Bilkent University

Department of Computer Engineering

Senior Design Project

Farket

Final Report

Deniz Alkışlar | H. Buğra Aydın | M. Erim Erdal | M. Enes Keleş | Hakan Türkmenoğlu

Supervisor: Eray Tüzün

Jury Members: Mustafa Özdal and Özcan Öztürk

Innovation Expert: Barış Misman

Final Report

May 9, 2019

This report is submitted to the Department of Computer Engineering of Bilkent University in partial fulfillment of the requirements of the Senior Design Project course CS491/2.

Introduction

Finding the best price is always a daunting task. Comparing prices between stores is both time and energy inefficient especially on essential items such as food, cleaning products, and cosmetics.

This problem highly affects lives of people with low income including students, homemakers, working class families, especially in countries that have high inflation which results in market price fluctuations nearly on a daily basis. These people have to constantly track prices in order to cut off expenses. To this end they buy from gross markets and online stores, and collect discount coupons. Both of these solutions have downsides, collecting coupons is time and energy consuming and going to a gross market is not the best option in most times which we discuss in the following table.

We conducted a case study on grocery prices in order to see the price difference using a prepared list of most crucial items corresponding to basic needs such as provisions and cleaning products. When purchased from the overall cheapest grocery store, the total cost is 109 Turkish Liras. However when the cheapest option of every item is bought the total is 99 Turkish Liras which is about 10% cheaper than the former *(Table I)*

	Carrefour	**A-101**	**BIM**	**Akakçe**	**Cheapest option selected**
500g Spagetti	2.35 TL	1.15 TL	1.15 TL	3 TL	1.15
1 kg potatoes	2.79 TL	*	2.73 TL	*	2.73
1 kg tomatoes	6.95 TL	*	6.95 TL	*	6.95

1 kg onions	2.79 TL	*	2.98 TL	*	2.79
15 eggs	11.99 TL	8 TL	8 TL	10 TL	8
1 lt milk	2.75 TL	3.25 TL	3.25 TL	*	2.75
Price / diaper**	0.67 TL	0.36 TL	0.365 TL	0.342 TL	0.342
32x toilet papers**	31.90 TL	31 TL	31 TL	24 TL	24
Detergent / KG**	5.28 TL	4.18 TL	3.75 TL	*	3.75
Disher Capsule**	1.01 TL	0.79 TL	0.35 TL	0.83 TL	0.35
-TOTAL-	**169 TL**	**131 TL**	**107 TL**	**128 TL**	**99 TL**

Table 1. Price comparison between different markets [on 30th of October 2018]
*note that missing prices are plugged with the price of the cheapest option
**prices are normalized for quantities

Farkett therefore aims to cut down grocery costs by creating a distributed shopping list which we are able to buy cheapest products from 4 different store options instead of buying all the items from the single cheapest store. As it can be seen from the case study, this results on an average 10-15% profit.

Architecture

We used a server client architecture in Farkett. Farkett's backend makes use of repository pattern which provides an abstraction of data, so that the application can work with a an interface approximating that of a collection. Adding, removing, updating, and selecting items from the database is done through a series of straightforward methods, without any concerns like connections, commands, cursors, or readers in the rest of the code. The backend serves the data to clients via a REST API in JSON format, and the frontend renders it on device screen.

Front-end

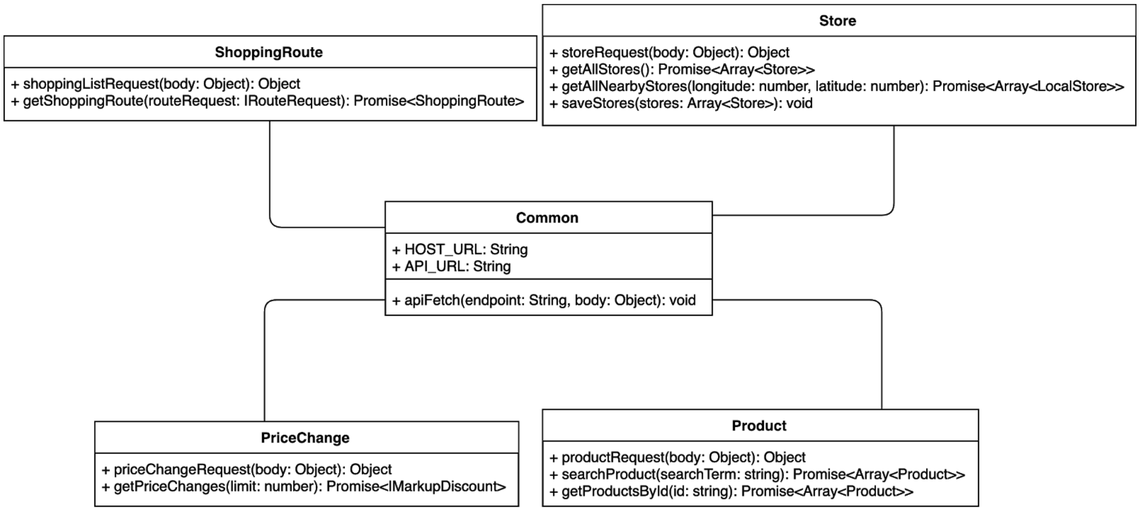
Farkett uses React Native to develop a cross-platform app for both Android and iOS operating systems. React Native puts emphasis into composition over inheritance, for this reason we tried to reduce inheritance in our code base which reduces coupling, therefore increasing maintainability and scalability. In React, each view is composed of different views, and each component has child and parent relationship, forming a UI tree. These views are updated according to the reactive programming pattern, which is very similar to the observer pattern. The difference between the two is that reactive pattern also facilitates a way of automatically updating the view using the data flow whenever the UI model changes. The update process is implemented as efficient as possible, only updating views that can be affected by the data change. Thus, reactive programming pattern can express reactivity in members of objects, while observer pattern updates the whole object structure.

Serialization and Server-Client architecture is a big part of our application. This is achieved via `api` and `models` packages. `models` package describes the model of the objects we receive and send, while `api` package is responsible from sending and receiving the objects themselves using the uniform REST API through HTTP requests to the backend. JSON is used for serialization format.

Our implementation language of choice is Typescript which is a superset of Javascript, providing static type definitions for better developer experience, reducing bugs related to types.
In some of these diagrams we had to use Function<A, B, C…> pseudo-type which represents the closure/lambda that is widely used in functional programming and Javascript. The first type inside it denotes the return type, while the rest denotes the types of its parameters. Similarly, Promise<T> is a native type that is available in Javascript, mainly used for asynchronous functions. The return type of the Promise is denoted inside the brackets.

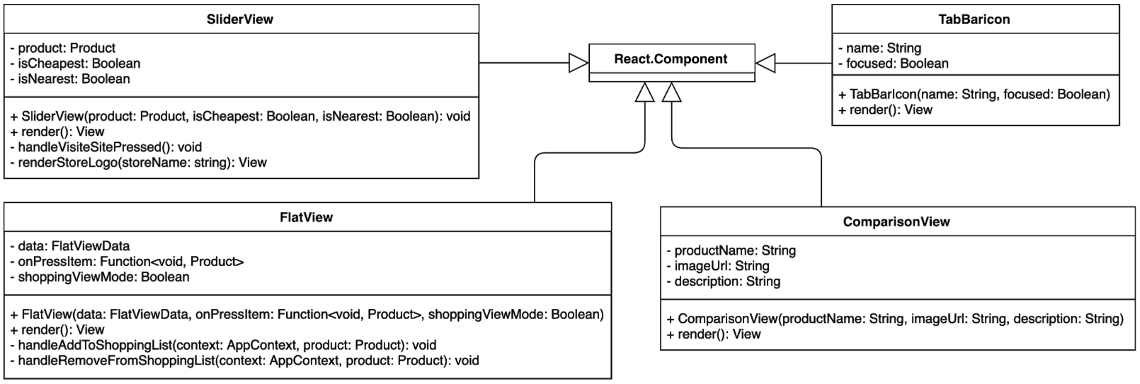
API Package

This package is responsible from handling HTTP requests and implementing the endpoints of backend. Returned data is transferred to Models package for easier management.


