

# Business Process Automation

BPA - *Wintersemester 24/25*

**Deliverable 2, Group 09**

**Process:** Amazon order - Order-To-Cash

**Benedict da Silva Araújo Finsterwalder, 617756**

**Wei Jin, 620999**

**Denis Kasakow, 648880**

**Marcel Lorenz, 641125**

Humboldt-Universität zu Berlin

December 11, 2024

## 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 Camunda

Instead of selecting a subprocess of the model, we decided to make the whole model we discovered in deliverable 1 executable in Camunda, since a single subprocess would not fulfill all requirements - or we would have to restart the process discovery step for that subprocess. So we took the whole model but we had to further simplify the processes to keep the workload appropriate. Figure 1 shows our original Process model from deliverable 1 using SAP Signavio while Figure 2 shows our implementation for deliverable 2 in Camunda 8.

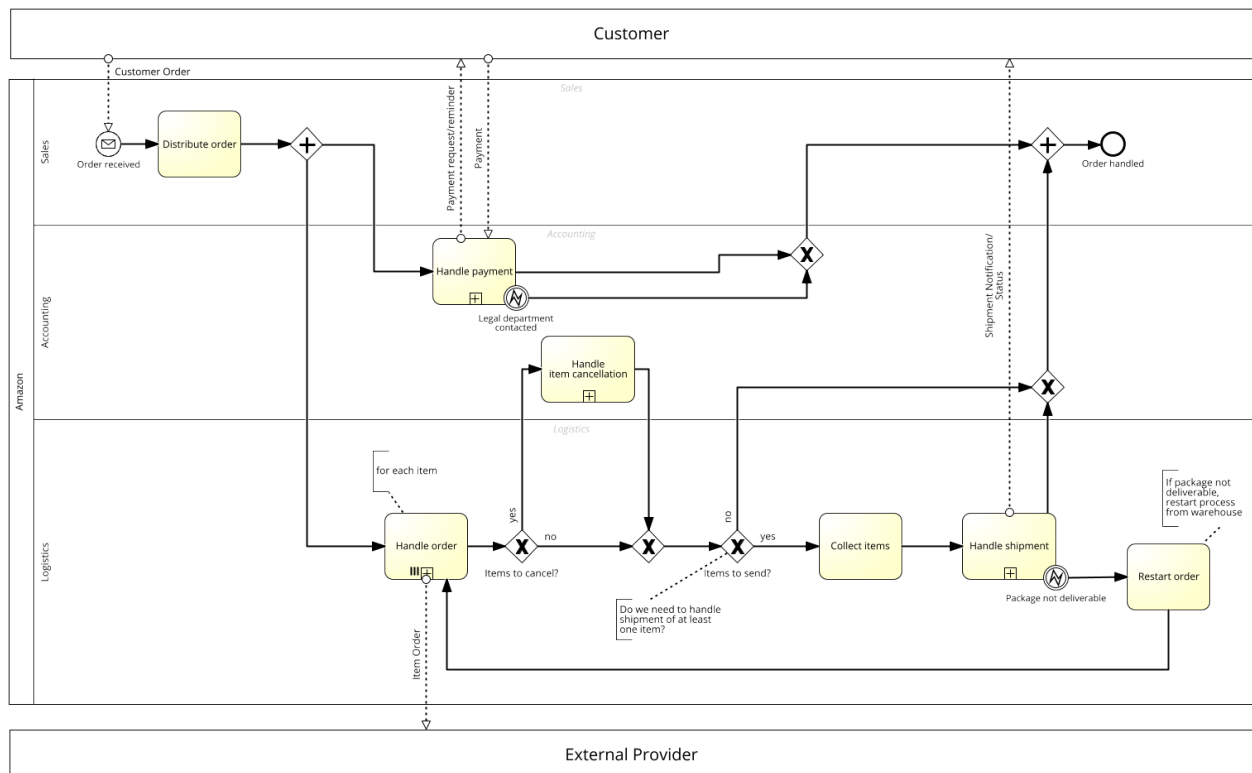


Figure 1: Process - Amazon order (Screenshot from SAP Signavio)

