Business Process Automation

BPA - *Wintersemester 24/25*
Deliverable 3, Group 09

Process: Amazon order - Order-To-Cash

Benedict da Silva Araújo Finsterwalder, 617756
Wei Jin, 620999
Denis Kasakow, 648880
Marcel Lorenz, 641125
Damian Rychlicki, 572581

Humboldt-Universität zu Berlin

January 8, 2025

1 Video and Github

The link to the GitHub repository: github project.
The link to the HU Box with the screencast video: HU Box.

2 Implementation into UiPath

For this deliverable, we chose to implement our whole process into UiPath except for *Handle Shipmment* and tried to automate most of the tasks. In our case, this includes the following steps: **reading a customer's order**, **handling the order**, **handling refilling any missing stock** and **validating the payment**. We assume in this scenario that we always want to restock any missing items with the cheapest product from Amazon to maximize profits. We divided the tasks into five main fragments:

- reading an order (PDF scraping)
- handling order while checking stock availability
- refilling stock using web scraping for cheapest product
- validate payment (PDF scraping)

All of this can be found in the github repository in the **main.xaml** file. The *Validate payment* task is done separately and also in another repository, because that task is executed differently (for more detail look into the subsection 2.6).

2.1 Requirements

Our deliverable should include all necessary requirements for the deliverable:

- **file/folder handling**: reading PDF file and SQLite database
- **screen scraping**: scraping of cheapest product on Amazon website
- **browser actions**: entering item name into search bar as well as putting products into the basket
- **loops/decision points**: for loop for every ordered item, decision points whether the item needs to be restocked or not

2.2 Structure

The structure of our UiPath implementation is rather straight forward. We start by reading the order PDF from our Google Drive folder, then we scrape the order from the PDF and turn it into a datatable. This datatable is used in Handle order to fulfill the order from the point of the warehouse. So if we sell the item, we look in our database if we still have it in stock or if we have to refill our stock. The refilling consists of going to the amazon website, searching for the item, scraping the website and then ordering the item(s). After this, we send the invoice with the cost to our customer using Sendgrid. The last step is validating the payment. We wait for the payment and check if the amount is correct. The shipment is not automated using UiPath since this is a manual task.

2.3 Reading the order

Reading the order is using a trigger for Google Drive. The trigger in UiPath always reads the newest created/added file in the Drive. It then extracts the PDF text using the Plugin in of *UiPath.IntelligentOCR.StudioWeb*. Now the text has to be parsed and processed into a datatable format for easier handling of the customer's order. The text is extracted for key information such as: **name**, **address**, **email** and **order**. The output is then used for the next activity *Handle order*.

2.4 Handle order

Handle order, as the name implies, represents the old process from Camunda 8 and consists of the processes Check stock and Refill stock in a for each loop. The UiPath implementation starts with getting the order as a datatable and connecting to the SQLite database of our stock. We then iterate through the order and first check the stock, then refill the stock if necessary, and updating the stock at the end.

1. Get input (order and stock)
2. Check stock
3. Refill stock if needed
4. Update stock
5. Update our price if refill item is more expensive
6. Return output (ordered items with prices)

2.4.1 Check stock

Check stock works by getting the item from the for each loop and then checking the SQLite database for this item. If the item is not in our database, we do not sell this item and discard it. If the item is in our database, we check if the amount of items we have in stock is enough to fulfill the order. If this is the case, we just deduct the ordered amount from our stock and go on to the next item. This is simply done by updating the database with the stock amount after the ordered items are sent. But if the amount of items in our stock is not enough to fulfill the order, we have to refill the stock of this item. This is done in the Refill stock part of Handle order and explained in the next subsection.