Store
+ storeRequest(body: Object): Object
+ getAllStores(): Promise<Array<Store>>
+ getAllNearbyStores(longitude: number, latitude: number): Promise<Array<LocalStore>>
+ saveStores(stores: Array<Store>): void
ShoppingRoute
+ shoppingListRequest(body: Object): Object
+ getShoppingRoute(routeRequest: IRouteRequest): Promise<ShoppingRoute>
Common
+ HOST_URL: String
+ API_URL: String
+ apiFetch(endpoint: String, body: Object): void
PriceChange
+ priceChangeRequest(body: Object): Object
+ getPriceChanges(limit: number): Promise<IMarkupDiscount>
Product
+ productRequest(body: Object): Object
+ searchProduct(searchTerm: string): Promise<Array<Product>>
+ getProductsById(id: string): Promise<Array<Product>>

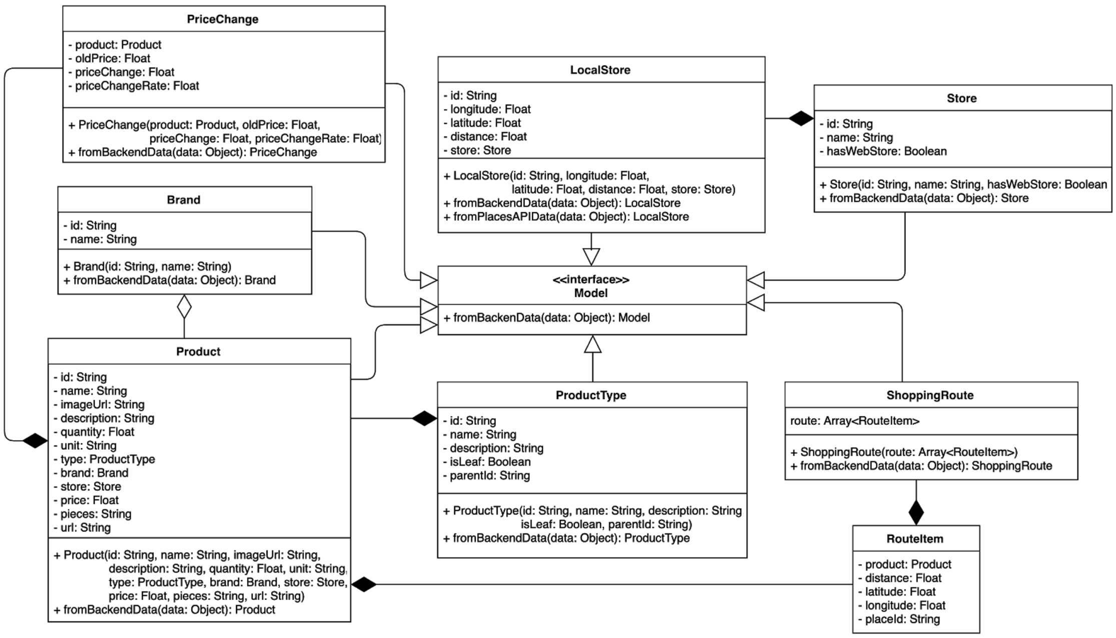
Components Package

This package includes small helper components (views) that are used in our application. SliderView implements the slider inside product comparison screen where user can swipe prices across different stores. TabBarIcon is used in the bottom tab bar. ComparisonView is used in product comparison screen to display product image, description and its name. FlatView implements common list view that is shared between shopping list and home screens. shoppingViewMode property is used to decide whether the view should display store icons or not.


SliderView
- product: Product
- isCheapest: Boolean
- isNearest: Boolean
+ SliderView(product: Product, isCheapest: Boolean, isNearest: Boolean): void
+ render(): View
- handleVisiteSitePressed(): void
- renderStoreLogo(storeName: string): View
React.Component
TabBaricon
- name: String
- focused: Boolean
+ TabBarIcon(name: String, focused: Boolean)
+ render(): View
FlatView
- data: FlatViewData
- onPressItem: Function<void, Product>
- shoppingViewMode: Boolean
+ FlatView(data: FlatViewData, onPressItem: Function<void, Product>, shoppingViewMode: Boolean)
+ render(): View
- handleAddToShoppingList(context: AppContext, product: Product): void
- handleRemoveFromShoppingList(context: AppContext, product: Product): void
ComparisonView
- productName: String
- imageUrl: String
- description: String
+ ComparisonView(productName: String, imageUrl: String, description: String)
+ render(): View

Models Package

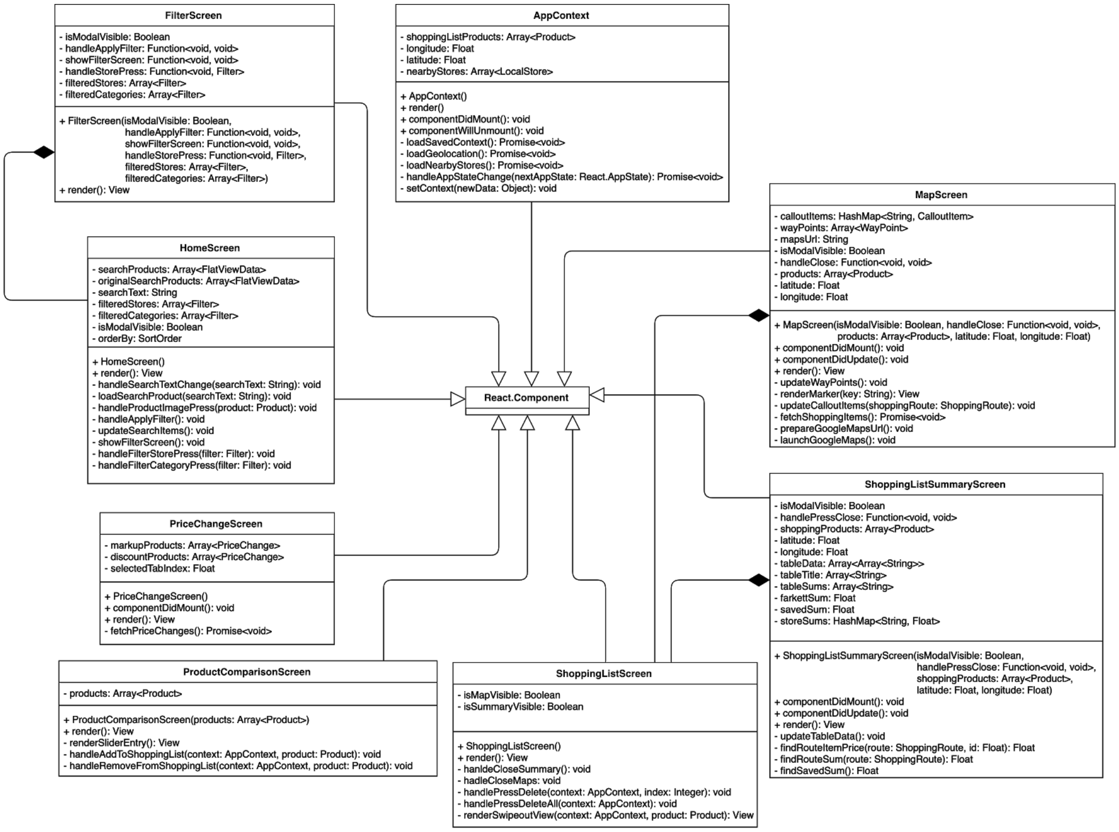
This package is responsible from translating and preprocessing data we have got from backend. Brand class handles the missing brand names received from the server. Product class converts unit and quantities into more human readable versions. For instance, in backend what would be expressed as 0.003 kilograms is displayed as 30 grams.ShoppingRoute class merges ProductToStore and StoreToLocation arrays and creates a single RouteItem object. ProductToStore is an array to hold product information and store information which the product is being sold. StoreToLocation is an array to match each store with a location. As a result shopping list is converted to a RouteItem array which holds the information of where to buy each product.


PriceChange
- product: Product
- oldPrice: Float
- priceChange: Float
- priceChangeRate: Float
+ PriceChange(product: Product, oldPrice: Float, priceChange: Float, priceChangeRate: Float)
+ fromBackendData(data: Object): PriceChange
LocalStore
- id: String
- longitude: Float
- latitude: Float
- distance: Float
- store: Store
+ LocalStore(id: String, longitude: Float, latitude: Float, distance: Float, store: Store)
+ fromBackendData(data: Object): LocalStore
+ fromPlacesAPIData(data: Object): LocalStore
Store
- id: String
- name: String
- hasWebStore: Boolean
+ Store(id: String, name: String, hasWebStore: Boolean
+ fromBackendData(data: Object): Store
Brand
- id: String
- name: String
+ Brand(id: String, name: String)
+ fromBackendData(data: Object): Brand
<<interface>>
Model
+ fromBackenData(data: Object): Model
Product
- id: String
- name: String
- imageUrl: String
- description: String
- quantity: Float
- unit: String
- type: ProductType
- brand: Brand
- store: Store
- price: Float
- pieces: String
- url: String
+ Product(id: String, name: String, imageUrl: String, description: String, quantity: Float, unit: String, type: ProductType, brand: Brand, store: Store, price: Float, pieces: String, url: String)
+ fromBackendData(data: Object): Product
ProductType
- id: String
- name: String
- description: String
- isLeaf: Boolean
- parentId: String
+ ProductType(id: String, name: String, description: String isLeaf: Boolean, parentId: String)
+ fromBackendData(data: Object): ProductType
ShoppingRoute
route: Array<RouteItem>
+ ShoppingRoute(route: Array<RouteItem>)
+ fromBackendData(data: Object): ShoppingRoute
RouteItem
- product: Product
- distance: Float
- latitude: Float
- longitude: Float
- placeId: String

Screens Package

This package is responsible from representing the application's main screens. HomeScreen welcomes the users and it has components such as search bar, filter and sort button. After taking a search text, it shows products in a horizontal list. FilterScreen provides checkboxes for each filter item. After pressing the apply button, FilterScreen is closed and filter will be applied for the search items. PriceChangeScreen shows markup and discounts as a list. To compare different prices for the same product, ProductComparisonScreen shows a carousel for different price options. ShoppingListScreen is a simple screen to show products which were added to the shopping list from HomeScreen or ProductComparisonScreen. MapScreen is responsible from showing the shopping route on a map. Markers are used to pin the store locations and each marker shows individual shopping list after a press. Finally, ShoppingListSummaryScreen shows price alternatives for each product in a table format as well as individual sums for each store.

