FastAPI REST – Part 8

CODE WITH FINESSE®

Ready to go?

Complaint system (course application – Part 5 - Wise)

TABLE OF CONTENTS

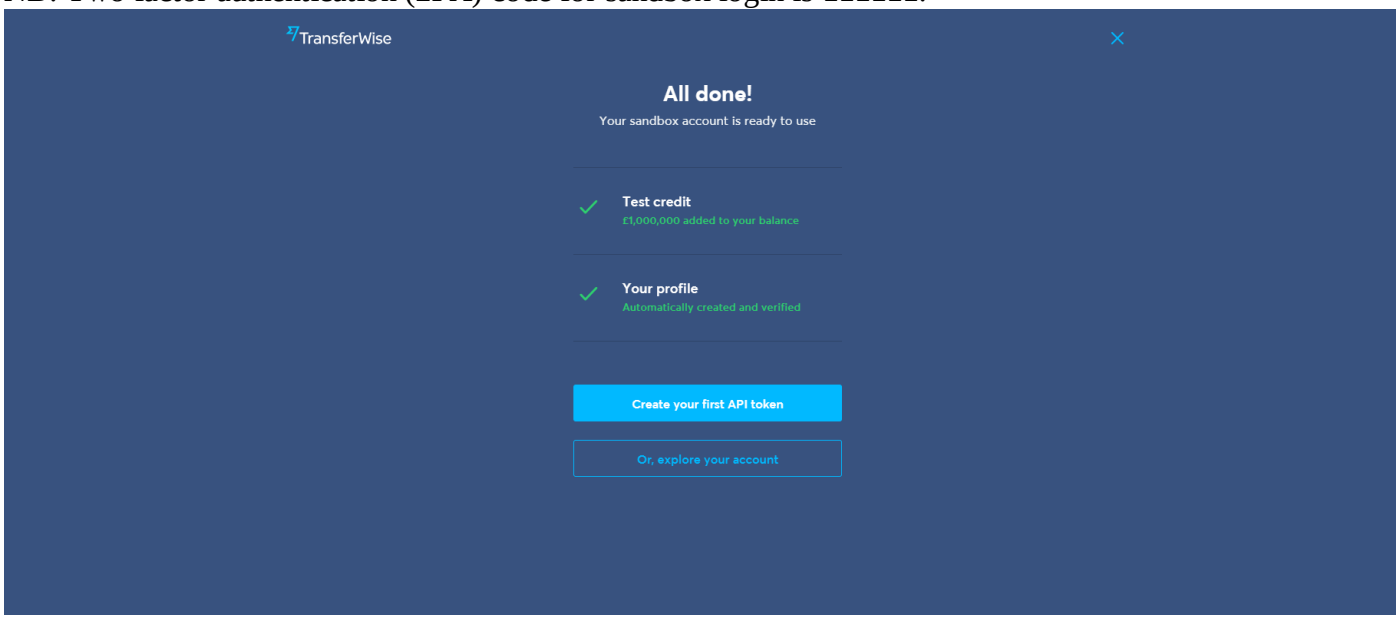
1. Introduction

Our goal would be to integrate Wise in our application.
This way, whenever an approver approves a complaint, we will release the claimed amount to the iban of the user that claimed it.
If the approver rejects it – we will just cancel the standby transaction in the service.