2.4.2 Refill stock

For this task we assume that the customer is very nice and does not mind, if the product they ordered varies (i.e. soccer ball from different vendors). We also assume that the customer agrees to pay any increase in prices if the restocked item is more expensive. The final implementation handles the increase in prices as followed:

Assuming our selling price for example item "*soccerball*" is 10€ and we have an customer ordering 5 soccerballs:

- if we have enough soccerballs in stock, the total cost is **50€**
- if we do not have enough in stock, we encounter two possible scenarios:
  1. The item that we buy to refill is cheaper (i.e. 7€):
     ⇒ we still sell the item for the initial price of **10€** resulting in a total cost of **50€**, we put the profits (10€ − 7€ = 3€ profit per item) in our pocket
  2. The item that we buy to refill is more expensive (i.e. 13€):
     ⇒ we update the cost of our price to 13€, which means a total cost of **65€** for the customer

Before we refill any stock, we first have to check whether the amount the customer ordered of any item is more than what we have in stock. In cases where we do not have enough items, we initiate the automation to refill the item. We save the name, refill amount and our price of the product which need to be refilled into variables.

We open the standard browser (in our case Google Chrome) and head to the URL www.amazon.de. Afterwards, we enter the saved name variable into the search bar and press the "*Enter*" key. Now, we should be on the page where numerous items are listed at different price points and sellers. We use UiPath's data extraction and select the following attributes to extract the data from: **Name**, **Price** and **URL**. These combinations yield a small yet consistent amount of results. Different attributes lead to chaotic results and make it more difficult to process the data. We also save the extracted data in a data table. If a row has any missing values, we ignore that product. For the rest we proceed as followed:

1. the first item in the list is saved as the cheapest
   - we also save the price and URL to the cheapest product
2. we iterate through each row to determine the cheapest item
3. now we calculate the cost discrepancy
   - if the item is cheaper, we take the difference as an increase in profit and also save the more expensive cost as our selling cost (that means we sell all the order items for that price cost)
   - if the item is more expensive than our listed price, we let the customer pay the difference (we calculate the extra cost by multiplying the difference times the amount we have to restock to fulfill the customer's order)

⇒ we update the price in our stock database to the new price

4. we save the name, price, URL, cost discrepancy (0.0 if cheaper) into a string array
5. for each item which needs to be refilled, this process will be repeated and finally added to another array

At the end of the automation, all the necessary data can be accessed through the variable "*refillOrder*". Of course, no real refill order will be done on Amazon and the automation stops before the Check-out. So no worries - we do not help amazon sell items ;)

2.5 Send invoice

In the previous deliverable with Camunda, we sent the invoice with the total cost to the customer using Sendgrid's API integration in Camunda 8. For this deliverable, we use UiPath to connect to Sendgrid and send the invoice. All the necessary data as well as the total price is calculated via UiPath and respective activities. We process them to make it easier to send them via Sendgrid email.

For the total price, we go through each product in the customer's order and receive the final cost out of our stock database via a query. We aggregate the cost by multiplying the order amount with the respective costs. We save the total cost for the email later. The extracted name, email and customer from the beginning is used to fill in the parameters for the Sendgrid activity. A messageJSON is also handed over to Sendgrid, which contains the customer's order, the item price and the total cost.

2.6 Validate payment

For this task, we originally planned to create an automated process that processes PDF bank statements to check for payments for corresponding order (IDs). Each time a new bank statement is received from the bank which lists all transactions, we check if the payment for the order has been received.

For this the bank statement is parsed by a Python script that is executed by a python actifity in the UiPath process. The function within the script returns the statements table data as JSON text, which is parsed into a DataTable wintin the process. Then each row where the transaction text fulills the structual requirements is filtered out using RegEx and the order-id is extracted.

The original task in the Camunda process set a boolean variable if a managing user manually confirmed the payment. Until then the user would receive payment reminders. But since we didn't integrate the UiPath processes into Camunda we had no input for the order-ids to check for without hardcoding them and also the could return confirmation information. That's why we didn't proceed on automating the payment validation because it was a fully detached process we couldn't integrate. Still we want to provide the result for this process too although it's not part of our final submission, so we uploaded it to a different GitHub Repo. This version of the process reads all files from a specified directory, by default the docs-dir within the process project folder, and parses the payments and stores the Oder ids and payments in a dictionary.