Note that AppContext provides global data that is shared across screens. The context API is provided by React and it basically helps re-rendering all the different components when one of the global data changes.


FilterScreen
- isModalVisible: Boolean
- handleApplyFilter: Function<void, void>
- showFilterScreen: Function<void, void>
- handleStorePress: Function<void, Filter>
- filteredStores: Array<Filter>
- filteredCategories: Array<Filter>
+ FilterScreen(isModalVisible: Boolean, handleApplyFilter: Function<void, void>, showFilterScreen: Function<void, void>, handleStorePress: Function<void, Filter>, filteredStores: Array<Filter>, filteredCategories: Array<Filter>)
+ render(): View
AppContext
- shoppingListProducts: Array<Product>
- longitude: Float
- latitude: Float
- nearbyStores: Array<LocalStore>
+ AppContext()
+ render()
+ componentDidMount(): void
+ componentWillUnmount(): void
- loadSavedContext(): Promise<void>
- loadGeolocation(): Promise<void>
- loadNearbyStores(): Promise<void>
- handleAppStateChange(nextAppState: React.AppState): Promise<void>
- setContext(newData: Object): void
MapScreen
- calloutItems: HashMap<String, CalloutItem>
- wayPoints: Array<WayPoint>
- mapsUrl: String
- isModalVisible: Boolean
- handleClose: Function<void, void>
- products: Array<Product>
- latitude: Float
- longitude: Float
+ MapScreen(isModalVisible: Boolean, handleClose: Function<void, void>, products: Array<Product>, latitude: Float, longitude: Float)
+ componentDidMount(): void
+ componentDidUpdate(): void
+ render(): View
- updateWayPoints(): void
- renderMarker(key: String): View
- updateCalloutItems(shoppingRoute: ShoppingRoute): void
- fetchShoppingItems(): Promise<void>
- prepareGoogleMapsUrl(): void
- launchGoogleMaps(): void
HomeScreen
- searchProducts: Array<FlatViewData>
- originalSearchProducts: Array<FlatViewData>
- searchText: String
- filteredStores: Array<Filter>
- filteredCategories: Array<Filter>
- isModalVisible: Boolean
- orderBy: SortOrder
+ HomeScreen()
+ render(): View
- handleSearchTextChange(searchText: String): void
- loadSearchProduct(searchText: String): void
- handleProductImagePress(product: Product): void
- handleApplyFilter(): void
- updateSearchItems(): void
- showFilterScreen(): void
- handleFilterStorePress(filter: Filter): void
- handleFilterCategoryPress(filter: Filter): void
React.Component
ShoppingListSummaryScreen
- isModalVisible: Boolean
- handlePressClose: Function<void, void>
- shoppingProducts: Array<Product>
- latitude: Float
- longitude: Float
- tableData: Array<Array<String>>
- tableTitle: Array<String>
- tableSums: Array<String>
- farkettSum: Float
- savedSum: Float
- storeSums: HashMap<String, Float>
+ ShoppingListSummaryScreen(isModalVisible: Boolean, handlePressClose: Function<void, void>, shoppingProducts: Array<Product>, latitude: Float, longitude: Float)
+ componentDidMount(): void
+ componentDidUpdate(): void
+ render(): View
- updateTableData(): void
- findRouteItemPrice(route: ShoppingRoute, id: Float): Float
- findRouteSum(route: ShoppingRoute): Float
- findSavedSum(): Float
PriceChangeScreen
- markupProducts: Array<PriceChange>
- discountProducts: Array<PriceChange>
- selectedTabIndex: Float
+ PriceChangeScreen()
+ componentDidMount(): void
+ render(): View
- fetchPriceChanges(): Promise<void>
ProductComparisonScreen
- products: Array<Product>
+ ProductComparisonScreen(products: Array<Product>)
+ render(): View
- renderSliderEntry(): View
- handleAddToShoppingList(context: AppContext, product: Product): void
- handleRemoveFromShoppingList(context: AppContext, product: Product): void
ShoppingListScreen
- isMapVisible: Boolean
- isSummaryVisible: Boolean
+ ShoppingListScreen()
+ render(): View
- hanldeCloseSummary(): void
- hadleCloseMaps: void
- handlePressDelete(context: AppContext, index: Integer): void
- handlePressDeleteAll(context: AppContext): void
- renderSwipeoutView(context: AppContext, product: Product): View

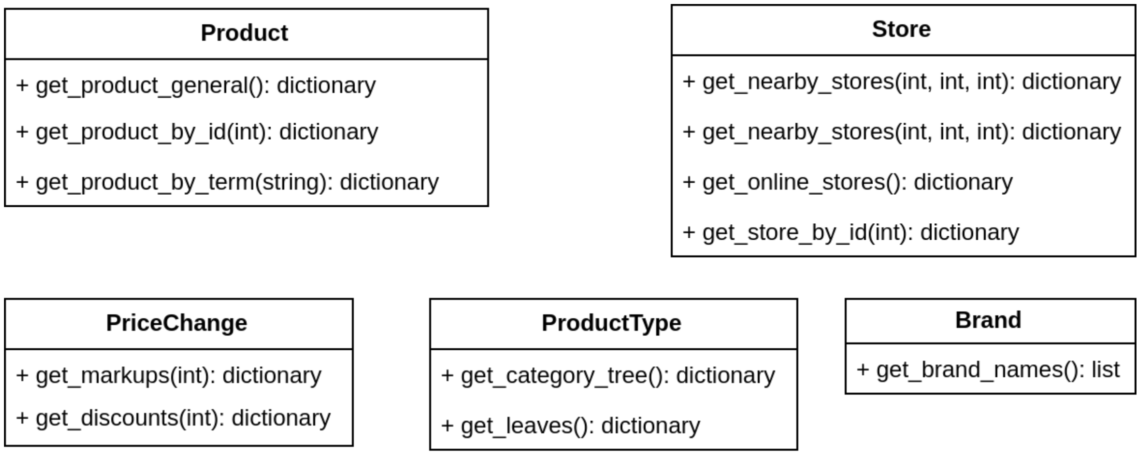
Back-end

Farkett’s backend implementation includes the logical backbone and the data interactions of the application. Communication with the client is achieved via HTTP on a uniform REST API. The backend consists of eight packages serving different purposes and a server module called app.py. In the following, these packages will be explained, and final low level design diagrams will be shown.

Note that all functions are implemented as static because of reusability and simplicity purposes. Therefore, classes in the backend are only used for organizing these functions, like function containers, they don’t hold state by themselves. Python literature is used here since we used Python in the backend. The term “dictionary” stands for HashMaps in Java language and “list” stands for ArrayLists.

