Routable

Recruiting take home challenge – Backend 2020

Take Home Challenge:

For this exercise, you will need to create a simple Django application that mimics some of our transaction flows. This application should have 2 versions, tagged in git for easy checkout, an initial version that has some limitations and a final version that will correct some of the limitations that we introduced in the initial version.

This application will have a handful of simple models, endpoints and Django admin pages. All the views will need to be API views returning JSON responses.

Packages that we use:

1. Django REST Framework
2. Django Object Actions

You are welcome to use *Django REST Framework* for making REST API requests and responses, or any other framework of your choice.

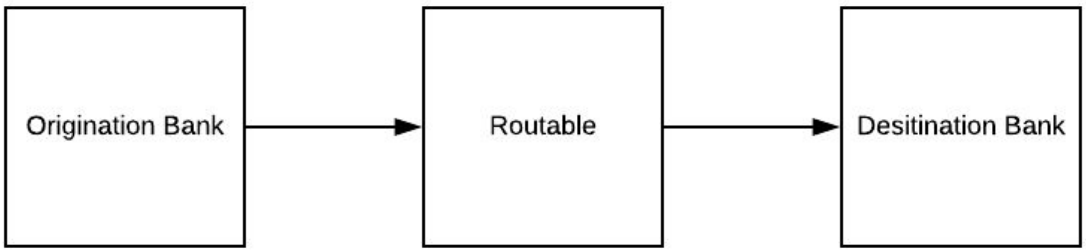
We use Django Object Actions for admin instance page actions. You're welcome to use that or any other package of your choice to allow admin page actions, which will be needed in v2.

Deliverables:

A link to a Github repository. The repository can be public. If you want it to be private, please invite: *tomharel*, *justmobilize*.

Things to note:

1. For this exercise, assume that our system's flow of funds is as follows:



2. Any identifiers should be in UUID format.

Version 1

This initial version of the application is intended to be an MVP that is intentionally missing an important flow. When an error on a Transaction occurs, the application will retry the full flow of funds again (see logic below) and will result in charging the Origination Bank multiple times. This will be fixed in v2.

Models:
1. *Item*

The Item model is what is created when you create a payment. Since our application supports both bill/payables or invoice/receivables, we opted to use a generic term. For this exercise you can assume that an *Item* is analogous to a payment.

Fields needed:

I. Field that represents the amount this item is for.

2. *Transaction*

The *Transaction* model tracks the actual flow of funds. There could be more than one *Transaction* per Item.

Fields needed:

I. A field that represents the current *status* of the *Transaction*. Possible values:
   - processing
   - completed
   - error

II. A field that represents the *location* of the funds transferred. Possible values:
   - origination_bank
   - routable
   - destination_bank

Endpoints:
1. Create *Item* - This endpoint is responsible for creating a new Item.

2. Create *Transaction* - This endpoint is responsible for creating a new *Transaction* on an Item (see logic section below).

3. Move *Item* - This endpoint is responsible for moving an *Item's* active *Transaction* between states (see logic section below).

4. Error *Item* - This endpoint is responsible for marking an *Item's* active *Transaction* status as an error (see logic section below).

Admin pages:
1. *Item* list
2. *Item* details with transactions inline
3. Transaction details

Logic:

Transaction states

Below is a list of the potential transaction states the application is supposed to support.
** indicates new transaction*

A*:

status	processing
location	originator_bank

B:

status	processing
location	routable

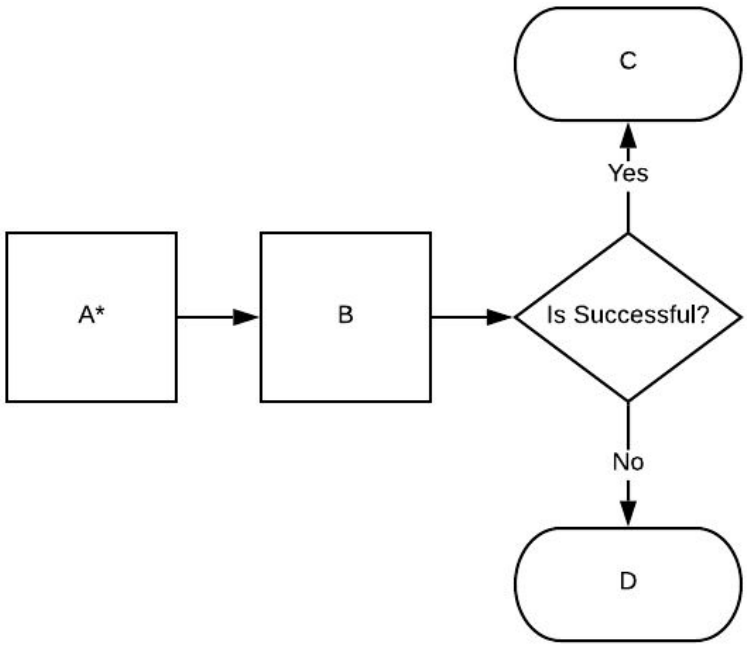
C:

status	completed
location	destination_bank

D:

status	error
location	routable

Transaction flow



Item and *Transaction* logic:
An Item can have multiple transactions, but at any point in time only one transaction is "active".

We have two flows a Transaction can take at this point. It's either successful (A-B-C) or it has an error reaching the Destination Bank (A-B-D).

In the case that we have a failure (A-B-D), we will need to create a new transaction to retry sending funds to the Destination Bank. Since we don't have logic in place to "fix" the first transaction created, it will go through the flow again (A-B-C).

