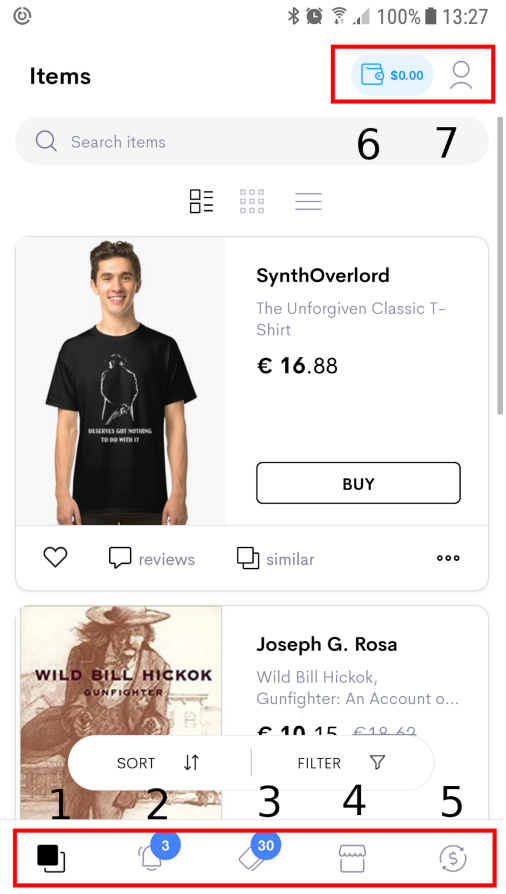
A Case Study of Using AWS Serverless to Implement a Shopping App

By Marcio Gualtieri

Overview

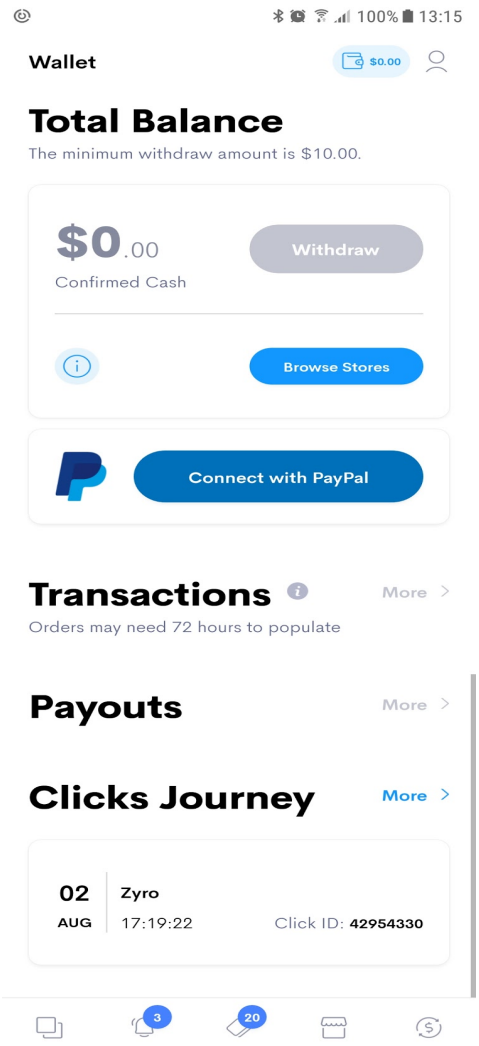
The purpose of this article is to serve as a case study for the architecture design of the back-end service of a shopping app using Amazon's Serverless components, taking into consideration non-functional features such as scalability, speed and cost.

To define the scope of features of our back-end, we have chosen the "Shoptagr" app, which consists of a single-page mobile app. Its main screen presents the following tabs, which one navigating to a specific screen:



At the Bottom	At the Top
1. Items. 2. Notifications. 3. Coupons. 4. Stores. 5. Cashback Picks.	6. Paypal Wallet. 7. (User) Profile.

"Paypal Wallet" is out of the scope of the back-end, as it can be directly integrated to the app using [Paypal's SDK](#).



Note the bottom "Browse Stores" and "Click Journey". In my opinion, these don't belong there.

"Browse Stores" redirects to the "Stores" tab, which is easily accessible at the bottom, and "Click Journey" would be a better fit to a dedicated tab or maybe for the "Profile" tab.

This doesn't change the back-end design, but I thought that would be interesting to comment on it.

It's important to mention that our main objective is not to reverse engineer the app, but to use this app analysis as a playground to discuss the best approaches to define a back-end serverless architecture using Amazon Web Services, particularly serverless.

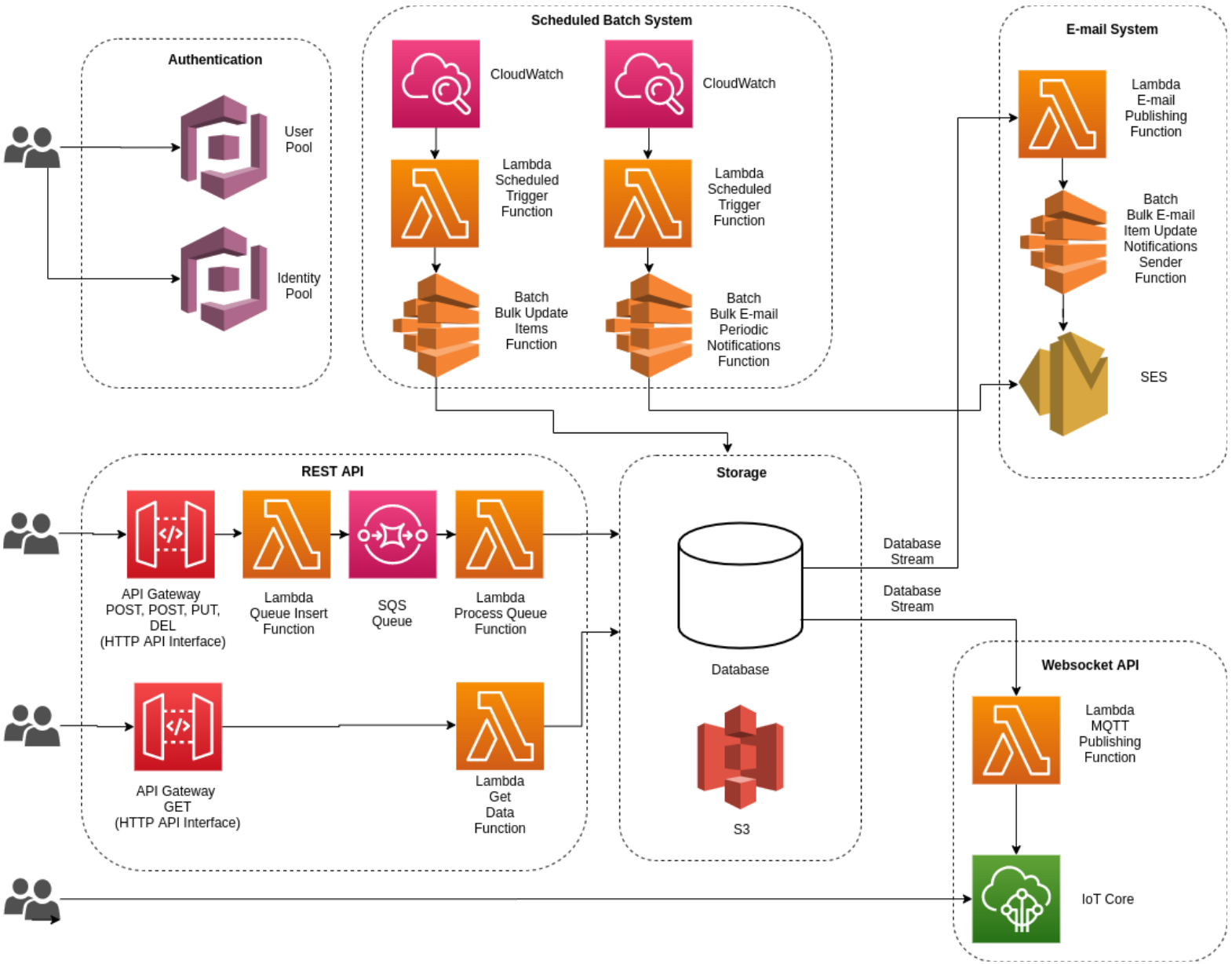
Basic Design

The objective of this section is to describe some of the basics components that comprise our back-end service and justifications on our choices. Some of these patterns repeat over and over (specifically for the REST API resources), so as a way to fit the overall design in a single diagram, this section will introduce these basic patterns.