Repository package

Repository package is responsible for fetching data from the SQLite database. Every class here abstracts SQL queries corresponding to database table with the class’ name so that other parts of the application will have easy access to data. For example functions in the class *Store* contains queries linked to Store table in the database. Classes in the repository package:



Product

+ get_product_general(): dictionary
+ get_product_by_id(int): dictionary
+ get_product_by_term(string): dictionary

Store

+ get_nearby_stores(int, int, int): dictionary
+ get_nearby_stores(int, int, int): dictionary
+ get_online_stores(): dictionary
+ get_store_by_id(int): dictionary

PriceChange

+ get_markups(int): dictionary
+ get_discounts(int): dictionary

ProductType

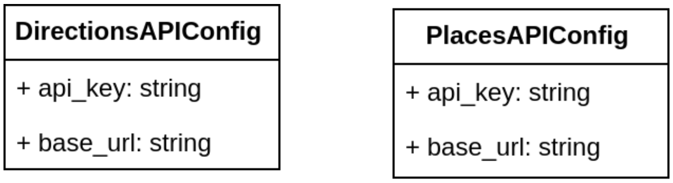
+ get_category_tree(): dictionary
+ get_leaves(): dictionary

Brand

+ get_brand_names(): list

Api-utils package

Api-utils package contains constants for Places API and Directions API which we use in order to find nearby stores and create shopping routes. Classes in the api-utils package:



DirectionsAPIConfig

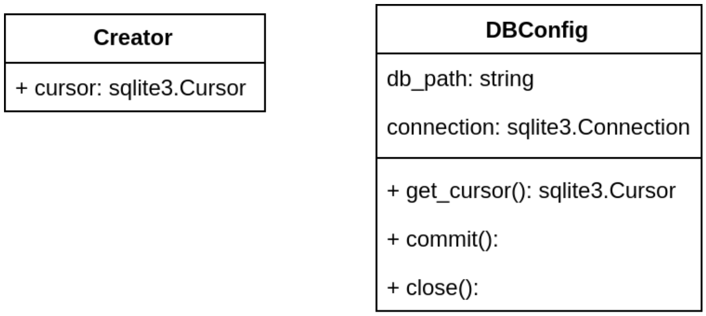
+ api_key: string
+ base_url: string

PlacesAPIConfig

+ api_key: string
+ base_url: string

Db-utils package

Db-utils package is responsible for creating the internal structure of the database and accessing to the database. Since Farkett will serve many clients it is a necessity to manage connections to database. Classes in the db-utils package:



Creator

+ cursor: sqlite3.Cursor

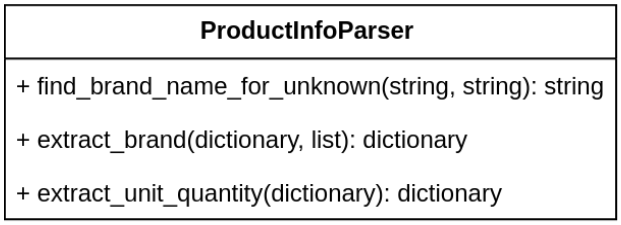
DBConfig

db_path: string
connection: sqlite3.Connection

+ get_cursor(): sqlite3.Cursor
+ commit():
+ close():

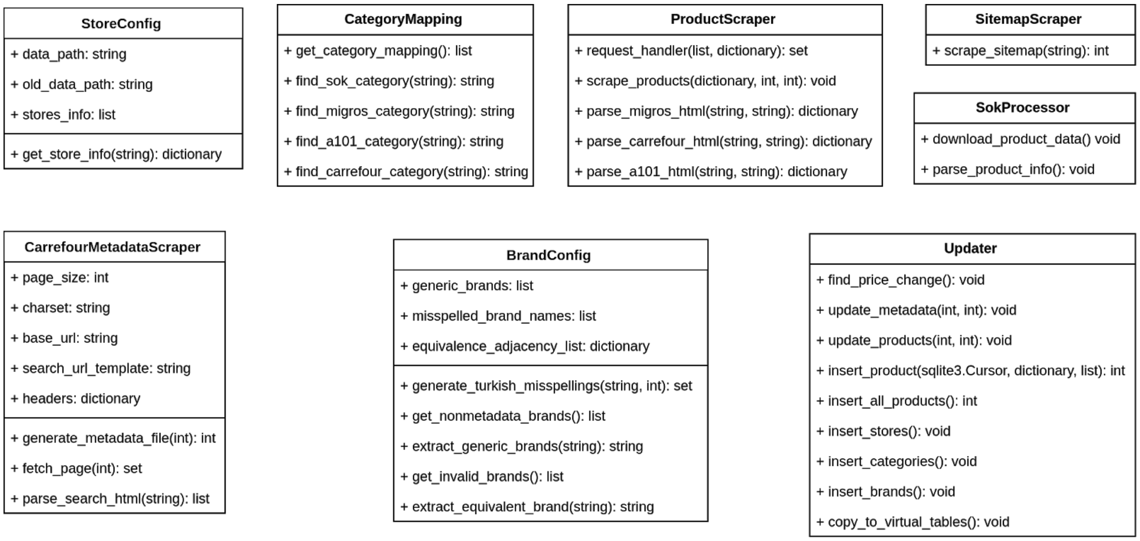
Normalization package

Normalization package has a single module named *ProductInfoParser* which extracts missing data fields of a product from its name. These data fields are brand name, unit and quantity. Low level design of the normalization package:

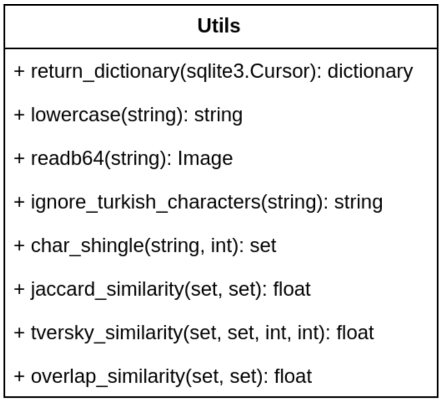

ProductInfoParser
+ find_brand_name_for_unknown(string, string): string
+ extract_brand(dictionary, list): dictionary
+ extract_unit_quantity(dictionary): dictionary

Crawler package
Crawler package is responsible for gathering data from online store websites and populating the database. In order to gather data from Migros and A101, we parse store sitemaps which contain links to product pages. For Şok and Carrefour we have to use the actual product search since these stores' sitemaps have missing products. For scraping product links we used grequests library which allowed us to send concurrent HTTP requests. To parse product HTMLs we used BeautifulSoup library.

Same products coming from different stores have to be matched before updating the database, which we to call "normalization" process. However there are missing data fields, and noise in the incoming product data, and every store uses a distinct naming convention. Noise such as misspellings are corrected in this package. By utilizing *normalization* package, missing data fields such as brand names, units, quantities etc. are inferred and fresh data is inserted into the database. Classes in the crawler package:


StoreConfig
+ data_path: string
+ old_data_path: string
+ stores_info: list
+ get_store_info(string): dictionary
CategoryMapping
+ get_category_mapping(): list
+ find_sok_category(string): string
+ find_migros_category(string): string
+ find_a101_category(string): string
+ find_carrefour_category(string): string
ProductScraper
+ request_handler(list, dictionary): set
+ scrape_products(dictionary, int, int): void
+ parse_migros_html(string, string): dictionary
+ parse_carrefour_html(string, string): dictionary
+ parse_a101_html(string, string): dictionary
SitemapScraper
+ scrape_sitemap(string): int
SokProcessor
+ download_product_data() void
+ parse_product_info(): void
CarrefourMetadataScraper
+ page_size: int
+ charset: string
+ base_url: string
+ search_url_template: string
+ headers: dictionary
+ generate_metadata_file(int): int
+ fetch_page(int): set
+ parse_search_html(string): list
BrandConfig
+ generic_brands: list
+ misspelled_brand_names: list
+ equivalence_adjacency_list: dictionary
+ generate_turkish_misspellings(string, int): set
+ get_nonmetadata_brands(): list
+ extract_generic_brands(string): string
+ get_invalid_brands(): list
+ extract_equivalent_brand(string): string
Updater
+ find_price_change(): void
+ update_metadata(int, int): void
+ update_products(int, int): void
+ insert_product(sqlite3.Cursor, dictionary, list): int
+ insert_all_products(): int
+ insert_stores(): void
+ insert_categories(): void
+ insert_brands(): void
+ copy_to_virtual_tables(): void

Utils package
Utils is a multi purpose package with a single module with the same name. Functionality which is not explicitly linked to other packages are put in here. Low level design of the utils package:

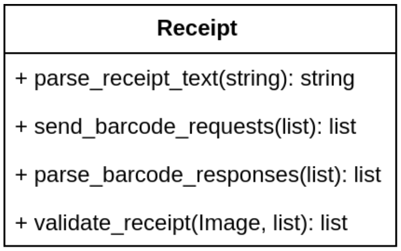


Utils

+ return_dictionary(sqlite3.Cursor): dictionary
+ lowercase(string): string
+ readb64(string): Image
+ ignore_turkish_characters(string): string
+ char_shingle(string, int): set
+ jaccard_similarity(set, set): float
+ tversky_similarity(set, set, int, int): float
+ overlap_similarity(set, set): float

Receipts package

This package contains functionality that is required to recognize characters on a receipt, extracting barcodes in the receipt text and finding products corresponding to these barcodes. For optical character recognition Tesseract is used. Low level design of the receipts package:

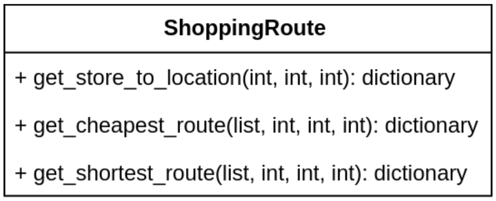


Receipt

+ parse_receipt_text(string): string
+ send_barcode_requests(list): list
+ parse_barcode_responses(list): list
+ validate_receipt(Image, list): list

Route-maker package

This package is responsible for calculating cheapest and shortest routes for specified shopping lists and locations using Directions API. Low level design of the receipts route-maker package:

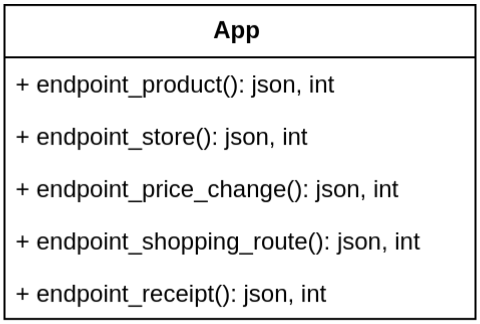


ShoppingRoute

+ get_store_to_location(int, int, int): dictionary
+ get_cheapest_route(list, int, int, int): dictionary
+ get_shortest_route(list, int, int, int): dictionary

App module

This module includes server endpoints which accepts requests from the client and responses accordingly. Low level design of the app module.