3 Running UiPath

To run the implemented process in UiPath, simply follow these steps:

- Start by donwloading UiPath Studio Desktop through the UiPath website. First create an account, then log into UiPath Cloud and click on "Studio" on the left side. Next click on "Install locally" and "Studio for Desktop" to download it.
- When UiPath Studio Desktop is installed locally, simply start it. In "Open" you can choose to clone a project from a Github repository. Fill in the necessary information to clone our repository from Github.
- After cloning, open the project. You will notice that the trigger event is not connected to Google Drive. For this click on the Trigger event (see figure 2) at the beginning of the main sequence and connect your Google account. Make sure to give UiPath enough permissions. Next you have to create a folder in Google Drive and connect the trigger to this folder. After creating the folder, you have to click on the small folder symbol and look for your folder using the drop down menu as the path is not correct otherwise - only the correct name **is not enough**. See figure 3.

- Now upload a customer order to aforementioned folder which has the same layout as the provided PDF (*dummy_invoice.pdf* - can be also found in the repository). Also, if the ordered item is not in our database, the code runs into an error - so make sure to only order "pencil", "tablet" or "soccerball". Make sure that the email is set correctly such that you can get the email - otherwise poor Wei has to be spammed by Nozama invoices which he has to pay. If you encounter problems when changing the email on the PDF file, it is also possible to change the email receiver in the code itself in UiPath (has to be of type string). Please look at the provided screenshot (see figure 1) to see where to change the value, it is in the lower/final part of the code.

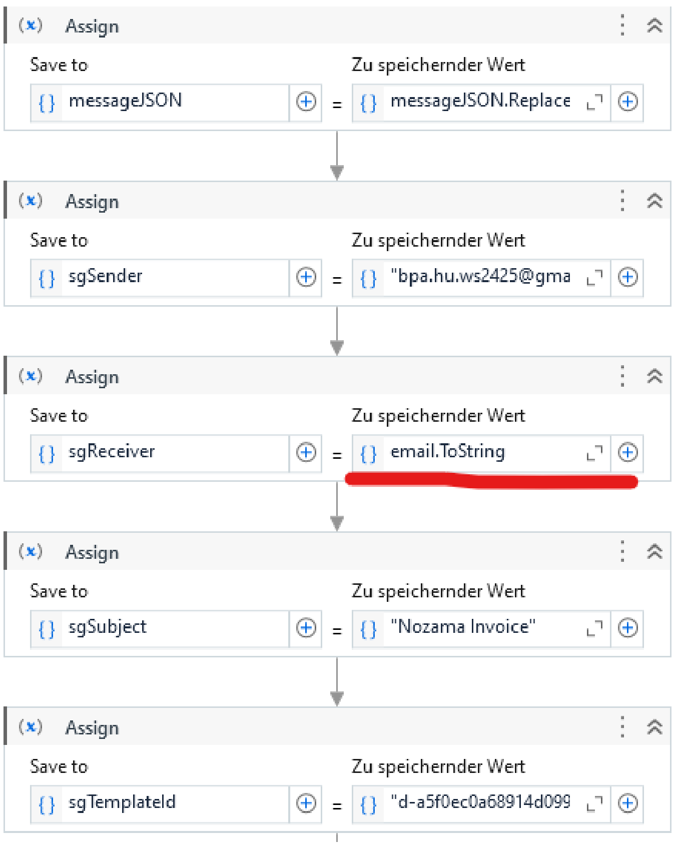
- The Database also has to be connected. Simply look for the "Stock" file which is included in the Github repository and copy the absolute path to this file. Insert the path into the "Connect to database" activity in your main sequence and don't forget to add the name of the database to the end of the path, e.g. "C:\Users\RealBenTen10\Downloads\Stock"".

- Google Chrome is used for scraping the web. Please install it if not already installed. A plug-in for UiPath exists which should be downloaded and enabled in the settings such that UiPath can search and scrape using Google Chrome. For this open UiPath Studio and click on "Start" → "Tools" → "UiPath-Extensions" and install the Chrome extension. After installing simply open Google Chrome and you will see a pop-up. Click on "Activate extension". To make sure that the extension is activated, simply go to the Google Chrome extensions. Here you should see "UiPath Browser Animation 24.10" and it should be enabled.
  Some troubleshooting: Make sure that when a new window in Google Chrome is opened, the url can be typed in directly and no pop-ups appear. If the website "www.amazon.de" is in german, change the language to english (next to the serach bar) but don't change the website name to "www.amazon.com".