The only details the diagram is missing are related to the REST API, specifically the resources and their respective data schemas. We will detail it in a proper section that will follow (using Swagger).

The overall design can be broken into the following basic patterns defined in the diagram that follows afterward:

1. **Authentication.**
2. **REST API.**
3. **Websocket API.**
4. **E-mail System.**
5. **Scheduled Batch System.**
6. **Storage.**


Authentication
User Pool
Identity Pool
Scheduled Batch System
CloudWatch
CloudWatch
Lambda Scheduled Trigger Function
Lambda Scheduled Trigger Function
Batch Bulk Update Items Function
Batch Bulk E-mail Periodic Notifications Function
E-mail System
Lambda E-mail Publishing Function
Batch Bulk E-mail Item Update Notifications Sender Function
SES
REST API
API Gateway POST, POST, PUT, DEL (HTTP API Interface)
Lambda Queue Insert Function
SQS Queue
Lambda Process Queue Function
API Gateway GET (HTTP API Interface)
Lambda Get Data Function
Storage
Database
S3
Database Stream
Database Stream
Websocket API
Lambda MQTT Publishing Function
IoT Core

Authentication

For authentication, we have decided to use Amazon Cognito. The purpose of "Cognito" is to authenticate users and generate tokens for them (Cognito's User Pool), which can be exchanged by AWS credentials (Cognito's Identity Pool) that can be used to access other AWS services.

An option to "Cognito" would be "Auth0", which is often used on many sample code from Amazon's serverless developer blog.

"Auth0" is easier to use and better documented, but is more suited for smaller organizations.

"Auth0" is also more expensive: It offers monthly fixed price plans (price is based on different users caps), while "Cognito" charges on a per-user basis (with discounts as the number of users gets larger).

If we are expecting our user base to grow massively, "Cognito" is the best option, even though the learning curve is somewhat steeper than "Auth0".

REST API

For operations such as storing users, retailers (stores), and items (products) information, we require a basic REST API that supports CRUD operations.

In the following sections, we will detail all of these resources, as well as the data schema associated with them using Swagger, but in this section we will only describe (and justify the design) of the basic patterns we will be using.

Generally speaking, we are employing the usual serverless combination of "API Gateway" + "Lambda Function" to implement this service, with a slight modification to guarantee scalability for write operations.

An important question when designing back-end services is *"How many concurrent users can my system handle?".*

We know that AWS Serverless will automatically scale very quickly to respond to demand, but how about the database? The database usually will become the bottleneck, running out of connections as the number of concurrent users grows too high.

Note that we have not defined an specific database in the diagram. Basically, we have two choices:

- **Relational Database (such Aurora Serverless).**
- **Key-value Database (such as DynamoDB).**

We will discussings the pro's and con's regarding these choices in the following sections, but regardless of the choice, we need to guarantee that the system can handle the demand if the database becomes the bottleneck.

Our solution to this problem is using an intermediary buffer for incoming requests using SQS. We will store our requests in a durable store (SQS) and control the batch size of the requests that are processed (in the "Process Queue" Lambda function) so that we can process them at a speed that our system can handle.

Another thing worth mentioning is the use of Lambda Layers. I have not specified this in the diagram, but, of course, it makes sense to store common code shared among all Lambda functions (such as logging, monitoring, etc) in a Lambda layer.

Regarding "API Gateway", there are two options for interface: REST API vs HTTP API.

Considering the required features, HTTP API's will do the job and its cost is about one-third of REST API's (they are both RESTful by the way, regardless of how AWS calls them).

Websocket API

We are using IoT Core topics to send instant notifications to the app. Particularly for items (products), we are going to create a topic per item in the database. When an item is updated in the database, a database stream will trigger a lambda function that will send a message to a topic named after the item's unique ID. On the app side, there will be a MQTT client listening to these topics.

The idea is distributing checks for item updates throughout the day by letting users do this when possible.

If a user adds an item to one of his lists, we will check if the item already exists in the database, and if there were any changes, the item will be updated in the database.

The database update will trigger a database stream, which will then trigger a Lambda function that will publish a notification to the pertinent topic. Every user that has subscribed to this topic (when added the product to one of their lists) will receive the notification.

Scheduled Batch System

In the previous section, we discussed item (product) notifications. Every time a user adds an item to one of ones lists, we check if there were any item changes (price, in stock, low stock, etc) and update the item in the database, which triggers a notification through database streams.

Could be the case that an existing item is not added to any list for a period of time though, so notifications would not be created even if there were any item changes during this period.

To fix this problem we need to schedule a daily task to check for any changes in items. To schedule tasks, we will be using CloudWatch, which can be used to trigger Lambda functions.

Lambda functions have a limit for execution of 15 minutes and for this reason they are not well suited for the massive batch processing that might be required for this task. No worries though, AWS Batch comes to our rescue.

"AWS Batch" cannot be integrated directly with "CloudWatch" though, and, for this reason, we are using a Lambda function as a mediator between "CloudWatch" and "AWS Batch".

"AWS Batch" allows us to define all sorts of parameters to guarantee that we can perform the task we are trying to accomplish in an efficient and timely manner, such as number of priority queues, number of processing cores, etc.

We will use the same pattern to implement any other required periodic scheduled tasks, such as "weekly summary" and "account activity".

For item notifications (the scheduled task), we only need to update items that have not been updated in the last date (a fraction of the total items). Thus, we need to store the last time the item was updated in the database.