App
+ endpoint_product(): json, int
+ endpoint_store(): json, int
+ endpoint_price_change(): json, int
+ endpoint_shopping_route(): json, int
+ endpoint_receipt(): json, int

Contextual Impacts

Short Term Impacts

Global

There is no global short term impact of Farkett since it will be released only in Turkey in short term. However, in long term if the project is successful, there are plans to increase project scope to a global scale.

Economic

Farkett will help people on tight-budget enormously on short-term. With its expected profits of around 10-15% in every grocery trip, elderly, students and homemakers will finally be able to get some relief economical-wise.

Environmental

Environmentally, if we are to assume people are driving for grocery shopping once a week to a store that is at 10 km afar and if we have 10.000 regular customers in the short term, with the usage of courier system 1 courier delivers to 10 customers with a 20 km trip, each week there will be $1000 * 20\ km$ instead of $10.000 * 10\ km$ trips, $80.000\ km's$ of gas saved, resulting in a happier environment. It corresponds to $80.000 / 7\ \approx\ 11.500\ km$, around $720\ liters$ of fuel saved daily.

Societal

It is a known fact that economical conditions are closely tied to social conditions, it is hard to expect one to go well when other is in a miserable condition. These metrics are also connected to general happiness of citizens, crime rates, education rates, level of prosperity. If lower class is able to afford more with the same budget, this will lead to a higher happiness and more satisfaction from life.

Long Term Vision

Global

On the long term, not only Turkish people but people all around the world, starting with USA and Europe, will have access to Farkett, therefore cheaper products. This will result in the whole word taking advantage of economic, environmental and societal benefits that we have listed on long term.

Economic

If Farkett is used extensively by many users in the long term, Farkett courier system will create many new jobs (up to 50.000 new freelancer courier jobs just like in Uber at 500.000 regular customers), in turn reducing the unemployment rate.

Environmental

Similar to the calculation above, if we have 500.000 regular customers in the long term, with the usage of courier system 1 courier delivers to 10 customers with a 20 km trip, each week there will be $50.000 * 20\ km$ instead of $500.000 * 10\ km$ trips, $4.000.000\ km's$ of gas saved, resulting in a much more happier environment. It corresponds to $4.000.000 / 7 \approx 571.500\ km$ of gas saved daily. With on average 16 km per 1 liter of fuel travelled, $35.7\ tonnes$ of fuel is saved daily.

Societal

In the long term, having a higher affordability will result in lower crime rates, higher education levels, higher level of prosperity, happier citizens. This will create a positive feedback loop and it will continue to get better in each loop.

Contemporary Issues Related with the Area of the Project

Market’s Issues

It is possible that supermarkets that we are gathering data from and by this method offering the cheapest products to our customers will not like the fact that we make them compete with each other in a more harsh way by looking at the cheapest of the cheapest products only, leading them to file lawsuits against Farkett, or finding a way to not share their data such as closing their online stores completely.

On the other hand, it is completely legal what we are currently offering to our customers and no lawsuit will possibly deny Farkett’s services to its customers. Also, the supermarkets that are focused on providing the cheapest products to their customers will benefit greatly from Farkett’s services, so there will still be no problem for Farkett if supermarkets with other intentions stop providing their services. Farkett will work hand-to-hand with discount markets to provide the best services to its customers.

New Tools and Technologies Used

SQLite

SQLite was used to create the database. SQLite was a good choice because it is cross platform and runs fast with multiple supports for triggers, views and many more functionalities that we needed for the project.

Python

Python was used while developing the backend of Farkett. Python was a fast and reliable option for us to use. It is highly flexible and it’s syntax is almost as easy as a natural language. However, the error prone nature of the Python caused some errors. One of them was the complexity of the debugging.

Flask

Flask was used as a webframe. It is developed for Python and it is much more lightweight compared to its alternatives.

Tesseract

Tesseract was used for image to text conversion. Tesseract OCR engine was developed by Google to recognize the characters embedded in images. It serves as a good convertor for receipt information to digital text information to be used in our project.

Grequests

Grequests is an asynchronous http request library. It helped us to send concurrent http requests while collecting our data and made the whole process much more faster. We provide developers concurrent requests parameters while retrieving data and make it possible to send more than one at a single time.

Beautiful Soup

Beautiful Soup is a HTML parsing library. The whole data collection was dependent on this library. Beautiful Soup handles string matching operations fast and makes it easier to use generalized queries to obtain relevant fields from the HTML text.

OpenCV

OpenCV was used for image preprocessing. OpenCV provides optimized and fast algorithms for image processing and computer vision. We used it on the receipt scanning operations. It served its purpose while detecting a perfect rectangular receipt photo and removing shadows from the receipt image.

Git

We used Git to keep track of the whole development process. New branches for different functionalities were opened and then merged on the develop branch. New version releases were made in the master branch. This process helped us to keep track of the development cycle and made it much more easier to handle any possible conflicts.

Google Directions API

Google Directions API was used while drawing an optimized shopping route for the user. It connects two different locations given with an optimized manner.

Google Places API

Google Places API was used while recognizing local markets near the user. Nearby Migros, Şok, A101, Carrefour markets were found by using this API.

Google Maps API

Google Maps API was used for opening the GPS service for the user if the user wanted to keep track of the optimized route.

React Native

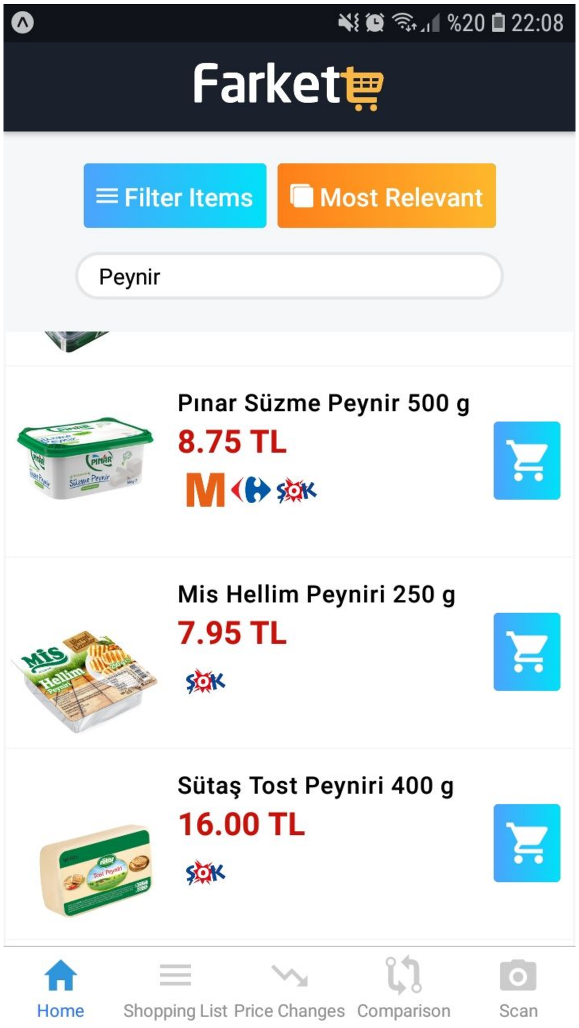
React Native was used to develop the front end of the Farkett. It uses native components of both iOS and Android while combining them using JavaScript rather than making a web applications unlike its other alternative React. Using React Native was a good design decision because it allowed us to benefit from functionalities of both Android and iOS while developing a cross platform application.