- Finally you have to connect Sendgrid. Scroll down to the activity "HTTP Request" (scroll down to the end of the xaml file - to the end of the last sequence) and connect Sendgrid. If you can't find it, look at figure 4.
  Use the following API-Key for this:
  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

- After ensuring that everything is connected and UiPath Studio does not show any warnings, simply click on "Run", "Run file" or "Debug file" to start the automated order handling.

The payment validation is implemented separately and can be run independently. To run this process you have to clone the additional repo. Afterwards you need to have a working python environment. You need to install these packages: *pdfplumber*, *pandas*. Then you need to specify the path to your python interpreter in the properties of the python scope activity. If everything is setup correctly and now directories relative to the repo where changed the process can now be executed. It should print the order id and the payment amount into the output.



Assign
Save to: messageJSON = Zu speichernder Wert: messageJSON.Replace

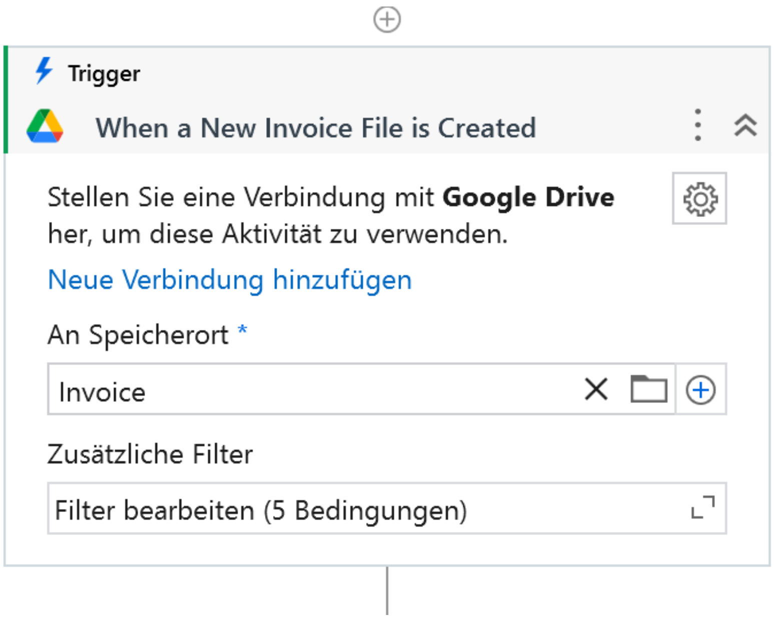
Assign
Save to: sgSender = Zu speichernder Wert: "bpa.hu.ws2425@gma

Assign
Save to: sgReceiver = Zu speichernder Wert: email.ToString

Assign
Save to: sgSubject = Zu speichernder Wert: "Nozama Invoice"

Assign
Save to: sgTemplateId = Zu speichernder Wert: "d-a5f0ec0a68914d099

Figure 1: Screenshot where to change email receiver



Trigger
When a New Invoice File is Created

Stellen Sie eine Verbindung mit **Google Drive** her, um diese Aktivität zu verwenden.

Neue Verbindung hinzufügen

An Speicherort *: Invoice

Zusätzliche Filter: Filter bearbeiten (5 Bedingungen)

