Order.class);
26
27         TextView tv_no_orders = findViewById(R.id.tv_no_orders);
28         RecyclerView rv_orders = findViewById(R.id.rv_orders);
29
30         if(orders.size() > 0){
31             tv_no_orders.setVisibility(View.GONE);
32             rv_orders.setVisibility(View.VISIBLE);
33
34             LinearLayoutManager linearLayoutManager = new LinearLayoutManager(context, this);
35             rv_orders.setLayoutManager(linearLayoutManager);
36
37             OrderAdapter adapter = new OrderAdapter(Order.all(Order.class));
38             rv_orders.setAdapter(adapter);
39         }
40     }
41 }
```

This activity lists all orders placed by the user through the retailer system. All orders are listing in a RecyclerView.

OrderDetail Activity

```
1 package com.georgemichailidis.ventail;
2
3 import ...
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23 public class OrderDetailActivity extends AppCompatActivity {
24
25     public static final String PARTIAL_ORDERS_TAG = "partial_orders";
26
27     private ArrayList<PartialOrder> partialOrders;
28
29     @Override
30     protected void onCreate(Bundle savedInstanceState) {
31         super.onCreate(savedInstanceState);
32         setContentView(R.layout.activity_order_detail);
33
34         Intent intent = getIntent();
35         partialOrders = null;
36         if(intent.hasExtra(PARTIAL_ORDERS_TAG)){
37             partialOrders = (ArrayList<PartialOrder>) intent.getSerializableExtra(PARTIAL_ORDERS_TAG);
38         }
39
40         if(partialOrders == null || partialOrders.size() <= 0)
41             finish();
42     }
43 }
```

```

43     final FrameLayout loader = findViewById(R.id.partial_orders_loader);
44
45     OrderStatusTask task = new OrderStatusTask(partialOrders);
46     task.setOnTaskCompleteListener((orders) -> {
47         RecyclerView rv_partial_orders = findViewById(R.id.rv_partial_orders);
48         LinearLayoutManager layoutManager = new LinearLayoutManager(
49             context: OrderDetailActivity.this
50         );
51         rv_partial_orders.setLayoutManager(layoutManager);
52
53         PartialOrderAdapter adapter = new PartialOrderAdapter(partialOrders);
54         rv_partial_orders.setAdapter(adapter);
55
56         loader.setVisibility(View.GONE);
57     });
58
59     task.execute();
60
61     Button order_details_btn_reorder = findViewById(R.id.order_details_btn_reorder);
62     order_details_btn_reorder.setOnClickListener((v) -> {
63         loader.setVisibility(View.VISIBLE);
64         ProductsReorderTask task = new ProductsReorderTask(partialOrders);
65         task.setOnTaskCompleteListener((orderItems) -> {
66             if(orderItems.size() > 0){
67                 CartHelper.clear();
68                 CartHelper.addAll(orderItems);
69                 Intent intent = new Intent(
70                     packageContext: OrderDetailActivity.this,
71                     CartActivity.class
72                 );
73                 startActivity(intent);
74                 finishAffinity();
75             }
76             else {
77                 loader.setVisibility(View.GONE);
78             }
79         });
80
81         task.execute();
82     });
83
84 }
85
86 }
87
88 }
89
90 }
91
92 }
93
94 }

```

OrderDetail activity shows all information about a specific order. An order in the retailer system can consist multiple partial or sub orders, each one of them associated with a specific vendor. An ArrayList of PartialOrder objects from the main order is passed through the intent. Furthermore, when this activity is launched, an async task is executed to download the status of each partial order from its respective vendor system.

AsyncTask Classes

ProductFetchTask