Hardware/Software System

Farkett application consist one server and one client. Communication between them is handled with endpoint functions using HTTP requests. Client requests relevant information from the server side using these endpoint functions, like searching a keyword, sending receipt image etc. Endpoints uses JSON format for relevant information transfers. This procedure ensures no direct connection between client and database, which is really important to encapsulate the system components. Server side receives information in JSON format from the client as explained before, and does the relevant operations according to them. Then sends its response in the same format to client side. Server side has a direct communication with the database and acts as a managing engine for it. The functionalities which are responsible for crawling, parsing and inserting normalized products to the database is called with the requests of the server. The server side runs libraries that the app depends on like Tesseract, OpenCV, Google Services etc. Relevant information received from them are processed and delivered to the client side.

Users Manual

Home Screen



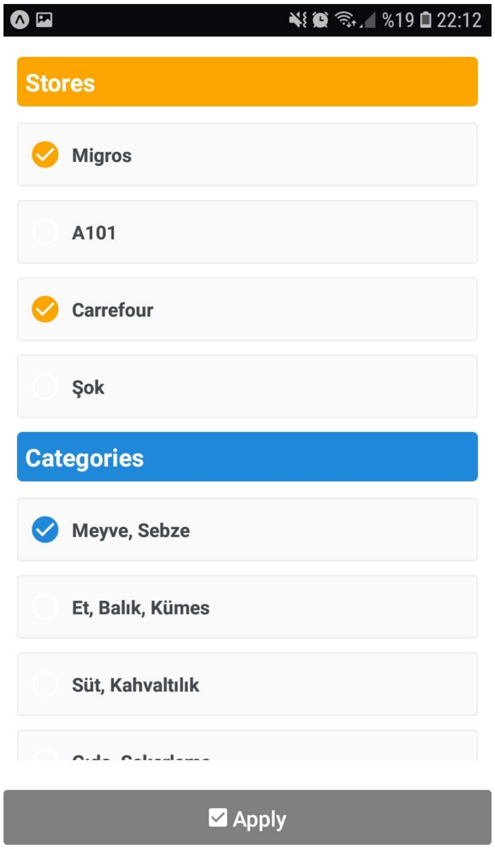
Peynir

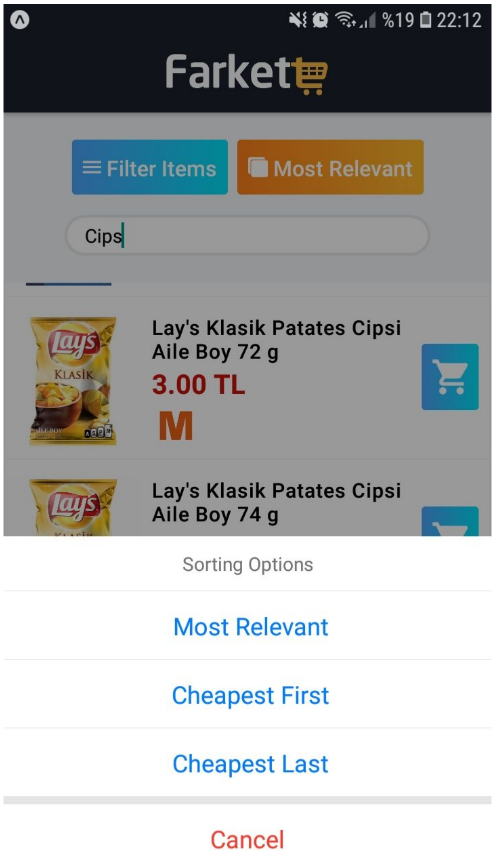
Pınar Süzme Peynir 500 g
8.75 TL

Mis Hellim Peyniri 250 g
7.95 TL

Sütaş Tost Peyniri 400 g
16.00 TL

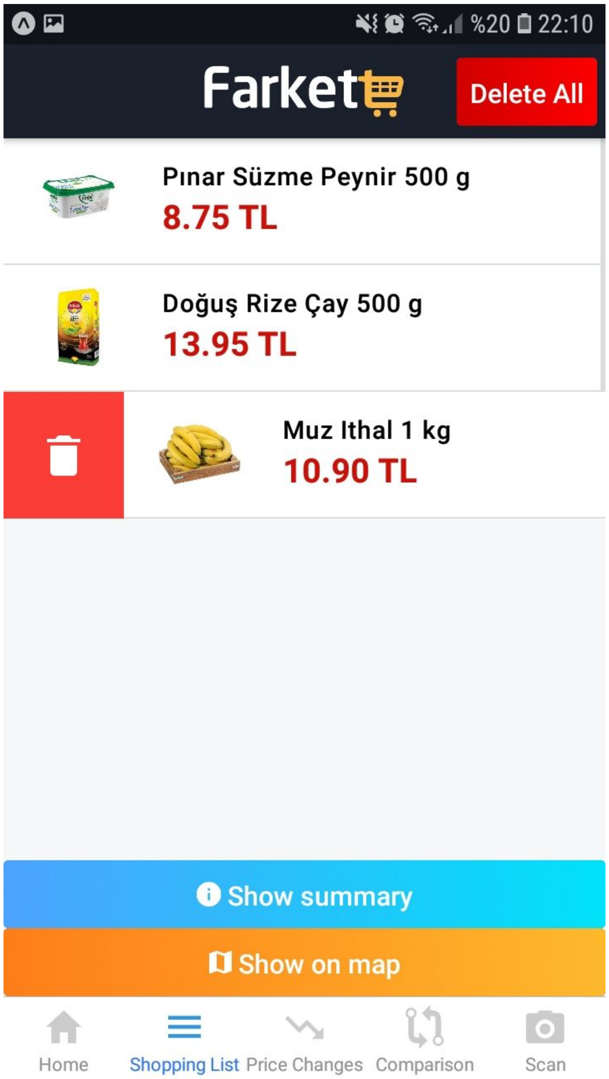
Users can search for products from the home screen. A list of products will be shown. A list item consists of product name, product price, product image, store names which the product is being sold and a button to add product to the shopping list. To see the different prices from stores, users can press the product and navigate to comparison screen.





To make more efficient searches, users can use store and category filters. Users can also sort items by their price and relevance.

Shopping List Screen

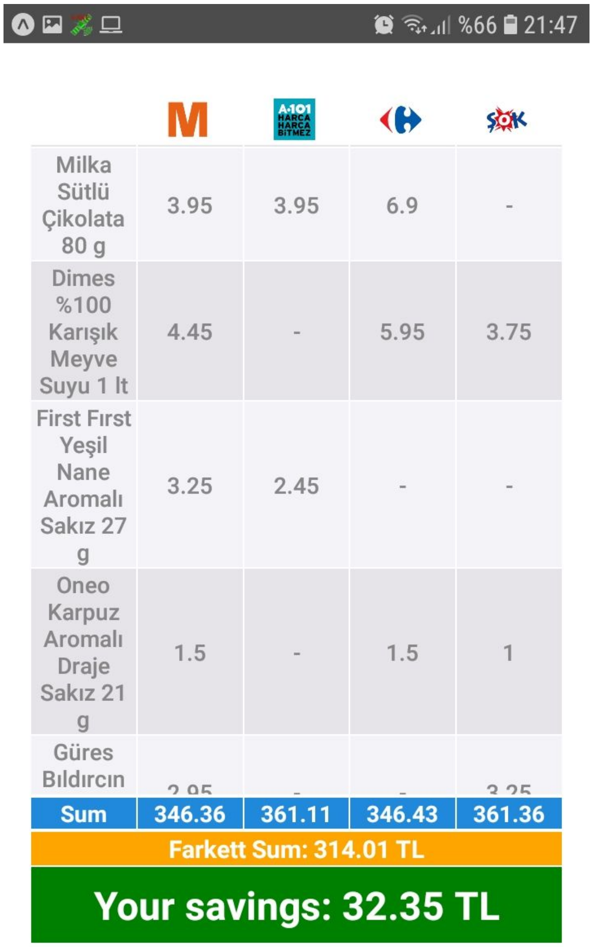


Pınar Süzme Peynir 500 g
8.75 TL

Doğuş Rize Çay 500 g
13.95 TL

Muz Ithal 1 kg
10.90 TL

Users can see their shopping list after adding products from the home screen. They can swipe left or right to delete items.