## 2.1 Model changes

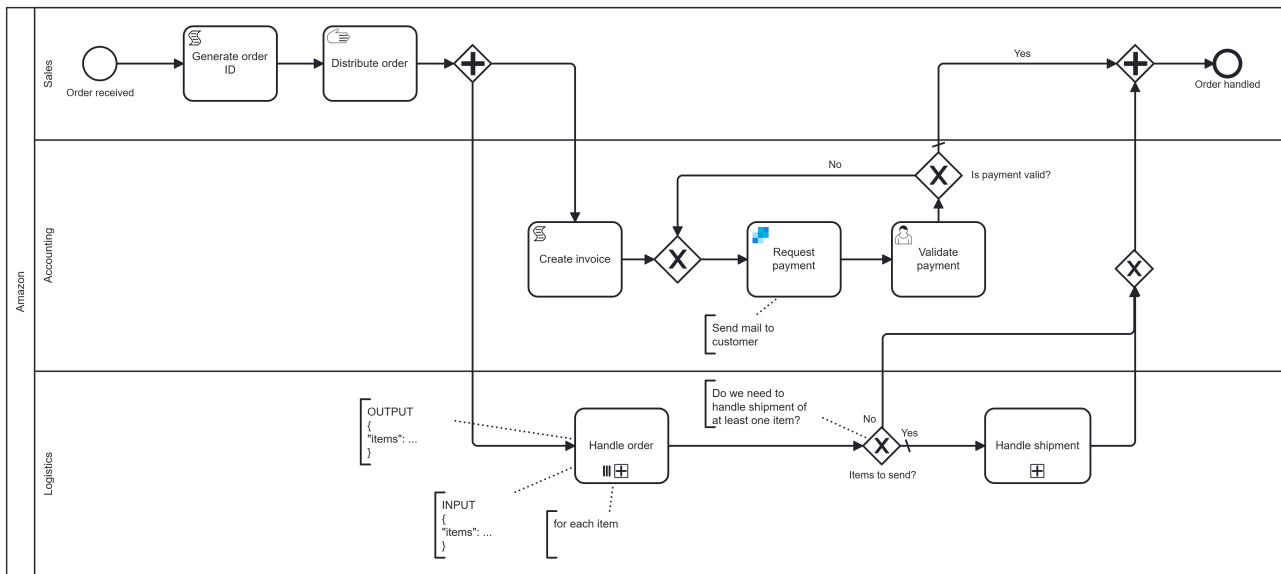


Figure 2: Process - Amazon order (Screenshot from Camunda)

The following enumerated list explains our steps and decisions for the changes on the model we discovered in the first deliverable. In the next subsections we go into more detail of each subprocess and explain the behaviour. This enumerated list shows the global context of the changes:

1. We started by importing the model from SAP Signavio into Camunda 8
2. First we trimmed the model by removing tasks which are not important for the later simulation (test run)
  - (a) "Handle item cancellation": We decided to drop this subprocess, since we wanted to focus on our main process which is an order-to-cash process and does not (necessarily) need a cancellation process. This also results in the XOR-gateway ("items to cancel") being removed.
  - (b) "Collect items": We added this process before handle payment with the intent to be able to send items together. We omitted this task, since in the simulation we don't gain anything from this step
  - (c) "Handle order": We simplified this subprocess by assuming that if we do not have an item in stock, we simply refill which is always successful. No more forwarding to external sellers and no more item cancellation
  - (d) "Handle Shipment": Overall this subprocess is quite similar to before. We start by packaging the ordered items, generate shipment documents, arrange the pickup and then send the package through a delivery service. The delivery status is sent per mail to the customer every 20 seconds (for simulation purposes). The loop is dependent on the duration which we define in our input (see start event). So for duration = 2, we send 2 notifications. We removed the "package not deliverable" scenario to further simplify the model.
3. Then, the tasks inside the subprocesses of our simplified model were adjusted to fit the necessary steps
  - (a) "Request payment": external service/sendgrid connector
  - (b) "Check stock": manual task
  - (c) "Refill stock": script task
  - (d) "Package items": manual task
  - (e) "Generate shipment documents": script task
  - (f) "Arrange pickup": user task
  - (g) "Arrange shipment": manual task
  - (h) "Update delivery status": script task
  - (i) "Send delivery status": external service/sendgrid connector
  - (j) "Drop off package": user task