```
1 package com.georgemichailidis.ventail.asyncTasks;
2
3 import ...
18
19 public class ProductFetchTask extends AsyncTask<Void, Void, ArrayList<Product>> {
20
21     private int limit;
22     private int connectionTimeout;
23     private AsyncTaskListener listener;
24     private Exception error;
25
26     public ProductFetchTask(int limit){
27         this.connectionTimeout = 3000; //milliseconds
28         this.limit = limit;
29     }
30
31     @Override
32     protected ArrayList<Product> doInBackground(Void... voids) {
33
34         ArrayList<Product> products = new ArrayList<>();
35         int currentLimit = limit;
36
37         Vendor vendor = VendorCoordinator.next();
38
39         //Loop until we run out of vendors OR fetch specified number of products
40         while(vendor != null && currentLimit > 0){
41
42             Uri.Builder builder = vendor.getUrlBuilder();
43
44             //apply necessary query params to http request
45             builder.appendQueryParameter("action", "catalog")
46                 .appendQueryParameter("limit", String.valueOf( currentLimit ))
47                 .appendQueryParameter(
48                     "offset",
49                     String.valueOf( VendorCoordinator.getCurrentVendorOffset())
50                 );
51
52             //apply any existing search filters
53             builder = FiltersHelper.applyFilters(builder);
54
55             HttpURLConnection urlConnection = null;
56
57             try {
58                 String urlString = builder.build().toString();
59                 Log.d( tag: "debug", urlString);
60                 URL url = new URL( urlString );
61
62                 urlConnection = (HttpURLConnection) url.openConnection();
63                 urlConnection.setConnectTimeout(connectionTimeout);
64                 urlConnection.setReadTimeout(5000);
65
66                 InputStream in = new BufferedInputStream(urlConnection.getInputStream());
67                 products.addAll( Product.jsonParseArray(in, vendor.getId()) );
68                 in.close();
69             } catch (SocketTimeoutException e){
70                 error = e;
```

```

71     } catch (Exception e){
72         Log.e(ProductFetchTask.class.getSimpleName(), msg: "Fetch Error", e);
73     } finally {
74         if(urlConnection != null)
75             urlConnection.disconnect();
76     }
77
78     //if current vendor has no more products
79     if( limit - products.size() > 0 ){
80         VendorCoordinator.resetCurrentVendorOffset(); //reset offset
81         vendor = VendorCoordinator.next();//move to next vendor
82     }
83     else{//if current vendor has more products
84         //keep offset/pagination, so next time request with this offset
85         VendorCoordinator.addCurrentVendorOffset( currentLimit );
86     }
87
88     currentLimit = limit - products.size();//calculate next loop limit
89 }
90
91 return products;
92 }
93
94 @Override
95 protected void onPostExecute(ArrayList<Product> products) {
96     super.onPostExecute(products);
97
98     if(this.listener != null) {
99         if(products.size() == 0 && error != null)
100             this.listener.onTaskFailed(error);
101         else
102             this.listener.onTaskCompleted(products);
103     }
104 }
105
106 public void setOnTaskCompletedListener(AsyncTaskListener l) { this.listener = l; }
107
108 public interface AsyncTaskListener {
109     public void onTaskCompleted(ArrayList<Product> products);
110     public void onTaskFailed(Exception e);
111 }
112
113 }
114 }

```

This async task is designed to download product data through HTTP requests from all vendors that are registered in the application. The task is executed each time the RecyclerView in the Catalog activity reaches a threshold before its ending. Additionally, the task uses the VendorCoordinator class to coordinate the process of downloading product data from multiple different vendor systems. After the task finishes its execution, if the AsyncTaskListener has been set, one of its two functions will be executed based on whether the execution was successful or not.

PlaceOrderTask