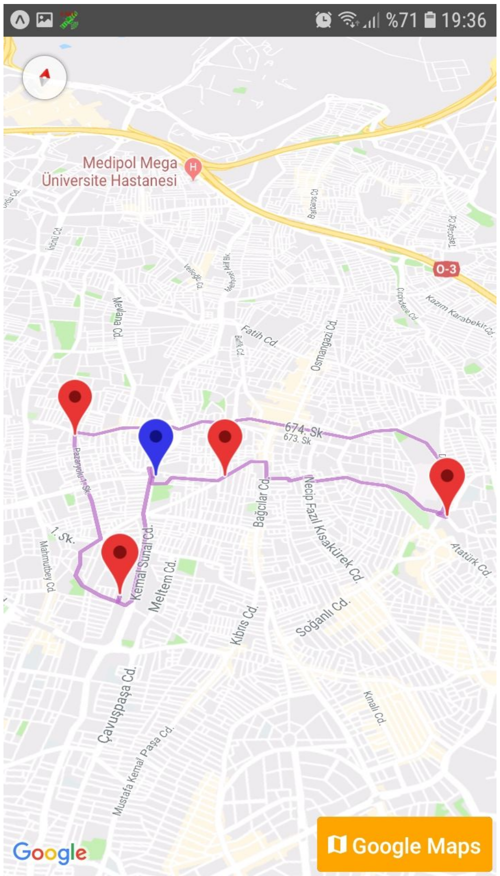
%66 21:47

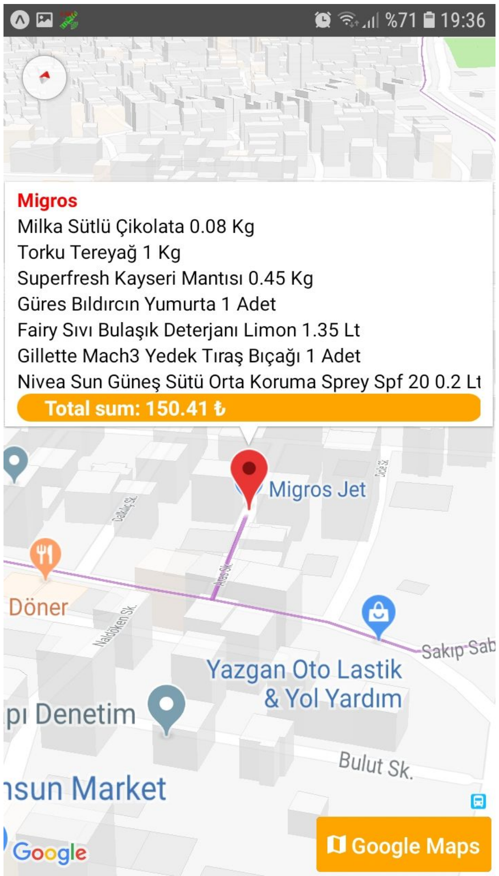
	M	A101 HARCA HARCA BİTMEZ		ŞOK
Milka Sütlü Çikolata 80 g	3.95	3.95	6.9	-
Dimes %100 Karışık Meyve Suyu 1 lt	4.45	-	5.95	3.75
First Fırst Yeşil Nane Aromalı Sakız 27 g	3.25	2.45	-	-
Oneo Karpuz Aromalı Draje Sakız 21 g	1.5	-	1.5	1
Güres Bıldırcın	2.95	-	-	3.25
Sum	346.36	361.11	346.43	361.36

Farkett Sum: 314.01 TL

Your savings: 32.35 TL

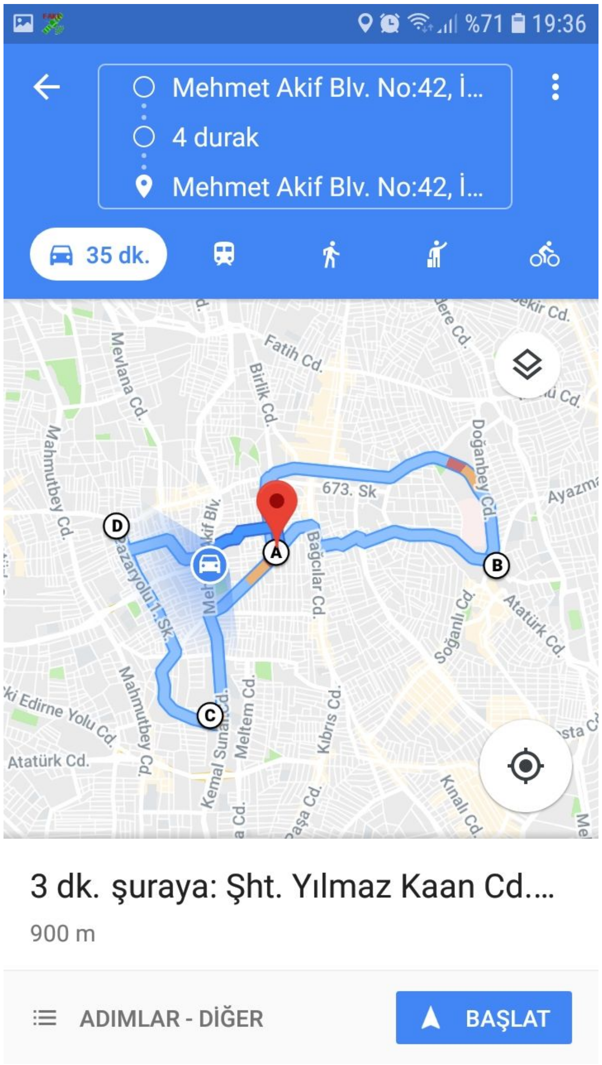
After pressing the show summary button from the shopping list screen, users can see their total savings. Individual sums are calculated by cumulating each column. The dash (-) symbol represents absence of a product at a particular store. In this case, minimum value of the row is added to the individual sum. Farkett sum is the optimal sum that we offer for our users. Your savings label gives the price difference between Farkett sum and minimum of the individual column sums.





Migros
Milka Sütlü Çikolata 0.08 Kg
Torku Tereyağ 1 Kg
Superfresh Kayseri Mantısı 0.45 Kg
Güres Bıldırcın Yumurta 1 Adet
Fairy Sıvı Bulaşık Deterjanı Limon 1.35 Lt
Gillette Mach3 Yedek Tıraş Bıçağı 1 Adet
Nivea Sun Güneş Sütü Orta Koruma Sprey Spf 20 0.2 Lt
Total sum: 150.41 ₺

After pressing the show on map button from the shopping list screen, users can see their optimal shopping route. The blue pin corresponds to the user's location and the red pins correspond to stores. By pressing the red pins, users can see what to buy from each store.



Mehmet Akif Blv. No:42, İ...

4 durak

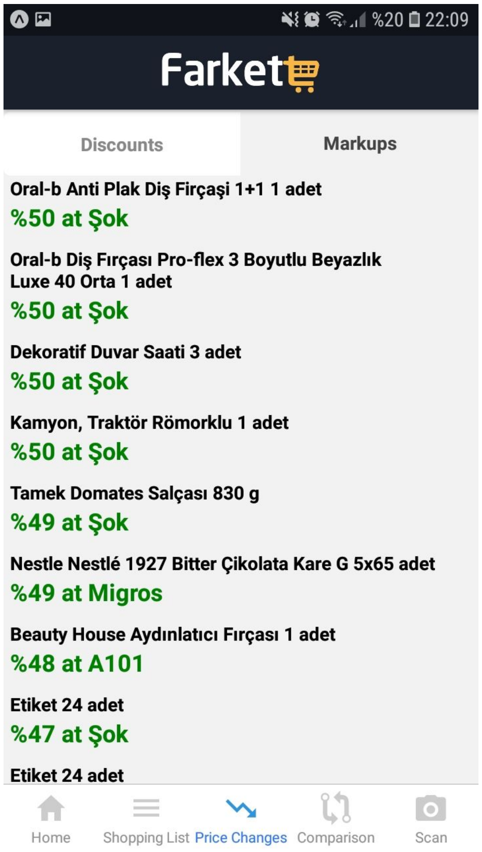
Mehmet Akif Blv. No:42, İ...

3 dk. şuraya: Şht. Yılmaz Kaan Cd.…

900 m

After pressing the Google maps button from the maps screen, Google Maps application will be launched to give navigation for the optimal shopping route.