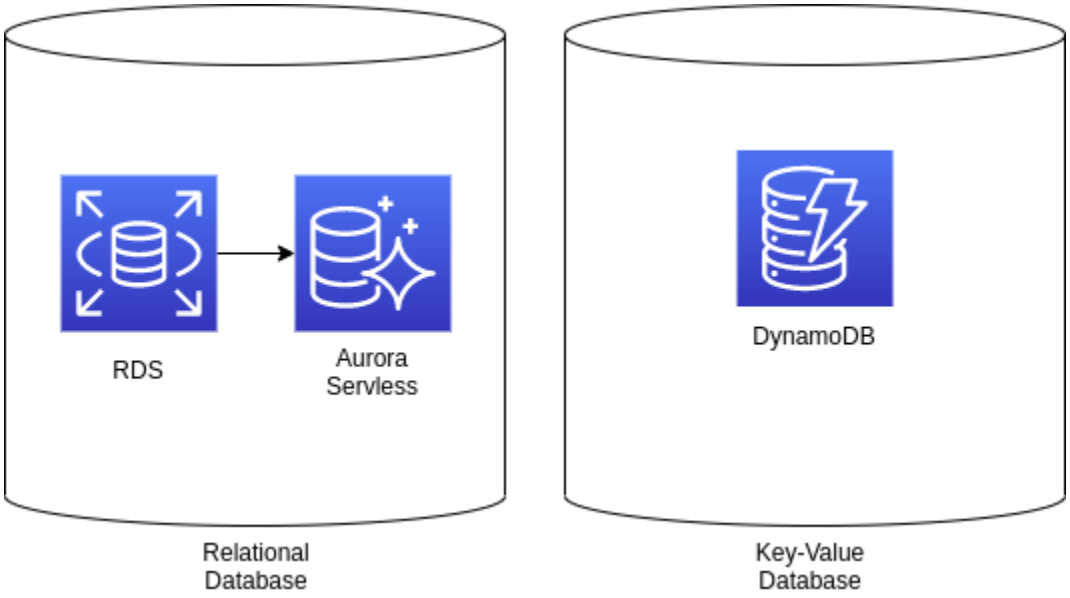
E-mail System

For email notifications, we will be using SES (Simple Email Service). Note that email sending can be triggered by the database stream (item changes notification) or as a scheduled task ("AWS Batch" job).

One downside of "SES" is that it doesn't support email lists, but it offers bulk sending using the "SES" API. It's worth mentioning that it's only possible to send batches of 50 messages at the time (asynchronously).

Storage

Database



There are two database choices:

- **Relational Database ("Aurora" being the most common option).**
- **Key-Value Database ("DynamoDB" being the most common option).**

In terms of performance and pricing, DynamoDB is the best option. Relational databases are, in general, priced hourly by instance size, while "DynamoDB is priced on use.

There's also the matter of cold-starts up to 10 seconds for Lambda functions, due to relational databases being network-partitioned in a private subnet of our VPC.

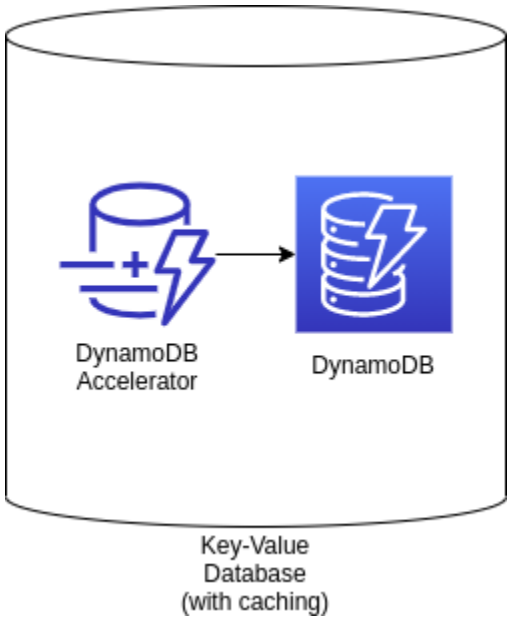
Relational databases also have connection limits, so Lambda functions may run out of connection when trying to connect.

Fortunately, Amazon has addressed all of these issues:

- **Aurora Serverless can actually be scaled up and down according to demand and its pricing model is also based on use.**
- **Amazon has improved VPC networking for Lambda functions, which reduced cold-start latency.**
- **Amazon has created the RDS Proxy service, which offloads connection management from Lambda functions, making possible for us to manage a much larger number of connections.**

That's all good news, because working with relational databases is much more familiar to developers and makes development much faster. Still, a relational database will never reach the same levels of performance of a key-value database.

In short, "DynamoDB" gives the potential for maximum performance, but at the cost of making development less flexible and slower. Relational databases are in general easier to understand, maintain, and perform migrations. Key-value databases, on the other hand, become a greater challenge if changes to the models are necessary, often requiring massive migrations.



We also have the option of adding DAX (DynamoDB Accelerator) to "DynamoDB", which would make the database performance even better (at the expense of additional costs, of course).

I believe that we should be able to use the best of both worlds though.

In a new project, where our models are not yet mature, the best solution is using a relational database, which will allow maximum agility of development. By the time performance becomes a problem, the database models should become pretty well defined. At this point we should refactor our back-end to use a key-value database instead.

S3

We are going to use S3 to store resources such as thumbnails for items and logos for retailers in S3. Images can be retrieved from S3 using AWS's Javascript SDK.

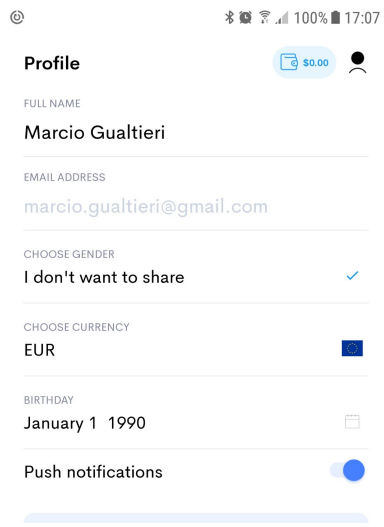
Data Models

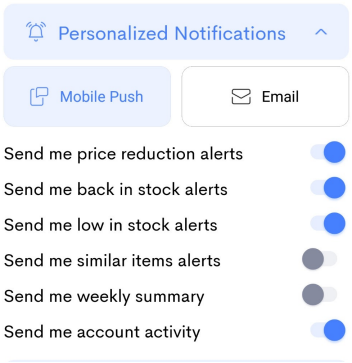
Supported Functionality

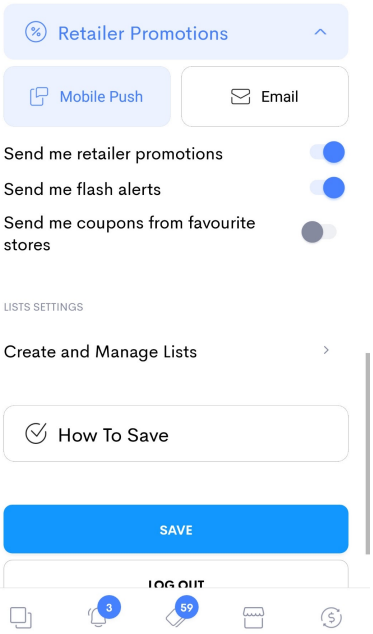
This section relates more specifically to the REST API. We need to define all functionality that is required by the app, so we can define the necessary resources.

User Management

We need the proper REST API resources to manage user information, even though authentication is being handled by "Cognito". The following screen cap (broken into three pieces to fit the page) shows the information we are required to store in the database.







The following notifications seems to not be supported by the back-end at the moment:

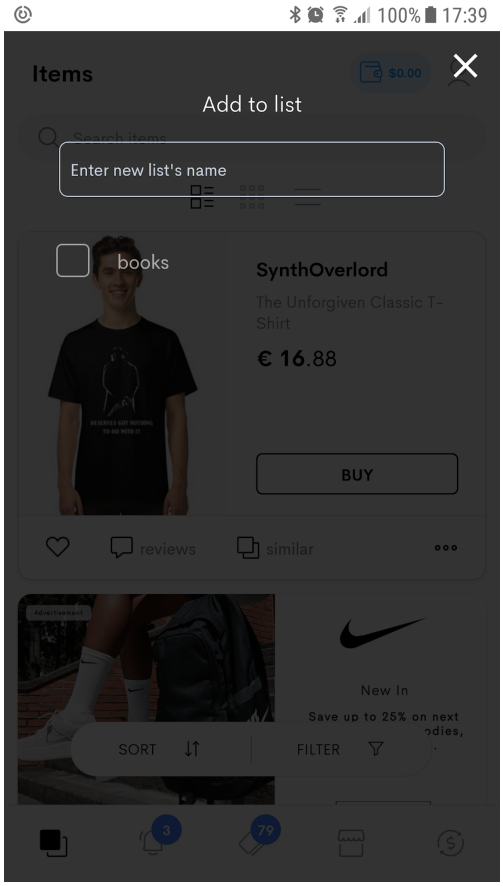
- **"Send me similar items alerts".**
- **"Send me weekly summary".**
- **"Send me coupons from my favourite stores".**

Also, I feel like "Create and Manage Lists" should have its own tab next to the "Items" tab instead of being part of the user profile.