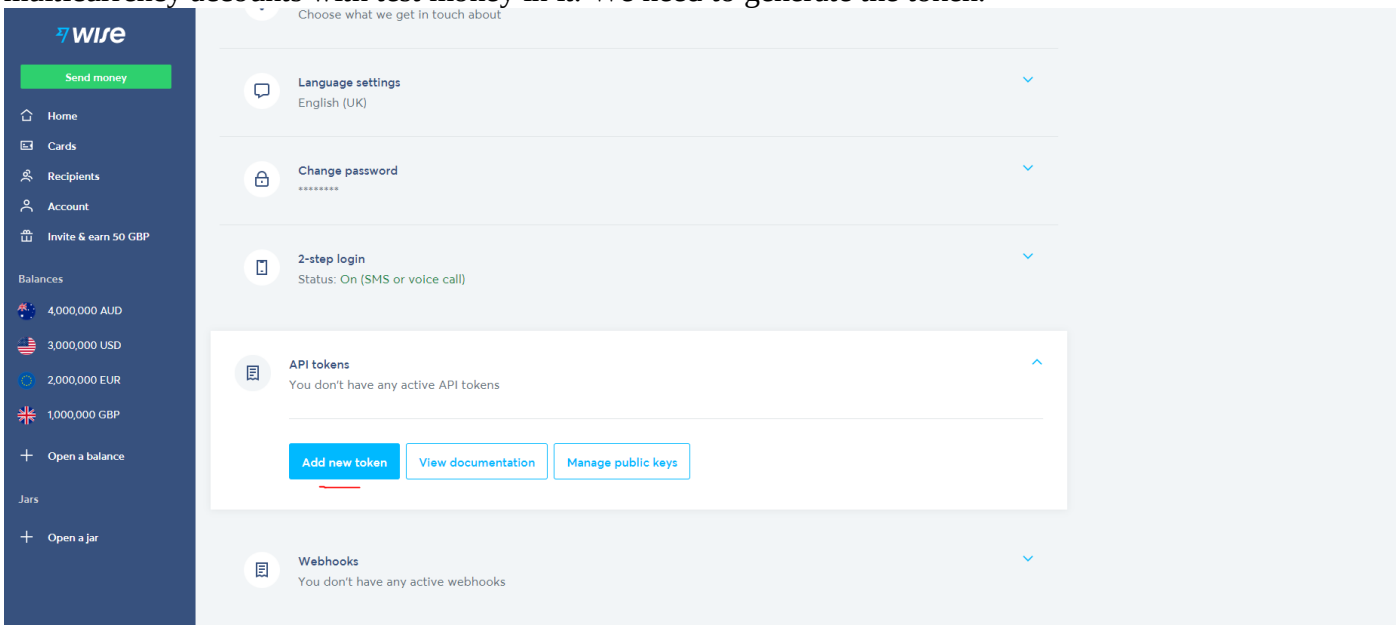
2. Wise setup

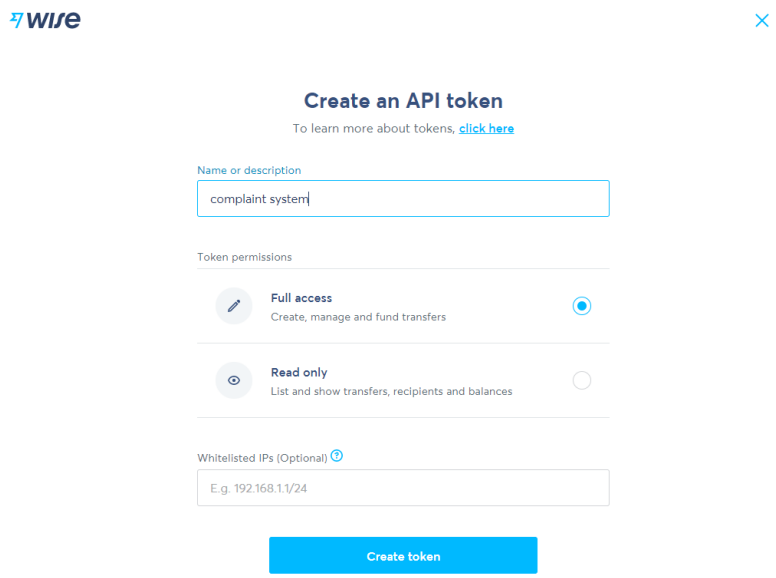
You can check the docs here. I suggest you follow the steps described in the docs, if they have changed their docs without notification and there are some differences.
Create a **developer** account from here.

NB! Two factor authentication (2FA) code for sandbox login is **111111**.