Markup/Discount Screen



Oral-b Anti Plak Diş Firçaşi 1+1 1 adet
%50 at Şok

Oral-b Diş Fırçası Pro-flex 3 Boyutlu Beyazlık Luxe 40 Orta 1 adet
%50 at Şok

Dekoratif Duvar Saati 3 adet
%50 at Şok

Kamyon, Traktör Römorklu 1 adet
%50 at Şok

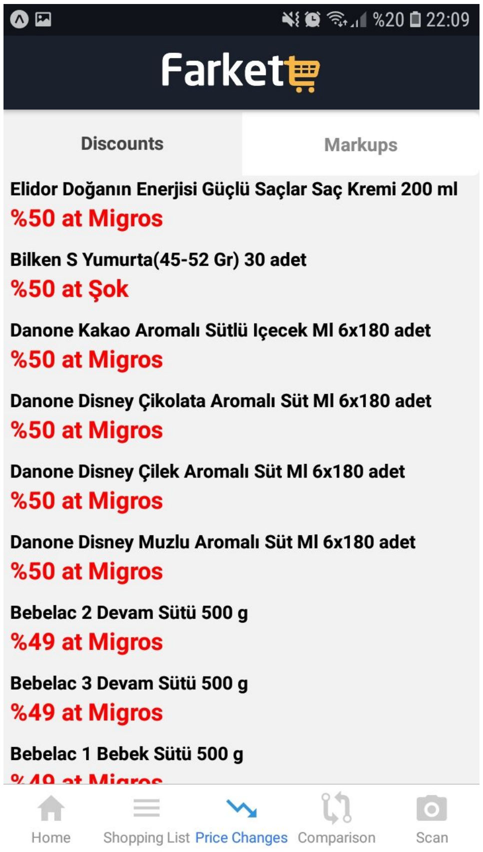
Tamek Domates Salçası 830 g
%49 at Şok

Nestle Nestlé 1927 Bitter Çikolata Kare G 5x65 adet
%49 at Migros

Beauty House Aydınlatıcı Fırçası 1 adet
%48 at A101

Etiket 24 adet
%47 at Şok

Etiket 24 adet



Elidor Doğanın Enerjisi Güçlü Saçlar Saç Kremi 200 ml
%50 at Migros

Bilken S Yumurta(45-52 Gr) 30 adet
%50 at Şok

Danone Kakao Aromalı Sütlü Içecek Ml 6x180 adet
%50 at Migros

Danone Disney Çikolata Aromalı Süt Ml 6x180 adet
%50 at Migros

Danone Disney Çilek Aromalı Süt Ml 6x180 adet
%50 at Migros

Danone Disney Muzlu Aromalı Süt Ml 6x180 adet
%50 at Migros

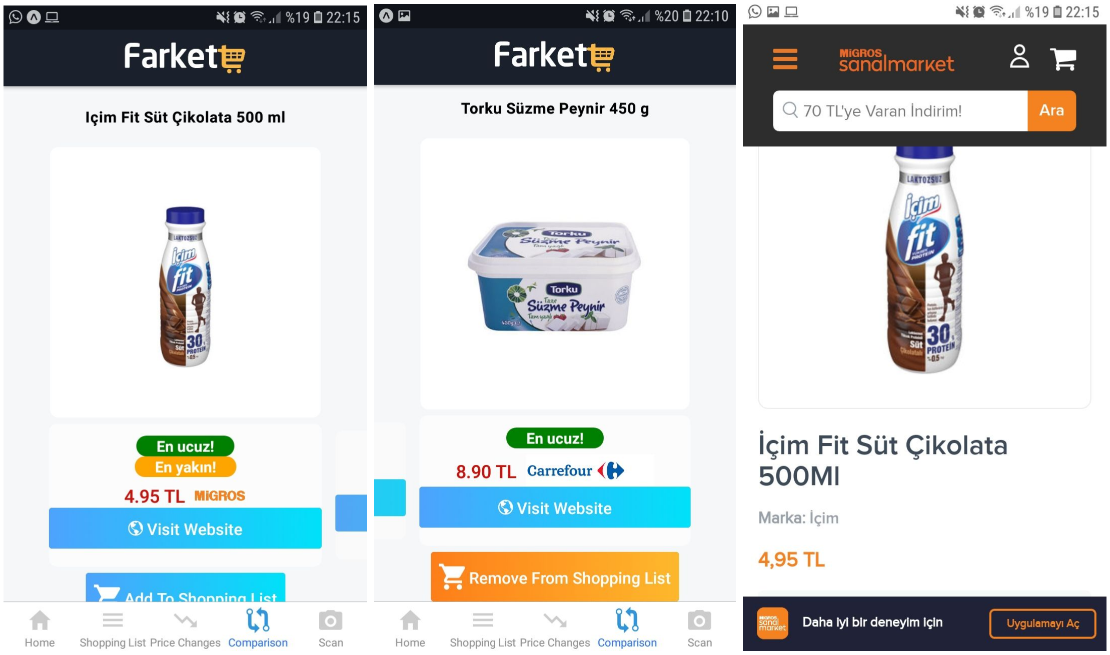
Bebelac 2 Devam Sütü 500 g
%49 at Migros

Bebelac 3 Devam Sütü 500 g
%49 at Migros

Bebelac 1 Bebek Sütü 500 g
%49 at Migros

Users can see price changes of the products after pressing the price changes button from the navigation bar. Discounts and markups can be seen separately by pressing the relevant tab at top of the screen.

Product Comparison Screen



Içim Fit Süt Çikolata 500 ml

En ucuz!
En yakın!
4.95 TL MiGROS

Torku Süzme Peynir 450 g

En ucuz!
8.90 TL Carrefour

İçim Fit Süt Çikolata 500Ml

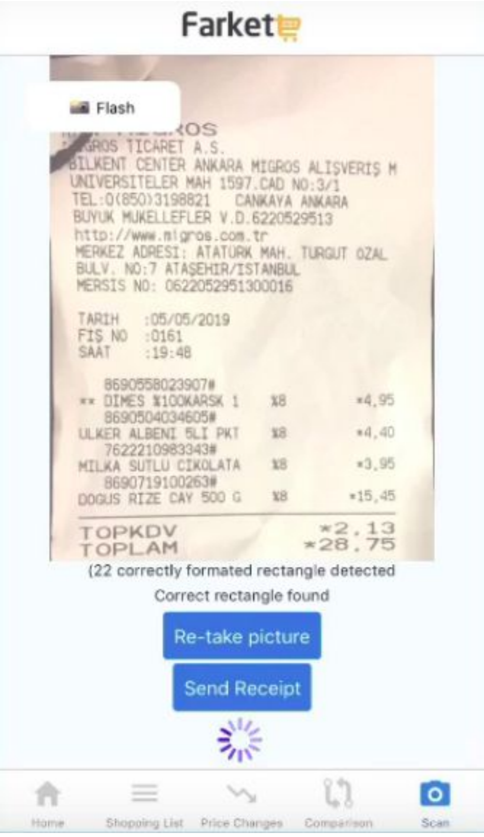
Marka: İçim

4,95 TL

Daha iyi bir deneyim için

Users can press a product from the home screen to compare price alternatives from different stores. They can also add a product to their shopping list from this screen. Moreover, to verify the price of the product, they can press visit website button to visit the store’s website about the product.

Receipt Scanning Screen



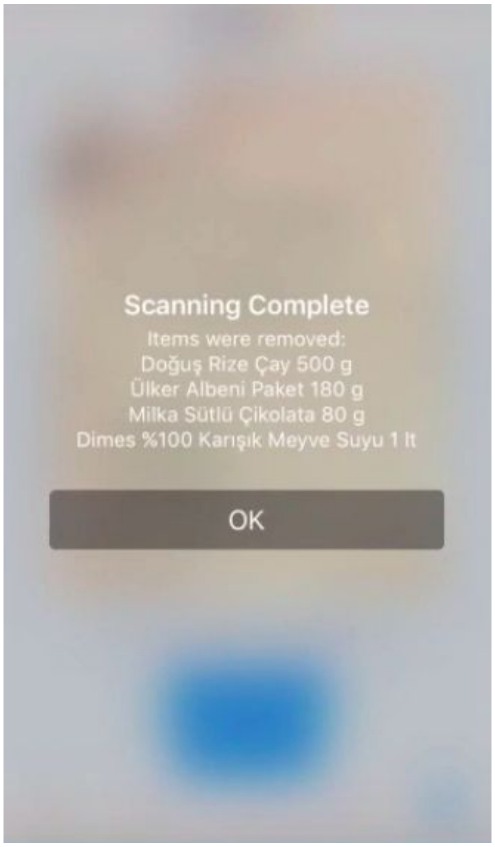
```
ROS
GROS TICARET A.S.
BILKENT CENTER ANKARA MIGROS ALIŞVERIŞ M
UNIVERSITELER MAH 1597.CAD NO:3/1
TEL:0(850)3198821    CANKAYA ANKARA
BUYUK MUKELLEFLER V.D.6220529513
http://www.migros.com.tr
MERKEZ ADRESI: ATATURK MAH. TURGUT OZAL
BULV. NO:7 ATAŞEHIR/ISTANBUL
MERSIS NO: 0622052951300016

TARIH   :05/05/2019
FIŞ NO  :0161
SAAT    :19:48

   8690558023907#
** DIMES %100KARSK 1    %8        *4,95
   8690504034605#
ULKER ALBENI 5LI PKT    %8        *4,40
   7622210983343#
MILKA SUTLU CIKOLATA    %8        *3,95
   8690719100263#
DOGUS RIZE CAY 500 G    %8       *15,45
TOPKDV                            *2.13
TOPLAM                           *28.75
```

(22 correctly formated rectangle detected

Correct rectangle found



Scanning Complete

Items were removed:

Doğuş Rize Çay 500 g

Ülker Albeni Paket 180 g

Milka Sütlü Çikolata 80 g

Dimes %100 Karışık Meyve Suyu 1 lt

From this screen, users can scan their shopping receipt to delete corresponding items from their shopping list. A dialog will be shown to notify users about which products are identified from their scan.