Figure 2: Screenshot for Google trigger activity

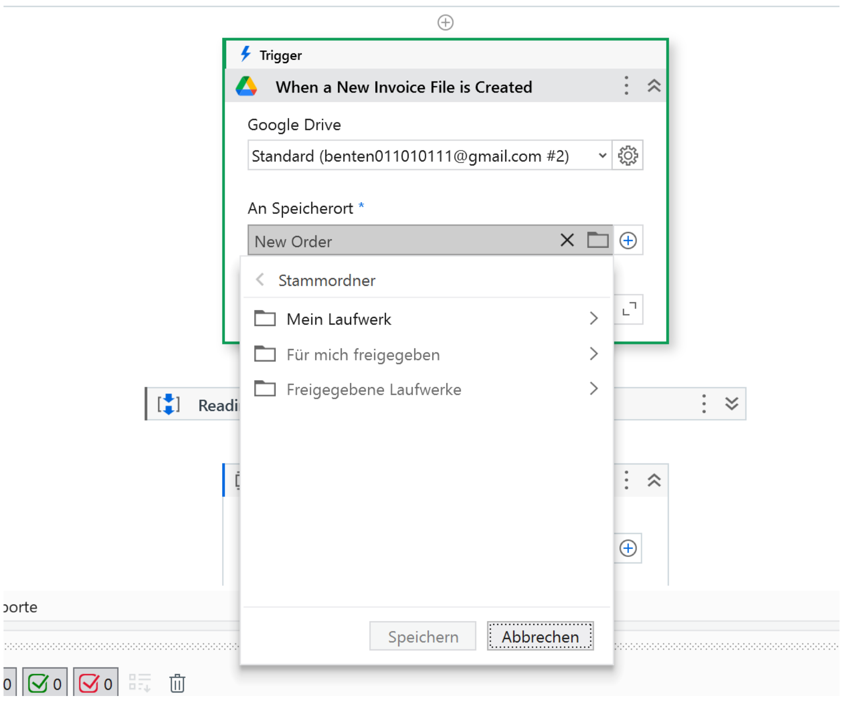
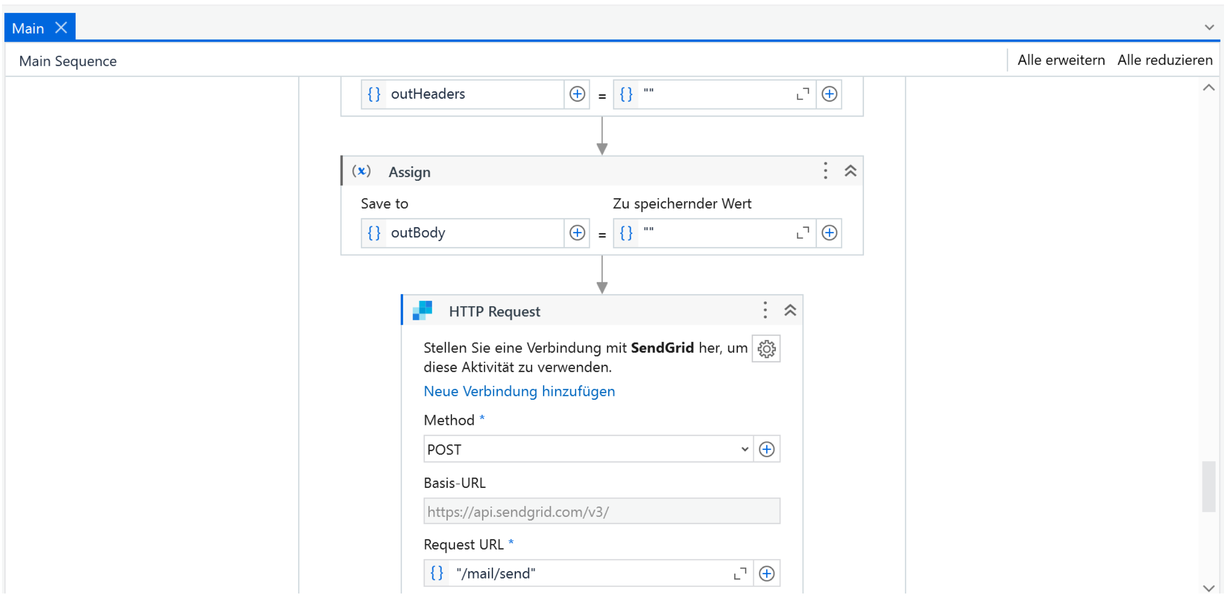

Trigger
When a New Invoice File is Created
Google Drive
Standard (benten011010111@gmail.com #2)
An Speicherort *
New Order
Stammordner
Mein Laufwerk
Für mich freigegeben
Freigegebene Laufwerke

Figure 3: Screenshot for Google folder



outHeaders = ""
Assign
Save to
Zu speichernder Wert
outBody = ""
HTTP Request
Stellen Sie eine Verbindung mit **SendGrid** her, um diese Aktivität zu verwenden.
Neue Verbindung hinzufügen
Method *
POST
Basis-URL
https://api.sendgrid.com/v3/
Request URL *
"/mail/send"

Figure 4: Screenshot for Sendgrid activity