I have not received a "Weekly Summary" and "User Activity" during the time I tried the app, thus I have no idea about the content of this, for this reason I have not defined data structures for these two.

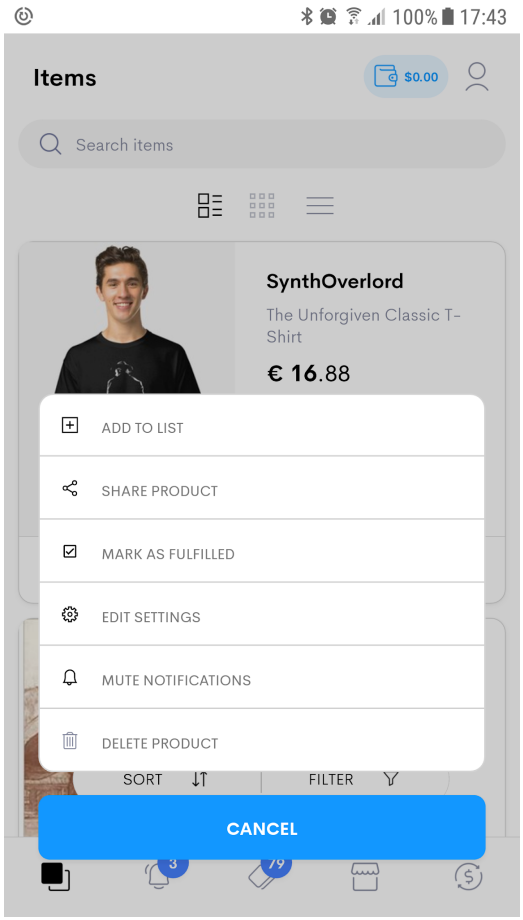
List Management

We require basic CRUD operations here, to store, update, retrieve and delete tags in the database.



Item Management

We should be able to perform CRUD operations on items, as well as adding and removing items to/from lists.



Notifications Management

The "Notifications" tab only shows the notifications that the app has received so far. It doesn't require the back-end to delete notifications though as they are stored in the app.

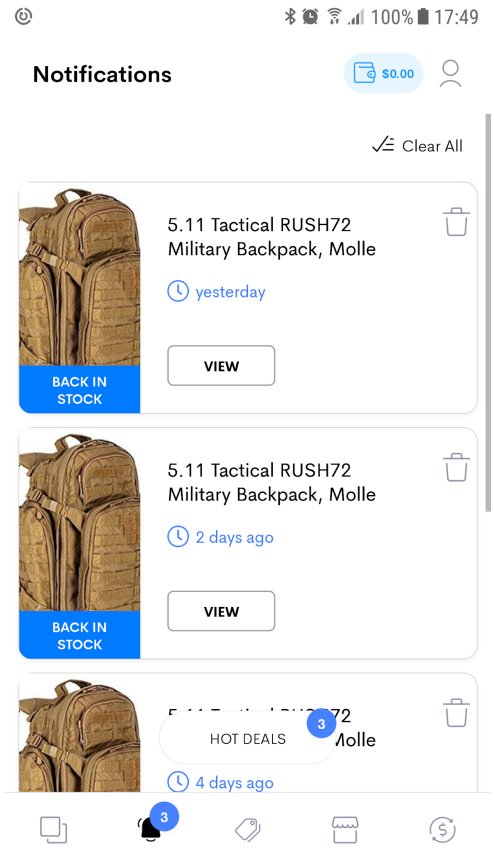
There are two kinds of notification:

- **Item Notification: When an item belonging to a user's list is updated (price, in stock, low stock, etc).**
- **Periodic Notification: Such as weekly summary and user's activity.**

We have decided to make a distinction between the two instead of trying to find a general abstraction that would address both. I believe these two notifications are different enough from one another to justify two entities.

Item notifications will be added to an user and item, based on the user profile choices.

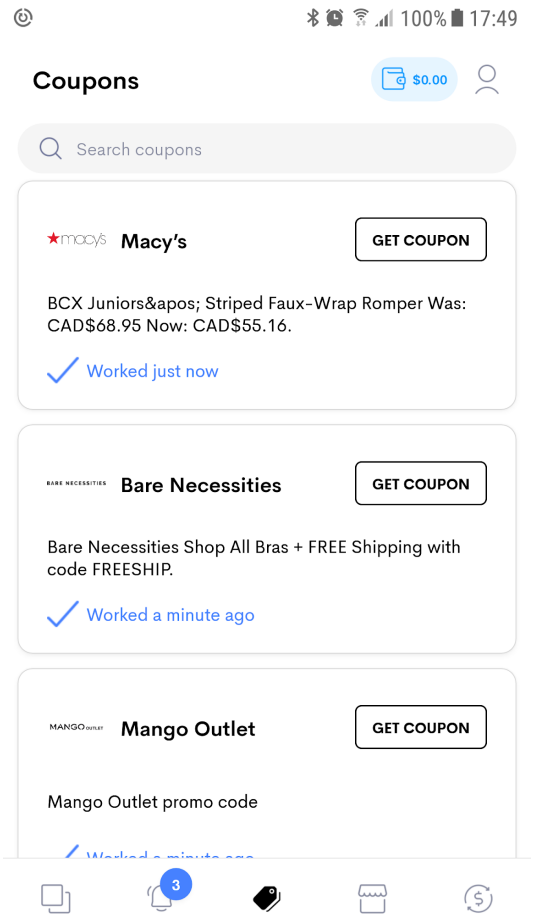
We also should be able to enable or disable item notifications (for changes in price, in stock, low stock, etc) for specific users and items.



Coupons Management

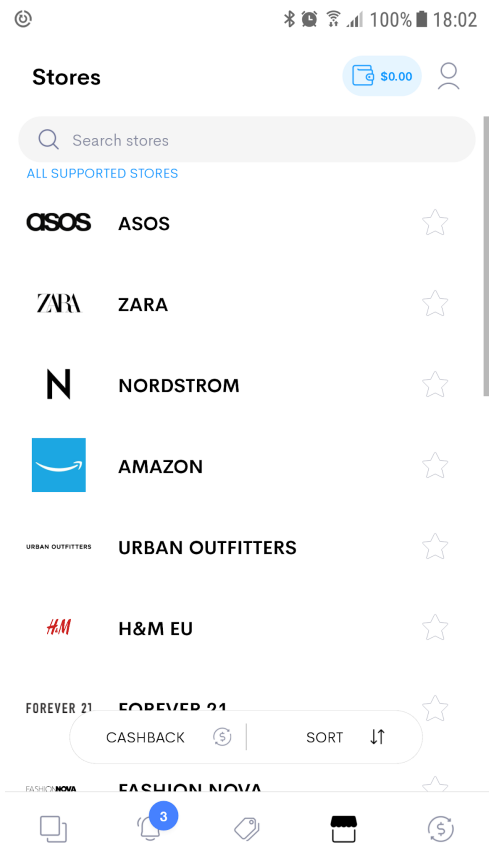
The "Coupons" tab only lists the available coupons, which are curated by staff. We imagine that there's a front-end somewhere in their private network that is used to manage coupons. Thus, we still need to support CRUD operations for the staff.