```
1 package com.georgemichailidis.ventail.asyncTasks;
2
3 import ...
28
29 public class PlaceOrderTask extends AsyncTask<Void, Void, Void> {
30
31     private int connectionTimeout;
32     private AsyncTaskListener listener;
33     private Exception error;
34     private ArrayList<PartialOrder> partialOrders;
35
36     public PlaceOrderTask(ArrayList<PartialOrder> pOrders){
37         this.connectionTimeout = 3000; //milliseconds
38         partialOrders = pOrders;
39     }
40
41     @Override
42     protected Void doInBackground(Void... voids) {
43
44         for (PartialOrder pOrder : new ArrayList<PartialOrder>(partialOrders)) {
45             boolean success = false;
46             Vendor vendor = pOrder.getVendor();
47             Uri.Builder builder = vendor.getUrlBuilder();
48
49             //apply necessary query params to http request
50             builder.appendQueryParameter("action", "place_order");
51
52             HttpURLConnection urlConnection = null;
53             try {
54                 String urlString = builder.build().toString();
55                 URL url = new URL( urlString );
56                 urlConnection = (HttpURLConnection) url.openConnection();
57                 urlConnection.setRequestMethod("POST");
58                 urlConnection.setRequestProperty("Content-Type", "application/json; charset=UTF-8");
59                 urlConnection.setDoOutput(true);
60
61                 byte[] input = pOrder.toJson().getBytes( charsetName: "UTF-8");
62                 urlConnection.setFixedLengthStreamingMode(input.length);
63                 OutputStream os = urlConnection.getOutputStream();
64                 os.write(input);
65                 os.flush();
66                 os.close();
67
68                 int responseCode = urlConnection.getResponseCode();
69                 Log.d( tag: "debug", msg: "response code: "+responseCode);
70                 BufferedReader in = new BufferedReader(
71                     new InputStreamReader(urlConnection.getInputStream(), charsetName: "utf-8")
72                 );
73                 StringBuilder response = new StringBuilder();
74                 String responseLine = null;
75                 while ((responseLine = in.readLine()) != null) {
76                     response.append(responseLine.trim());
77                 }
78             }
```

```

80     try {
81         Long orderId = Long.parseLong(response.toString());
82         pOrder.setId(orderId);
83         success = true;
84     } catch (NumberFormatException e){
85         error = e;
86     }
87     } catch (SocketTimeoutException e){
88         error = e;
89     } catch (Exception e){
90         e.printStackTrace();
91         error = e;
92     } finally {
93         if(urlConnection != null)
94             urlConnection.disconnect();
95     }
96
97     if(!success){
98         partialOrders.remove(pOrder);
99     }
100 }
101
102 return null;
103 }
104
105 @Override
106 protected void onPostExecute(Void aVoid) {
107     super.onPostExecute(aVoid);
108
109     if(this.listener != null) {
110         if(error != null)
111             this.listener.onTaskFailed(error);
112         else
113             this.listener.onTaskCompleted(partialOrders);
114     }
115 }
116
117 public void setOnTaskCompletedListener(AsyncTaskListener l) { this.listener = l; }
118
119
120
121 public interface AsyncTaskListener {
122     public void onTaskCompleted(ArrayList<PartialOrder> orders);
123     public void onTaskFailed(Exception e);
124 }
125 }

```

PlaceOrderTask is responsible to send through HTTP request each partial order to the appropriate vendor. The vendor system has to return the order id that its database generated and it will be used as the id of the PartialOrder record that will be stored in the retailer system. After the task finishes its execution, if the AsyncTaskListener has been set, one of its two functions will be executed based on whether the execution was successful or not.

Other Classes

EndlessRecyclerViewScrollListener

```
1 package com.georgemichailidis.ventail;
2
3 import ...
4
5
6
7
8
9
10 public abstract class EndlessRecyclerViewScrollListener extends RecyclerView.OnScrollListener {
11     // The minimum amount of items to have below your current scroll position
12     // before loading more.
13     private int visibleThreshold = 2;
14     // The current offset index of data you have loaded
15     private int currentPage = 0;
16     // The total number of items in the dataset after the last load
17     private int previousTotalItemCount = 0;
18     // True if we are still waiting for the last set of data to load.
19     private boolean loading = false;
20     // Sets the starting page index
21     private int startingPageIndex = 0;
22
23     RecyclerView.LayoutManager mLayoutManager;
24
25     public EndlessRecyclerViewScrollListener(LinearLayoutManager layoutManager) {
26         this.mLayoutManager = layoutManager;
27     }
28
29     @ public EndlessRecyclerViewScrollListener(GridLayoutManager layoutManager) {
30         this.mLayoutManager = layoutManager;
31         visibleThreshold = visibleThreshold * layoutManager.getSpanCount();
32     }
33
34     @ public EndlessRecyclerViewScrollListener(StaggeredGridLayoutManager layoutManager) {
35         this.mLayoutManager = layoutManager;
36         visibleThreshold = visibleThreshold * layoutManager.getSpanCount();
37     }
38
39     @ public int getLastVisibleItem(int[] lastVisibleItemPositions) {
40         int maxSize = 0;
41         for (int i = 0; i < lastVisibleItemPositions.length; i++) {
42             if (i == 0) {
43                 maxSize = lastVisibleItemPositions[i];
44             }
45             else if (lastVisibleItemPositions[i] > maxSize) {
46                 maxSize = lastVisibleItemPositions[i];
47             }
48         }
49         return maxSize;
50     }
51
52     // This happens many times a second during a scroll, so be wary of the code you place here.
53     // We are given a few useful parameters to help us work out if we need to load some more data,
54     // but first we check if we are waiting for the previous load to finish.
55     @Override
56     public void onScrolled(RecyclerView view, int dx, int dy) {
57         int lastVisibleItemPosition = 0;
58         int totalItemCount = mLayoutManager.getItemCount();
```