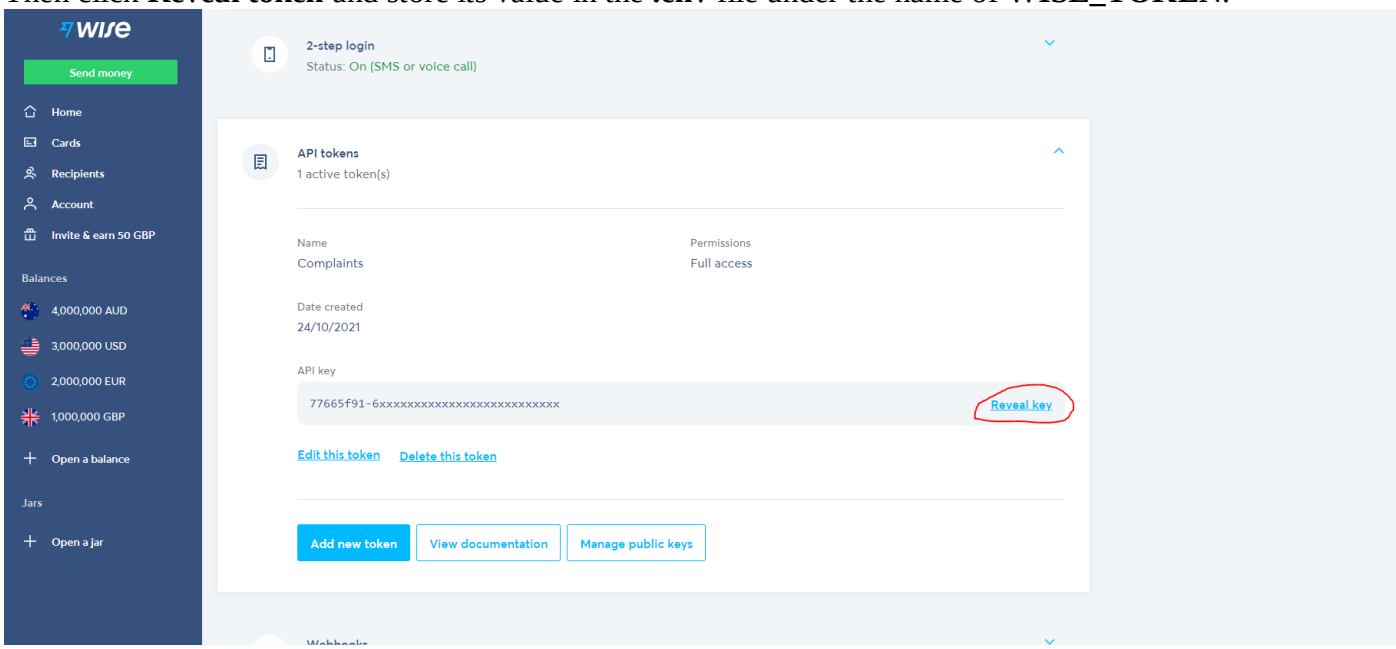


Click **Create your first API token**. Then choose personal account. Then you will see a couple multicurrency accounts with test money in it. We need to generate the token:





Then click **Reveal token** and store its value in the **.env** file under the name of **WISE_TOKEN**.



NB! Always work with sandbox while developing, otherwise if you set up a real card number the money will be fetched for real!

3. Issue transaction

There are a couple of steps to execute payouts according to the docs:

Step 0: Get your profile id
Step 1: Create a quote
Step 2: Create a recipient account
Step 3: Create a transfer
Step 4: Fund a transfer

Let's start by creating a Wise service in our code, which will describe all of this steps:
pip install requests

In the **wise.py** in **services** folder

```
import json
import uuid

import requests
from decouple import config
from fastapi import HTTPException


class WiseService:
    def __init__(self):
        self.main_url = config("WISE_URL")
        self.headers = {
            "Content-Type": "application/json",
            "Authorization": f"Bearer {config('WISE_TOKEN')}",
        }
        profile_id = self._get_profile_id()
        self.profile_id = profile_id

    def _get_profile_id(self):
        url = self.main_url + "/v1/profiles"
        resp = requests.get(url, headers=self.headers)

        if resp.status_code == 200:
            resp = resp.json()
            return [a["id"] for a in resp if a["type"] == "personal"][0]
        else:
            print(resp)
            raise HTTPException(status_code=500, detail="Payment provider is not
available at the moment")
```

Here we assign the headers and the profile id. We define a helper function _get_profile_id to make the request and extract the personal profile id. Refer to docs here to take a look at request fields and response format.