The buttons ("Get Coupon") only opens a web link for the coupon.



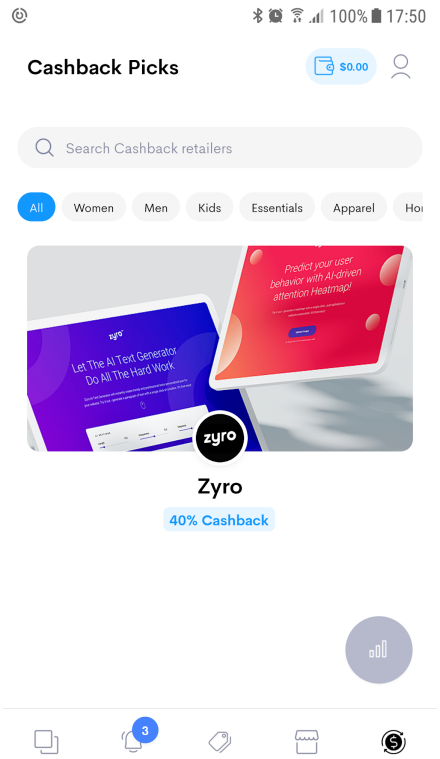
Retailer Management

Similar to the "Coupons" tab, the "Stores" tab only shows the supported stores, which are also curated by staff. CRUD operations are required.



We also should be able to add and remove tags to retailers. In the "Stores" tab, tags are not shown, but you will notice that the "Cashback" tab lists specific retailers, which have been hand-picked by staff due to offering cashback offers. In that particular screen, the retailers can be filtered by tags.

Cashback Picks Management

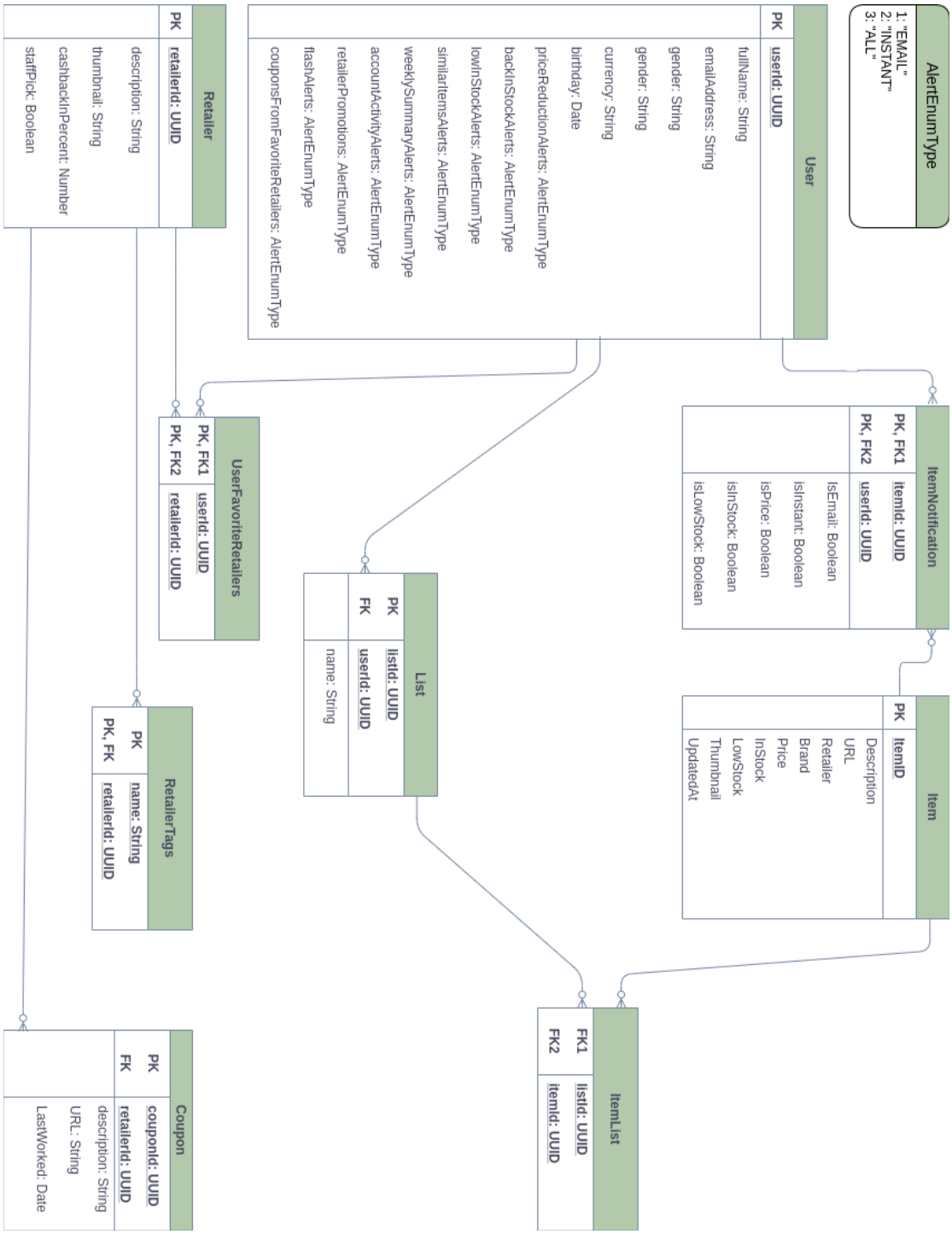


Cashback information comes from the retailers (stores), which are curated by staff. The "Cashback" tab only shows the cashback picks curated by staff. Cashback is a property of a retailer, thus the functionality in this screen can be fulfilled by the retailer's back-end resource. We don't need specific resources for cashback.

Notice that there's a button (bottom right), which seems to be used for sorting. Unfortunately, there's only a single cashback pick at the moment, thus, we have no idea what's the sorting criteria. But we are going to defined the REST API resource in such a way that we can provide query parameters to sort by any attribute.

Database Models

Relational Models



Key-Value Models

User Table

Primary Key		Attributes									
Partition Key	Sorty Key	UserID	FullName	EmailAddress	Gender	Currency	Birthday	PriceReductionAlerts	BackInStockAlerts
user#<UserID>	profile	LowInStockAlerts	SimilarItemsAlerts	WeeklySummaryAlerts	AccountActivityAlerts	RetailerPromotions	FlashAlerts	CouponsFromFavoriteRetailers	
	item#<ListID>#<ItemID>	UserID	ListID	List::Description	ItemID				
	favorite-retailer#<RetailerID>	UserID	RetailerID						
	item-notification#<NotificationID>	UserID	NotificationID	Notification::IsEmail	Notification::IsInstant	Notification::IsPrice	Notification::IsInStock	Notification::IsLowInStock	

Item Table

Primary Key		Attributes						
Partition Key	Sorty Key	URL	Retailer	Brand	Price	InStock	LowStock	Thumbnail
item#<ItemID>	updated#<UpdatedAt>							

Retailer Table

Primary Key		Attributes				
Partition Key	Sorty Key	RetailerID	Description	Thumbnail	Cashback	StaffPick
retailer#<RetailerID>	retailer	RetailerID	Description	Thumbnail	Cashback	StaffPick
	tag#<Tag::Name>	RetailerID	Tag::Name			
	coupon#<CouponID>	RetailerID	CouponID	Description	URL	LastWorked

Coupon Table

Primary Key		Attributes				
Partition Key	Sorty Key	RetailerID	CouponID	Description	URL	LastWorked
coupon#<CouponID>	coupon#<RetailerID>#<CouponID>					

Note that I have tried to avoid using Global secondary index for performance sakes. I'm also trying to avoid joins when possible.