```

60     if (mLayoutManager instanceof StaggeredGridLayoutManager) {
61         int[] lastVisibleItemPositions = ((StaggeredGridLayoutManager) mLayoutManager)
62             .findLastVisibleItemPositions( into: null);
63         // get maximum element within the list
64         lastVisibleItemPosition = getLastVisibleItem(lastVisibleItemPositions);
65     } else if (mLayoutManager instanceof GridLayoutManager) {
66         lastVisibleItemPosition = ((GridLayoutManager) mLayoutManager)
67             .findLastVisibleItemPosition();
68     } else if (mLayoutManager instanceof LinearLayoutManager) {
69         lastVisibleItemPosition = ((LinearLayoutManager) mLayoutManager)
70             .findLastVisibleItemPosition();
71     }
72
73     // If the total item count is zero and the previous isn't, assume the
74     // list is invalidated and should be reset back to initial state
75     if (totalItemCount < previousTotalItemCount) {
76         this.currentPage = this.startingPageIndex;
77         this.previousTotalItemCount = totalItemCount;
78         if (totalItemCount == 0) {
79             this.loading = true;
80         }
81     }
82     // If it's still loading, we check to see if the dataset count has
83     // changed, if so we conclude it has finished loading and update the current page
84     // number and total item count.
85     if (loading && (totalItemCount > previousTotalItemCount)) {
86         loading = false;
87         previousTotalItemCount = totalItemCount;
88     }
89
90     // If it isn't currently loading, we check to see if we have breached
91     // the visibleThreshold and need to reload more data.
92     // If we do need to reload some more data, we execute onLoadMore to fetch the data.
93     // threshold should reflect how many total columns there are too
94     if (!loading && (lastVisibleItemPosition + visibleThreshold) > totalItemCount) {
95         currentPage++;
96         onLoadMore(currentPage, totalItemCount, view);
97         loading = true;
98     }
99 }
100
101 // Call this method whenever performing new searches
102 public void resetState() {
103     this.currentPage = this.startingPageIndex;
104     this.previousTotalItemCount = 0;
105     this.loading = true;
106 }
107
108 // Defines the process for actually loading more data based on page
109 public abstract void onLoadMore(int page, int totalItemsCount, RecyclerView view);
110 }

```

This class is the heart of the endless-scrolling list of the product catalog. It is a scroll listener which is set on the RecyclerView that contains the products in the Catalog activity. With every scroll, unless the end of the RecyclerView list is reached, it calculates how many items are visible on screen and if the number is close to a pre-defined threshold, its state changes to loading state. That means that the scroll is close to the end of the list and more products must be downloaded, if there are any more left to download.

VendorCoordinator