Then, we need to create a quote. We will extend our class by defining the following function:

```
def create_quote(self, amount):
    url = self.main_url + "/v2/quotes"
    data = {
        "sourceCurrency": "EUR",
        "targetCurrency": "EUR",
        "targetAmount": amount,
        "profile": self.profile_id,
    }
    resp = requests.post(url, headers=self.headers, data=json.dumps(data))

    if resp.status_code == 200:
        resp = resp.json()
        return resp["id"]
    else:
        print(resp)
        raise HTTPException(status_code=500, detail="Payment provider is not
available at the moment")
```

We are defining fixed currencies, because our system will pay only in EUR. For the **sourceCurrency** you should pick one of the predefined currency accounts. If you wish to work with different currency, then you need to open a new currency account from your profile.
Now, we will create a recipient account:

```
def create_recipient_account(self, full_name, iban):
    url = self.main_url + "/v1/accounts"
    data = {
        "currency": "EUR",
        "type": "iban",
        "profile": self.profile_id,
        "accountHolderName": full_name,
        "legalType": "PRIVATE",
        "details": {"iban": iban},
    }
    resp = requests.post(url, headers=self.headers, data=json.dumps(data))

    if resp.status_code == 200:
        resp = resp.json()
        return resp["id"]
    else:
        print(resp)
        raise HTTPException(status_code=500, detail="Payment provider is not
available at the moment")
```

Please note that we need only the iban here, because we will send in Europe. If you wish to try it for different countries, you need to refer to these dynamic forms and see the required fields.

We need to create a transfer:

```
def create_transfer(self, target_account_id, quote_id):
    customer_transaction_id = str(uuid.uuid4())

    url = self.main_url + "/v1/transfers"
    data = {
        "targetAccount": target_account_id,
        "quoteUuid": quote_id,
        "customerTransactionId": customer_transaction_id,
        "details": {},
    }
    resp = requests.post(url, headers=self.headers, data=json.dumps(data))

    if resp.status_code == 200:
        resp = resp.json()
        return resp["id"]
    else:
        print(resp)
        raise HTTPException(status_code=500, detail="Payment provider is not
available at the moment")
```

Last, we will define the fund transfer function:

Please, note that all POST requests require the data dictionary to be dumped (transformed to JSON format).

```
def fund_transfer(self, transfer_id):
    url = self.main_url +
f"/v3/profiles/{self.profile_id}/transfers/{transfer_id}/payments"
    data = {"type": "BALANCE"}

    resp = requests.post(url, data=json.dumps(data), headers=self.headers)
    if resp.status_code == 201:
        resp = resp.json()
        return resp["id"]
    else:
        print(resp)
        raise HTTPException(status_code=500, detail="Payment provider is not
available at the moment")
```

Now, we will define new model which will store our transaction information to the database. In file **transaction.py** in **models** folder:

```
import sqlalchemy

from db import metadata


transaction = sqlalchemy.Table(
    "transactions",
    metadata,
    sqlalchemy.Column("id", sqlalchemy.Integer, primary_key=True),
    sqlalchemy.Column("quote_id", sqlalchemy.String(120), nullable=False),
    sqlalchemy.Column("transfer_id", sqlalchemy.Integer, nullable=False),
    sqlalchemy.Column("target_account_id", sqlalchemy.String(100), nullable=False),
    sqlalchemy.Column("amount", sqlalchemy.Float),
    sqlalchemy.Column("complaint_id", sqlalchemy.ForeignKey("complaints.id")),
)
```

Do not forget to export it in the models/__init__.py
We need to migrate and upgrade once more.

```
from models.complaint import *
from models.users import *
from models.transaction import *
```