Data Schema (Swagger)

In the next pages follows the data schema created using Swagger. It consists of a list of all resources with their corresponding data schema (body and response). I only created resources for features that have been described in the "Supported Functionality" section.

Shoptagr Back-end

This is the data schema for the Shoptagr's back-end service (reversed engineered, not the actual). It's been generated with the Swagger Editor.
More information: https://helloreverb.com
Contact Info: marcio.gualtieri@gmail.com
Version: 1.0.0
BasePath:/v1
Apache 2.0
http://www.apache.org/licenses/LICENSE-2.0.html

Access

Methods

[Jump to Models]

Table of Contents

- DELETE /users/{userId}/lists/{listId}/items/{itemId}
- GET /users/{userId}/items
- GET /users/{userId}/lists/{listId}/items

UserManagementLists

- POST /users/{userId}/lists/{name}
- DELETE /users/{userId}/lists/{name}
- GET /users/{userId}/lists

CouponManagement

POST /coupons

Up

Create a new coupon (**createCoupon**)

Consumes
This API call consumes the following media types via the Content-Type request header:

- application/json

Request body

body CouponBody (required)
Body Parameter — Coupon object that needs to be created

Return type
array[CouponResponse]

Example data
Content-Type: application/json

```
[ {
  "description" : "description",
  "couponId" : "couponId",
  "retailerId" : "retailerId",
  "lastWorked" : "2000-01-23T04:56:07.000+00:00",
  "URL" : 0
}, {
  "description" : "description",
  "couponId" : "couponId",
  "retailerId" : "retailerId",
  "lastWorked" : "2000-01-23T04:56:07.000+00:00",
  "URL" : 0
} ]
```

Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- application/json

Responses
201
successful operation
400
Invalid input

Up

DELETE /coupons/{couponId}

Delete coupon (**deleteCoupon**)

Path parameters

couponId (required)
Path Parameter — ID of the coupon

Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- `application/json`

Responses
200
successful operation
404
Coupon not found

Up

GET /coupons

Get all coupons (**getAllCoupons**)

Query parameters

offset (optional)
Query Parameter — The number of items to skip before starting to collect the result set

limit (optional)
Query Parameter — The numbers of items to return

order_by (optional)
Query Parameter — Order by expression, i.e. "<attribute name> [asc|desc]". "asc" is default.

Return type
array[CouponResponse]

Example data
Content-Type: application/json

```
[ {
  "description" : "description",
  "couponId" : "couponId",
  "retailerId" : "retailerId",
  "lastWorked" : "2000-01-23T04:56:07.000+00:00",
  "URL" : 0
}, {
  "description" : "description",
  "couponId" : "couponId",
  "retailerId" : "retailerId",
  "lastWorked" : "2000-01-23T04:56:07.000+00:00",
  "URL" : 0
} ]
```

Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- application/json

Responses
200
successful operation

GET /coupons/{couponId}

Up

Get coupon by ID (**getCouponById**)

Returns a single coupon

Path parameters

couponId (required)
Path Parameter — ID of the coupon

Return type
CouponResponse

Example data
Content-Type: application/json

```
{
  "description" : "description",
  "couponId" : "couponId",
  "retailerId" : "retailerId",
  "lastWorked" : "2000-01-23T04:56:07.000+00:00",
  "URL" : 0
}
```

Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- application/json

Responses
200
successful operation CouponResponse
404
Retailer not found

PUT /coupons/{couponId}

Up

Update a coupon (**updateCoupon**)

Path parameters

couponId (required)
Path Parameter — ID of the coupon

Consumes
This API call consumes the following media types via the Content-Type request header:

- application/json

Request body

body CouponBody (required)

Body Parameter — Updated coupon object

Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- `application/json`

Responses
200
successful operation
400
Invalid coupon ID supplied
404
Coupon not found

NotificationManagementItems

`GET /users/{userId}/items/{itemId}/notifications` Up

Get user item notifications (**getUserItemNotifications**)

Path parameters

userId (required)
Path Parameter — ID of the user

itemId (required)
Path Parameter — ID of the item

Return type
NotificationBody

Example data
Content-Type: application/json

```
{
  "isPrice" : true,
  "isEmail" : true,
  "isInstant" : true,
  "isInStock" : true,
  "isLowStock" : true
}
```

Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- `application/json`

Responses
200
successful operation NotificationBody
404
User not found

`PUT /users/{userId}/items/{itemId}/notifications` Up

Update user item notifications (**updateUserItemNotifications**)

Path parameters

userId (required)
Path Parameter — ID of the user

itemId (required)
Path Parameter — ID of the item

Consumes
This API call consumes the following media types via the Content-Type request header:

- `application/json`

Request body

body NotificationBody (required)
Body Parameter — Updated notification object

Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- `application/json`

Responses
201
successful operation
400
Invalid input

RetailerManagement

Up

`POST /retailers`

Create a new retailer (**createRetailer**)

Consumes
This API call consumes the following media types via the Content-Type request header:

- `application/json`

Request body

body RetailerBody (required)
Body Parameter — Retailer object that needs to be created

Return type
RetailerResponse

Example data
Content-Type: application/json

```
{
  "thumbnail" : "thumbnail",
  "staffPick" : true,
  "description" : "description",
  "retailerId" : "retailerId",
```

```
  "cashback" : 0
}
```

Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- `application/json`

Responses
201
successful operation [RetailerResponse](#)
400
Invalid input

`DELETE /retailer/{retailerId}` Up

Delete retailer (**deleteRetailer**)

Path parameters

retailerId (required)
Path Parameter — ID of the retailer

Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- `application/json`

Responses
200
successful operation
404
Retailer not found

`GET /retailers` Up

Get all retailers (**getAllRetailers**)

Query parameters

offset (optional)
Query Parameter — The number of items to skip before starting to collect the result set

limit (optional)
Query Parameter — The numbers of items to return

order_by (optional)
Query Parameter — Order by expression, i.e. "<attribute name> [asc|desc]". "asc" is default.

cashback_pick (optional)
Query Parameter — Returns only the retailers with cashback that have been picked by staff.

Return type
array[RetailerResponse]

Example data
Content-Type: application/json

```
[ {
  "thumbnail" : "thumbnail",
  "staffPick" : true,
  "description" : "description",
  "retailerId" : "retailerId",
  "cashback" : 0
}, {
  "thumbnail" : "thumbnail",
  "staffPick" : true,
  "description" : "description",
  "retailerId" : "retailerId",
  "cashback" : 0
} ]
```

Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- `application/json`

Responses
200
successful operation

`GET /retailer/{retailerId}`

Up

Get retailer by ID (**getRetailerById**)

Returns a single retailer

Path parameters

retailerId (required)
Path Parameter — ID of the retailer

Return type
RetailerResponse

Example data
Content-Type: application/json

```
{
  "thumbnail" : "thumbnail",
  "staffPick" : true,
  "description" : "description",
  "retailerId" : "retailerId",
  "cashback" : 0
}
```

Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- `application/json`

Responses
200
successful operation RetailerResponse
404
Retailer not found

PUT /retailer/{retailerId}

Up

Update a retailer (**updateRetailer**)

Path parameters

retailerId (required)
Path Parameter — ID of the retailer

Consumes
This API call consumes the following media types via the Content-Type request header:

- `application/json`

Request body

body RetailerBody (required)
Body Parameter — Updated retailer object

Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- `application/json`

Responses
200
successful operation
400
Invalid input
404
Retailer not found

RetailerManagementTags

POST /retailers/{retailerId}/tags/{tag}

Up

Add single tag to retailer (**addTagToRetailer**)

Path parameters

retailerId (required)
Path Parameter — ID of the retailer

tag (required)
Path Parameter — Tag to add to retailer

Consumes
This API call consumes the following media types via the Content-Type request header:

- `application/json`

Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- `application/json`

Responses
201

successful operation
400
Invalid input

POST /retailers/{retailerId}/tags

Up

Add tags to retailer (**addTagsToRetailer**)

Path parameters

retailerId (required)
Path Parameter — ID of retailer

Consumes
This API call consumes the following media types via the Content-Type request header:

- `application/json`

Request body

body string (required)
Body Parameter — Updated retailer tags

Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- `application/json`

Responses
201
successful operation
400
Invalid input

GET /retailers/{retailerId}/tags

Up

Get retailer tags (**getRetailerTags**)

Tags

Path parameters

retailerId (required)
Path Parameter — ID of retailer

Return type
array[String]

Example data
Content-Type: application/json

```
[ "", "" ]
```

Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- `application/json`

Responses
200
successful operation

DELETE /retailers/{retailerId}/tags/{tag}

Up

Remove retailer tag (**removeRetailerTag**)

Path parameters

retailerId (required)
Path Parameter — ID of the retailer

tag (required)
Path Parameter — Tag to add to retailer

Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- `application/json`

Responses
200
successful operation
404
Retailer not found

UserManagement

POST /users

Up

Create a new user (**createUser**)

Consumes
This API call consumes the following media types via the Content-Type request header:

- `application/json`

Request body

body UserBody (required)
Body Parameter — User object that needs to be created

Return type
UserResponse

Example data
Content-Type: application/json

```
{
  "birthday" : "birthday",
  "backInStockAlerts" : "backInStockAlerts",
  "flashAlerts" : "flashAlerts",
  "gender" : "gender",
  "retailPromotions" : "retailPromotions",
  "fullName" : "fullName",
  "userId" : "userId",
```

```
  "priceReductionAlerts" : "priceReductionAlerts",
  "emailAddress" : "emailAddress",
  "similarItemsAlerts" : "similarItemsAlerts",
  "accountActivityAlerts" : "accountActivityAlerts",
  "lowInStockAlerts" : "lowInStockAlerts",
  "currency" : "currency",
  "weeklySummaryAlerts" : "weeklySummaryAlerts",
  "couponsFromFavoriteRetailers" : "couponsFromFavoriteRetailers"
}
```

Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- `application/json`

Responses
201
successful operation [UserResponse](#)
400
Invalid input

`DELETE /users/{userId}`

Up

Delete user (**deleteUserById**)

Path parameters

userId (required)
Path Parameter — ID of the user

Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- `application/json`

Responses
200
successful operation
404
User not found

`GET /users`

Up

Get all users (**getAllUsers**)

Query parameters

offset (optional)
Query Parameter — The number of items to skip before starting to collect the result set

limit (optional)
Query Parameter — The numbers of items to return

order_by (optional)
Query Parameter — Order by expression, i.e. "<attribute name> [asc|desc]". "asc" is default.

Return type
array[UserResponse]

Example data
Content-Type: application/json


```
[ {
  "birthday" : "birthday",
  "backInStockAlerts" : "backInStockAlerts",
  "flashAlerts" : "flashAlerts",
  "gender" : "gender",
  "retailPromotions" : "retailPromotions",
  "fullName" : "fullName",
  "userId" : "userId",
  "priceReductionAlerts" : "priceReductionAlerts",
  "emailAddress" : "emailAddress",
  "similarItemsAlerts" : "similarItemsAlerts",
  "accountActivityAlerts" : "accountActivityAlerts",
  "lowInStockAlerts" : "lowInStockAlerts",
  "currency" : "currency",
  "weeklySummaryAlerts" : "weeklySummaryAlerts",
  "couponsFromFavoriteRetailers" : "couponsFromFavoriteRetailers"
}, {
  "birthday" : "birthday",
  "backInStockAlerts" : "backInStockAlerts",
  "flashAlerts" : "flashAlerts",
  "gender" : "gender",
  "retailPromotions" : "retailPromotions",
  "fullName" : "fullName",
  "userId" : "userId",
  "priceReductionAlerts" : "priceReductionAlerts",
  "emailAddress" : "emailAddress",
  "similarItemsAlerts" : "similarItemsAlerts",
  "accountActivityAlerts" : "accountActivityAlerts",
  "lowInStockAlerts" : "lowInStockAlerts",
  "currency" : "currency",
  "weeklySummaryAlerts" : "weeklySummaryAlerts",
  "couponsFromFavoriteRetailers" : "couponsFromFavoriteRetailers"
} ]
```


Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- `application/json`

Responses
200
successful operation

`GET /users/{userId}`

Up

Get user by ID (**getUserById**)

Returns a single user

Path parameters

userId (required)
Path Parameter — ID of the user

Return type

UserResponse

Example data
Content-Type: application/json

```
{
  "birthday" : "birthday",
  "backInStockAlerts" : "backInStockAlerts",
  "flashAlerts" : "flashAlerts",
  "gender" : "gender",
  "retailPromotions" : "retailPromotions",
  "fullName" : "fullName",
  "userId" : "userId",
  "priceReductionAlerts" : "priceReductionAlerts",
  "emailAddress" : "emailAddress",
  "similarItemsAlerts" : "similarItemsAlerts",
  "accountActivityAlerts" : "accountActivityAlerts",
  "lowInStockAlerts" : "lowInStockAlerts",
  "currency" : "currency",
  "weeklySummaryAlerts" : "weeklySummaryAlerts",
  "couponsFromFavoriteRetailers" : "couponsFromFavoriteRetailers"
}
```

Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- `application/json`

Responses
200
successful operation UserResponse
404
User not found

PUT /users/{userId}

Up

Update a user (**updateUserById**)

Path parameters

userId (required)
Path Parameter — ID of the user

Consumes
This API call consumes the following media types via the Content-Type request header:

- `application/json`

Request body

body UserBody (required)
Body Parameter — Updated user object

Responses
200
successful operation
400
Invalid user ID supplied
404

User not found

UserManagementItems

POST /users/{userId}/lists/{listId}/items

Up

Add single item to user list (**addItemToUserList**)

Path parameters

userId (required)
Path Parameter — ID of the user

listId (required)
Path Parameter — ID of the list

Consumes
This API call consumes the following media types via the Content-Type request header:

- `application/json`

Request body

body ItemBody (required)
Body Parameter — Updated item object

Return type
ItemResponse

Example data
Content-Type: application/json

```
{
  "itemId" : "itemId",
  "thumbnail" : "thumbnail",
  "retailer" : "retailer",
  "price" : 0.8008281904610115020659577567130327224731445312 5,
  "lists" : [ "lists", "lists" ],
  "description" : "description",
  "lowStock" : true,
  "inStock" : true,
  "brand" : "brand",
  "URL" : "URL",
  "notifications" : {
    "isPrice" : true,
    "isEmail" : true,
    "isInstant" : true,
    "isInStock" : true,
    "isLowStock" : true
  },
  "updatedAt" : "2000-01-23T04:56:07.000+00:00"
}
```

Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- `application/json`

Responses

201
successful operation ItemResponse
400
Invalid input

DELETE /users/{userId}/lists/{listId}/items/{itemId}

Up

Delete user list (**deleteUserListItem**)

Path parameters

userId (required)
Path Parameter — ID of the user

listId (required)
Path Parameter — ID of the list

itemId (required)
Path Parameter — ID of the item

Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- `application/json`

Responses
200
successful operation
404
User or List not found

GET /users/{userId}/items

Up

Get user items (**getUserItems**)

Path parameters

userId (required)
Path Parameter — ID of user

Query parameters

offset (optional)
Query Parameter — The number of items to skip before starting to collect the result set

limit (optional)
Query Parameter — The numbers of items to return

order_by (optional)
Query Parameter — Order by expression, i.e. "<attribute name> [asc|desc]". "asc" is default.