```
1 package com.georgemichailidis.ventail;
2 import ...
5
6 public class VendorCoordinator {
7
8     private static int currentVendor;
9     private static int currentVendorOffset;
10    private static Vendor vendorFilter;
11    private static ArrayList<Vendor> vendors;
12
13    public static void initialize(ArrayList<Vendor> v){
14        if(v != null)
15            vendors = v;
16        else
17            vendors = new ArrayList<Vendor>();
18
19        reset();
20    }
21
22    public static void reset(){
23        currentVendor = (vendors.size() > 0) ? -1 : -2;
24        currentVendorOffset = 0;
25        vendorFilter = null;
26    }
27
28    public static void resetWithFilter(Vendor v){
29        currentVendor = -2;
30        currentVendorOffset = 0;
31        vendorFilter = v;
32    }
33
34    public static boolean hasNext(){
35        return
36            //current vendor in range
37            (currentVendor+1 > -1 && currentVendor+1 <= vendors.size()-1) ||
38            //if last vendor has possible remaining items
39            ( currentVendor == vendors.size() - 1 && currentVendorOffset > 0 ) ||
40            (vendorFilter != null);
41    }
42
43    @ public static Vendor next(){
44        if(hasNext()){
45
46            if(vendorFilter != null)
47                return vendorFilter;
48
49            //if has remaining offset (possibly has more products), do not move to next yet.
50            if(currentVendorOffset <= 0){
51                currentVendor++;
52            }
53
54            return vendors.get(currentVendor);
55        }
56        else
57            return null;
58    }
59
60    public static void addCurrentVendorOffset(int n) { currentVendorOffset += n; }
63
64    public static void resetCurrentVendorOffset(){
65        vendorFilter = null;
66        currentVendorOffset = 0;
67    }
68
69    public static int getCurrentVendorOffset() { return currentVendorOffset; }
72 }
```

This class is consisted only with static methods and it is used to help the ProductFetchTask by coordinating the process of downloading product data from multiple different vendor systems. It contains a list of all registered vendors, the index of the current vendor, the offset of the current vendor and the selected vendor from the catalog filters. The fetching task retrieves from the coordinator the current vendor and its offset and downloads products from this vendor by the current offset, until no more products are returned. Then, the coordinator moves the index to the next vendor and this process continues, until the index reaches the end of the list. If the user has selected a specific vendor from the catalog filters, the coordinator will return only the selected vendor to the fetching task, thus downloading products only from this vendor.

Model Class

```
1 package com.georgemichailidis.ventail.models;
2
3 import ...
4
11
12 public abstract class Model {
13     public static final String COLUMN_ID = "id";
14
15     protected Long id;
16     protected ContentValues values;
17
18     public Model(){
19         id = null;
20         values = new ContentValues();
21     }
22
23     public Model(Long id){
24         this.id = id;
25         values = new ContentValues();
26     }
27
28     public Long getId(){return id;}
29
30     public void setId(Long id) { this.id = id; }
31
32
33
34     public boolean hasId(){
35         return (this.id != null);
36     }
37
38     public boolean create(){
39         ContentValues v = new ContentValues(values);
40         Long newId = -11;
41         if( this.hasId() ){
42             v.put(COLUMN_ID, getId());
43             newId = DatabaseHandler.instance.insert(this.getClass(), v);
44         }
45         else{
46             newId = DatabaseHandler.instance.insert(this.getClass(), v);
47             setId(newId);
48         }
49
50         return newId != -11;
51     }
52
53     public boolean update(){
54         if( this.hasId() ){
55             DatabaseHandler.instance.update(this.getClass(), getId(), values);
56             return true;
57         }
58         else
59             return false;
60     }
61 }
```

```

61
62 public boolean delete() {
63     DatabaseHandler.instance.delete( getClass(), getId() );
64     return true;
65 }
66
67 public static <T extends Model> ArrayList<T> all(Class<T> type){
68     return DatabaseHandler.instance.selectAll(type);
69 }
70
71 public static <T extends Model> T find(Class<T> type, Long id){
72     return DatabaseHandler.instance.selectById(type, id);
73 }
74 }

```

Model is an abstract class that is extended by the persistent classes of the retailer system. Typically, persistent classes are those that their objects have data needed to be permanently stored in a filesystem or a database. So in Ventail, those Model children classes represent all different database tables, such as vendors, orders, partial orders, and all common fields and methods are gathered in one place, thus the existence of Model abstract class. One such common field is the id field which represent a database column that contains a unique identifier number for each record. Finally, some common methods are the create method which inserts a new record in the database based on the object's data, the update method which uses the id field to update the specified record based on the object's current data and the delete method which uses the id field to delete the specific record from the database.