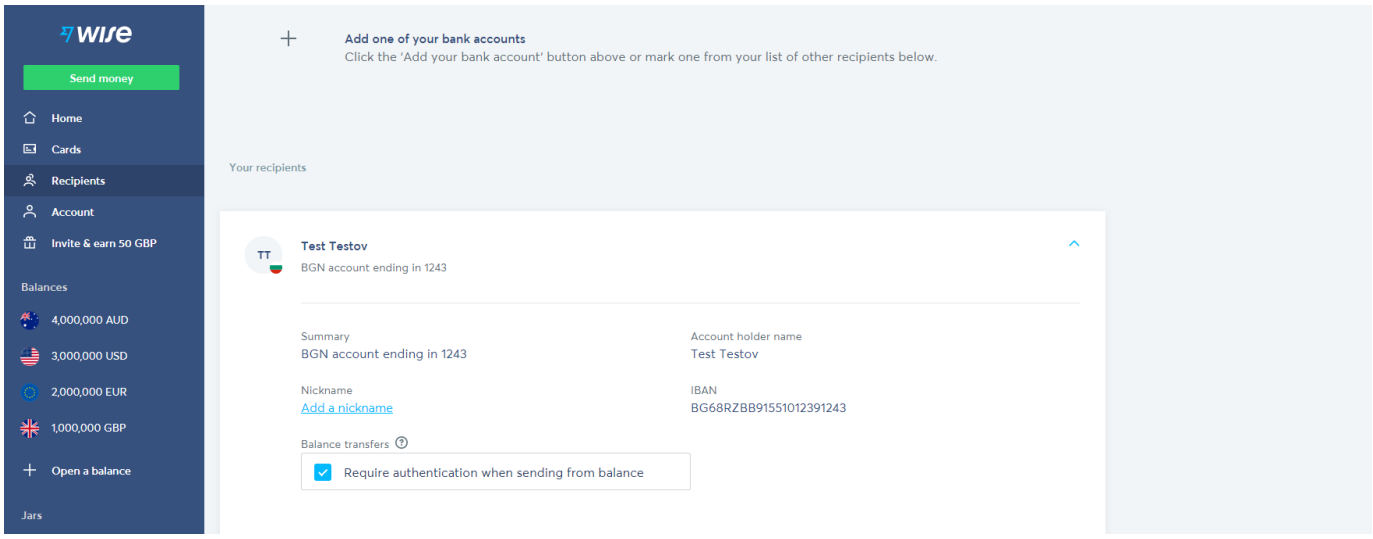
Now in the ComplaintManager we will define a function responsible for creating a transaction:

```
@staticmethod
async def issue_transaction(amount, full_name, iban, complaint_id):
    wise_service = WiseService()
    quote_id = wise_service.create_quote(amount)
    recipient_id = wise_service.create_recipient_account(full_name, iban)
    transfer_id = wise_service.create_transfer(recipient_id, quote_id)
    data = {
        "quote_id": quote_id,
        "transfer_id": transfer_id,
        "target_account_id": str(recipient_id),
        "amount": amount,
        "complaint_id": complaint_id,
    }
    await database.execute(transaction.insert().values(**data))
```

We will change a bit the create method in the manager:

```
@staticmethod
async def create_complaint(complaint_data, user):
    data = complaint_data.dict()
    data["complainer_id"] = user["id"]
    encoded_photo = data.pop("encoded_photo")
    ext = data.pop("extension")
    name = f"{uuid.uuid4()}.{ext}"
    path = os.path.join(TEMP_FILE_FOLDER, name)
    decode_photo(path, encoded_photo)
    data["photo_url"] = s3.upload_photo(path, name, ext)
    id_ = await database.execute(complaint.insert().values(**data))
    await ComplaintManager.issue_transaction(data["amount"],
f"{user['first_name']} {user['last_name']}", user['iban'], id_)
    return await database.fetch_one(complaint.select().where(complaint.c.id ==
id_))
```

If you make a request and the check the Wise's dashboard it will be something like this:



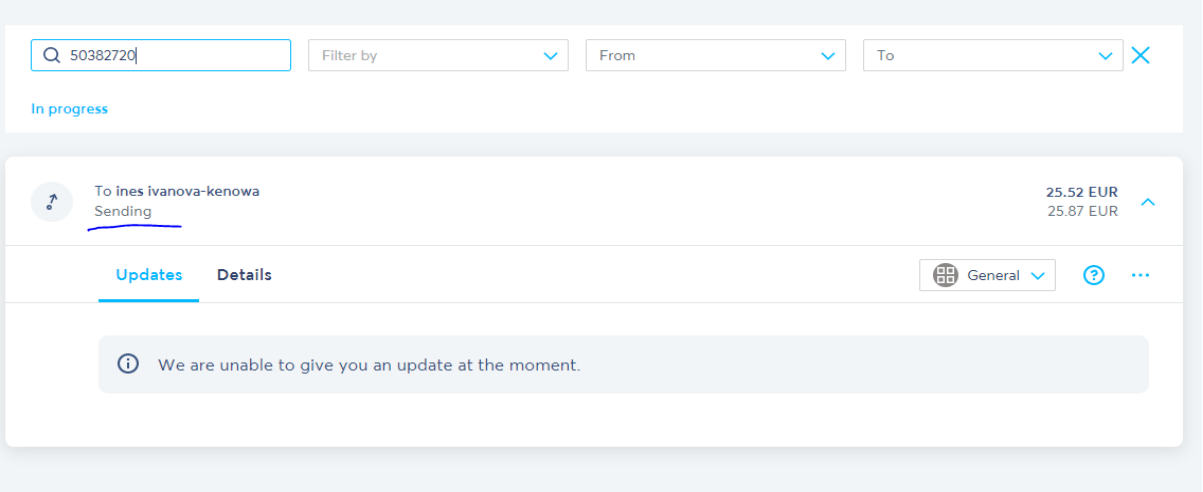
4. Release/Cancel funds

This is a very exciting moment, because now we are going to fulfill the business logic of the application – so far, we have created a complaint, storing its photo to s3, then we are issuing a transaction in Wise ready to be released or canceled.

We already have the code needed for releasing the funds from WiseService perspective, but this is not enough as we have to integrate this part in our managers.
The best place to integrate the releasing the funds part should be in ComplaintManager, the *approve* method.
Now we can add the following logic:

```
@staticmethod
async def approve(id_):
    await database.execute(complaint.update().where(complaint.c.id ==
id_).values(status=State.approved))
    wise = WiseService()
    transaction_data = await
database.fetch_one(transaction.select().where(transaction.c.complaint_id == id_))
    wise.fund_transfer(transaction_data["transfer_id"])
    ses.send_mail("Your complaint is approved", ["ines.iv.ivanova@gmail.com"],
"Congrats! You complaint is approved. Please check your bank account after 2
business days to verify the claimed amount is there.\n King regards!")
```

We are initializing the service and then we are calling the *fund_transfer*, in order to work properly it needs the *transfer_id*, we are fetching this from the transaction table for this *complaint_id* that we are approving at the moment.
If the request is successful, you can check the Wise's dashboard, search for the transfer id, and then you will see something like this:



It won't change further, because we are in sandbox mode and the payment would not be transferred for real.

The cancel part should be easy enough. Looking at the wise docs for cancelling we see we have to make PUT request to `/v1/transfers/{transferId}/cancel` request.
We can implement something like this in the WIseService:

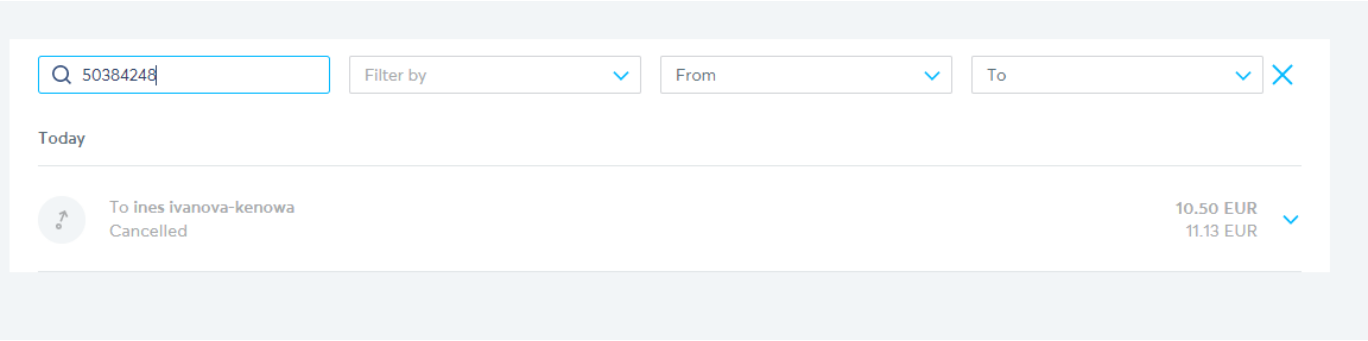
```
def cancel_transfer(self, transfer_id):
    url = self.main_url + f"/v1/transfers/{transfer_id}/cancel"

    resp = requests.put(url, headers=self.headers)
    if resp.status_code == 200:
        resp = resp.json()
        return resp["id"]
    else:
        print(resp)
        raise HTTPException(status_code=500, detail="Payment provider is not
available at the moment")
```