Return type
array[ItemResponse]

Example data
Content-Type: application/json

```
[ {
  "itemId" : "itemId",
```

```
  "thumbnail" : "thumbnail",
  "retailer" : "retailer",
  "price" : 0.8008281904610115020659577567130327224731445312 5,
  "lists" : [ "lists", "lists" ],
  "description" : "description",
  "lowStock" : true,
  "inStock" : true,
  "brand" : "brand",
  "URL" : "URL",
  "notifications" : {
    "isPrice" : true,
    "isEmail" : true,
    "isInstant" : true,
    "isInStock" : true,
    "isLowStock" : true
  },
  "updatedAt" : "2000-01-23T04:56:07.000+00:00"
}, {
  "itemId" : "itemId",
  "thumbnail" : "thumbnail",
  "retailer" : "retailer",
  "price" : 0.8008281904610115020659577567130327224731445312 5,
  "lists" : [ "lists", "lists" ],
  "description" : "description",
  "lowStock" : true,
  "inStock" : true,
  "brand" : "brand",
  "URL" : "URL",
  "notifications" : {
    "isPrice" : true,
    "isEmail" : true,
    "isInstant" : true,
    "isInStock" : true,
    "isLowStock" : true
  },
  "updatedAt" : "2000-01-23T04:56:07.000+00:00"
} ]
```


Produces

This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- `application/json`

Responses

200
successful operation
404
User not found

`GET /users/{userId}/lists/{listId}/items`

Up

Get user list items (**getUserListItems**)

Path parameters

userId (required)
Path Parameter — ID of the user

listId (required)
Path Parameter — ID of the list

Query parameters

offset (optional)
Query Parameter — The number of items to skip before starting to collect the result set

limit (optional)
Query Parameter — The numbers of items to return

order_by (optional)
Query Parameter — Order by expression, i.e. "<attribute name> [asc|desc]". "asc" is default.

Return type
array[ItemResponse]

Example data
Content-Type: application/json


```
[ {
  "itemId" : "itemId",
  "thumbnail" : "thumbnail",
  "retailer" : "retailer",
  "price" : 0.8008281904610115020659577567130327224731445312 5,
  "lists" : [ "lists", "lists" ],
  "description" : "description",
  "lowStock" : true,
  "inStock" : true,
  "brand" : "brand",
  "URL" : "URL",
  "notifications" : {
    "isPrice" : true,
    "isEmail" : true,
    "isInstant" : true,
    "isInStock" : true,
    "isLowStock" : true
  },
  "updatedAt" : "2000-01-23T04:56:07.000+00:00"
}, {
  "itemId" : "itemId",
  "thumbnail" : "thumbnail",
  "retailer" : "retailer",
  "price" : 0.8008281904610115020659577567130327224731445312 5,
  "lists" : [ "lists", "lists" ],
  "description" : "description",
  "lowStock" : true,
  "inStock" : true,
  "brand" : "brand",
  "URL" : "URL",
  "notifications" : {
    "isPrice" : true,
    "isEmail" : true,
    "isInstant" : true,
    "isInStock" : true,
    "isLowStock" : true
  },
  "updatedAt" : "2000-01-23T04:56:07.000+00:00"
} ]
```


Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- `application/json`

Responses
200
successful operation
404
User not found

UserManagementLists

`POST /users/{userId}/lists/{name}`

Up

Add single list to a user (**addListToUser**)

Path parameters

userId (required)
Path Parameter — ID of the user

name (required)
Path Parameter — Name of the list to add

Consumes
This API call consumes the following media types via the Content-Type request header:

- `application/json`

Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- `application/json`

Responses
201
successful operation
400
Invalid input

`DELETE /users/{userId}/lists/{name}`

Up

Delete user list (**deleteUserList**)

Path parameters

userId (required)
Path Parameter — ID of the user

name (required)
Path Parameter — Name of the list to add

Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- `application/json`

Responses
200
successful operation

404
User or List not found

GET /users/{userId}/lists

Up

Get user lists (**getUserLists**)

Path parameters

userId (required)
Path Parameter — ID of user

Return type
array[ListResponse]

Example data
Content-Type: application/json

```
[ {
  "listId" : "listId",
  "name" : "name"
}, {
  "listId" : "listId",
  "name" : "name"
} ]
```

Produces
This API call produces the following media types according to the Accept request header; the media type will be conveyed by the Content-Type response header.

- `application/json`

Responses
200
successful operation
404
User not found

Models

[Jump to Methods]

Table of Contents

CouponBody -

Up

retailerId
String

description
String

URL
String

lastWorked
Date format: date-time

`CouponResponse` - Up

couponId (optional)
String

retailerId (optional)
String

description (optional)
String

URL (optional)
Integer

lastWorked (optional)
Date format: date-time

`ItemBody` - Up

description
String

URL
String

retailer
String

brand
String

price
BigDecimal

inStock
Boolean

lowStock
Boolean

thumbnail
String

`ItemResponse` - Up

itemId (optional)
String

description (optional)
String

URL (optional)
String

retailer (optional)
String

brand (optional)
String

price (optional)
BigDecimal

inStock (optional)
Boolean

lowStock (optional)
Boolean

thumbnail (optional)
String

updatedAt (optional)
Date format: date-time

notifications (optional)
NotificationBody

lists (optional)
array[String]

`ListResponse` - Up

listId (optional)
String

name (optional)
String

`NotificationBody` - Up

isEmail
Boolean

isInstant
Boolean

isPrice
Boolean

isInStock
Boolean

isLowStock
Boolean

`RetailerBody` - Up

description
String

thumbnail
String

cashback (optional)
Integer

staffPick (optional)
Boolean

`RetailerResponse` - Up

retailerId (optional)
String

description (optional)
String

thumbnail (optional)
String

cashback (optional)
Integer

staffPick (optional)
Boolean

UserBody - [Up]

fullName
String

emailAddress
String

gender (optional)
String

currency (optional)
String

birthday (optional)
Date format: date-time

priceReductionAlerts (optional)
String

backInStockAlerts (optional)
String

lowInStockAlerts (optional)
String

similarItemsAlerts (optional)
String

weeklySummaryAlerts (optional)
String

accountActivityAlerts (optional)
String

retailPromotions (optional)
String

flashAlerts (optional)
String

couponsFromFavoriteRetailers (optional)
String

UserResponse - [Up]

userId (optional)
String

fullName (optional)
String

emailAddress (optional)

String

gender (optional)
String

currency (optional)
String

birthday (optional)
String

priceReductionAlerts (optional)
String

backInStockAlerts (optional)
String

lowInStockAlerts (optional)
String

similarItemsAlerts (optional)
String

weeklySummaryAlerts (optional)
String

accountActivityAlerts (optional)
String

retailPromotions (optional)
String

flashAlerts (optional)
String

couponsFromFavoriteRetailers (optional)
String