We expect the application to work as follows:

Successful flow (A-B-C)
1. Create an item by using the "Create *Item*" endpoint.
2. Create a transaction by using the "Create *Transaction*" endpoint, starting in A (we now have an "active" transaction).
3. Move the "active" transaction to B, by using the "Move *Item*" endpoint.
4. Complete the "active" transaction moving it to C, by using the "Move *Item*" endpoint.

Error flow (A1-B1-D1, A2-B2-C2)
1. Create an item by using the "Create *Item*" endpoint.
2. Create a transaction by using the "Create *Transaction*" endpoint, starting in A (we now have an "active" transaction).
3. Move the "active" transaction to B, by using the "Move *Item*" endpoint.
4. Add an error on the "active" transaction, moving it to D, by using the "Error *Item*" endpoint.
5. Create a transaction by using the "Create *Transaction*" endpoint, starting in A (we now have an "active" transaction).
6. Move the "active" transaction to B, by using the "Move *Item*" endpoint (note: you should be able to error here again as in step 4 above).
7. Complete the "active" transaction moving it to C, by using the "Move *Item*" endpoint.

Version 2

This version of the application is intended to be an augmentation to the initial version, adding a missing flow, as well as allows us to fix the issues created in v1, i.e. will let us fix the instances in which we charged the Destination Bank multiple times.

Models:
1. *Item*

No changes from v1.

2. *Transaction*

Same fields as in v1, but with the following options for the status field:

- refunding
- refunded
- fixing

Endpoints:
All endpoints from v1, with the following additions:

5. Fix *Item* - This endpoint is responsible for fixing an item whose "active" transaction is in an error state.

Admin pages:
All pages from v1, with the following tweaks:

1. Transaction details should now include an action to refund a transaction that is stuck in the error state (D). See the logic section below for more information.

Logic:

Transaction states
All states from v1, with the addition of the following:
** indicates new transaction*

E*:

status	refunding
location	routable

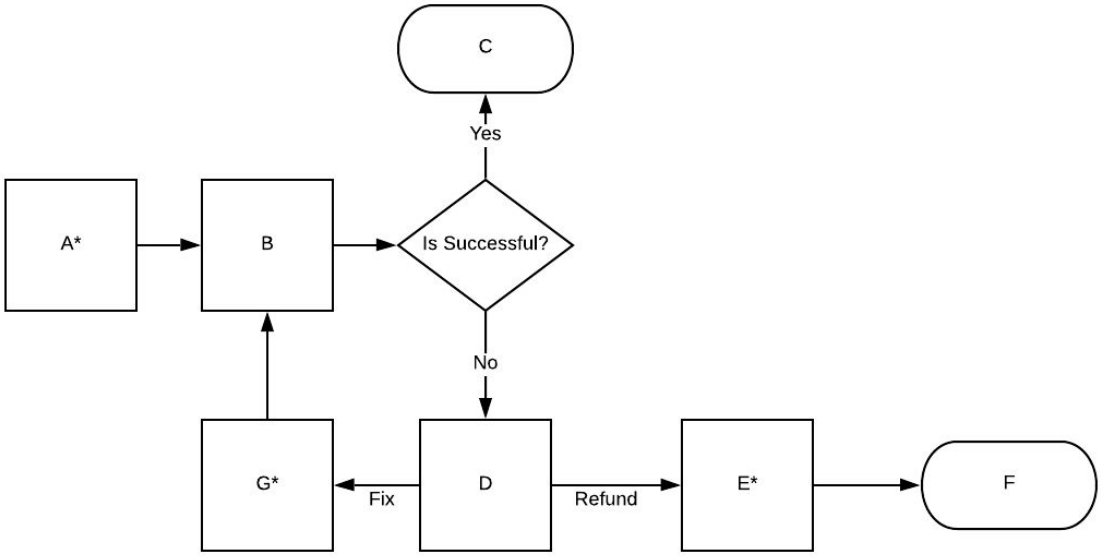
F:

status	refunded
location	originator_bank

G*:

status	fixing
location	routable

Transaction flow



Item and *Transaction* logic:
We want to tweak the flows from v1 a bit to have:

In the case that we have a failure (D) we now want to have 2 options:
1. Fix the transaction - retry sending funds to the Destination Bank.
2. Refund the transaction - sending the funds back to the Origination Bank.

We expect the application to work as follows:

Successful flow
Same as in v1.

Error flow
1. Create an item by using the "Create *Item*" endpoint.
2. Create a transaction by using the "Create *Transaction*" endpoint, starting in A (we now have an "active" transaction).
3. Move the "active" transaction to B, by using the "Move *Item*" endpoint.
4. Add an error on the "active" transaction, moving it to D, by using the "Error *Item*" endpoint.

And now, we can do either:

Refund item flow
1. Create a new transaction, starting it in E, by using the "Refund Item" button on the Item admin page.
2. Complete the "active" transaction moving it to F, by using the "Move *Item*" endpoint.

Fix item flow
1. Create a new transaction, starting it in G, by using the "Fix Item" endpoint.
2. Move the "active" transaction to B, by using the "Move *Item*" endpoint.
3. Complete the "active" transaction moving it to C, by using the "Move *Item*" endpoint.

Bonus:

Add a field on the Item model with the following values:

- processing (first time processing)
- correcting (any unfinished correction)
- error (in error)
- resolved (any positive finish state)

Each time a Transaction status changes, update this field. This should be a method that can be called on any Transaction, including v1