And the manager:

```
@staticmethod
async def reject(id_):
    wise = WiseService()
    transaction_data = await
database.fetch_one(transaction.select().where(transaction.c.complaint_id == id_))
    wise.cancel_transfer(transaction_data["transfer_id"])
    await database.execute(complaint.update().where(complaint.c.id ==
id_).values(status=State.rejected))
```

If you make a request as an approver to reject a complaint (which is not approved, but its status is still 'pending'), you will see the following in Wise:



Small quest – refactor the two function - Wise() should be initialized above, so you can remove the repetitive code.

5. Transactions

Our application is in a good state now, but it has some tricky parts: What if we create a complaint in the database, but for some reason, Wise is not reachable, or some error appears? This way, we will end up with a created complaint, but we won't have a transaction for it, and this indeed it is not what we want because every complaint should have an issued transaction. The easiest would be to use a try/except block, and if Wise is down, we will make another request to the database to delete it, right?
Well, this might seem a good solution at first, but as a second thought - this will make another transaction to the database, which for such small case it is not a big problem, but if we continue to do it, we will end up with a slowed application.
There is a solution to this common programming problem, and it is called transactions - this is a way of just 'flushing' the database instead of committing to it - we will use a with-manager to create a transaction block - if everything succeeds in this block, it will automatically commit, if an error appears, it will automatically rollback.
We will change a little bit the *issue_transaction* we will pass down a transaction like this:

```
@staticmethod
async def issue_transaction(tconn, amount, full_name, iban, complaint_id):
    quote_id = wise.create_quote(amount)
    recipient_id = wise.create_recipient_account(full_name, iban)
    transfer_id = wise.create_transfer(recipient_id, quote_id)
    data = {
        "quote_id": quote_id,
        "transfer_id": transfer_id,
        "target_account_id": str(recipient_id),
        "amount": amount,
        "complaint_id": complaint_id
    }
    await tconn._connection.execute(transaction.insert().values(**data))
```

Then in the *create* method we will do the following changes:

```
@staticmethod
async def create_complaint(complaint_data, user):
    data = complaint_data.dict()
    data["complainer_id"] = user["id"]
    encoded_photo = data.pop("encoded_photo")
    ext = data.pop("extension")
    name = f"{uuid.uuid4()}.{ext}"
    path = os.path.join(TEMP_FILE_FOLDER, name)
    decode_photo(path, encoded_photo)
    data["photo_url"] = s3.upload_photo(path, name, ext)
    async with database.transaction() as tconn:
        id_ = await tconn._connection.execute(complaint.insert().values(**data))
        await ComplaintManager.issue_transaction(tconn, data["amount"],
f"{user['first_name']} {user['last_name']}", user['iban'], id_)
    return await database.fetch_one(complaint.select().where(complaint.c.id ==
id_))
```

This way *async with database.transaction() as tconn*: we are saying – use this transaction, if everything is successful, commit everything to the databse, if an error appears do not commit anything – rollback (so if we have an error in this block it won't commit neither in the complaint table nor in the transaction table).

This was a slight change but very powerful because this way, we are guarantee that there won't be inconsistent data in our database. If you decide to extend the application further and somewhere again appears this behavior (one transaction to depend on another), you can implement *transactions* like above.