4. Next we defined our input and output of each step (first using text annotation) to ensure a flow of data from start to end
5. To start our flow of data, we modified the start of our process such that we can input example data and have an `example_order` object which we can reference inside the subprocesses
  - (a) **Order received** is still our message start event but includes the customer data in the output `example_order`
  - (b) **Generate order ID** is a script task and follows right after the start event and generates a random order ID which is used for the emails in the simulation run
  - (c) **Distribute order** is a manual task and is still the same as before. It represents the distribution of the order to "Accounting" and "Logistics" from "Sales"
6. Afterwards we made sure that all inputs and outputs of all tasks reference the correct data, e.g. `example_customer.customer.name` for the name of the customer on forms in the user tasks
7. We then decided to put the subprocess **Handle payment** into the main process so we expanded the subprocess and then integrated it directly into the main process.
8. At the end we started a test run (play) and simulated the process to check for errors or wrong implementations

### 2.1.1 Main process

For the whole order-to-cash process we kept the original logic from deliverable 1 but simplified a bit. We kept **Handle payment**, **Handle order** and **Handle shipment**, but removed **Handle item cancellation** with its' XOR-gateway "Items to cancel?", **Collect order**, **Restart order**, as well as the error events "Legal department contacted" and "Package not deliverable". We added a task called **Generate order ID** at the beginning.

The model starts with the message start event **Order received** where the customer input is generated. Next we generate the order ID using a random generator inside the script task **Generate order ID**. Then we have a manual task where we distribute the order called **Distribute order**. After we created our global input, we have an AND-split. One branch consists of the subprocess **Handle payment** which is explained in subsection 2.1.2. The second branch consists of the **Handle order** (see subsection 2.1.3) and **Handle shipment** (see subsection 2.1.4) subprocesses where the second subprocess is only executed if we have items to send. We only have no items to send, if the order has an orderCount of 0, since we always restock the missing items inside **Handle order** as shown in Figure 4.

### 2.1.2 Handle payment

For the subprocess **Handle payment**, we chose to drop the payment reminder tasks and events since the responsibility can be shifted to another process outside the order-to-cash process and is therefore not our focus here. Additionally, we decided to expand the subprocess and deleted the collapsed subprocess, since it was small enough to fit into the main model. This also enhances the readability of our process. The events of request and receive payment are changed to tasks and we changed the task of receiving the payment to **Validate payment** and added an external service to the task of **Request payment**. For the external service of **Request payment** we chose SendGrid, since it is free and we can send emails through this service. The Sendgrid outbound connector is implemented using a Template (already implemented by Camunda), SendGrid API key (creatable after signing up in SendGrid), Sender name and email-address, Receiver name and email-address (which can be read from the initial input) and a function called "Compose email" where we can specify the content of our email (e.g. dynamic template with dynamic template data from the input). It is also possible to add error handling, retries for mail delivery and output mapping, but we did not use these functionalities. **Receive payment** is not further specified. For the user task, the only thing necessary is to check whether the received payment equals the total cost and checking the corresponding checkbox. If the payment is false, we repeat again with the **Request payment**.

At the end we collapsed the subprocess and integrated the tasks into the main process but in the same order and with the same logic.

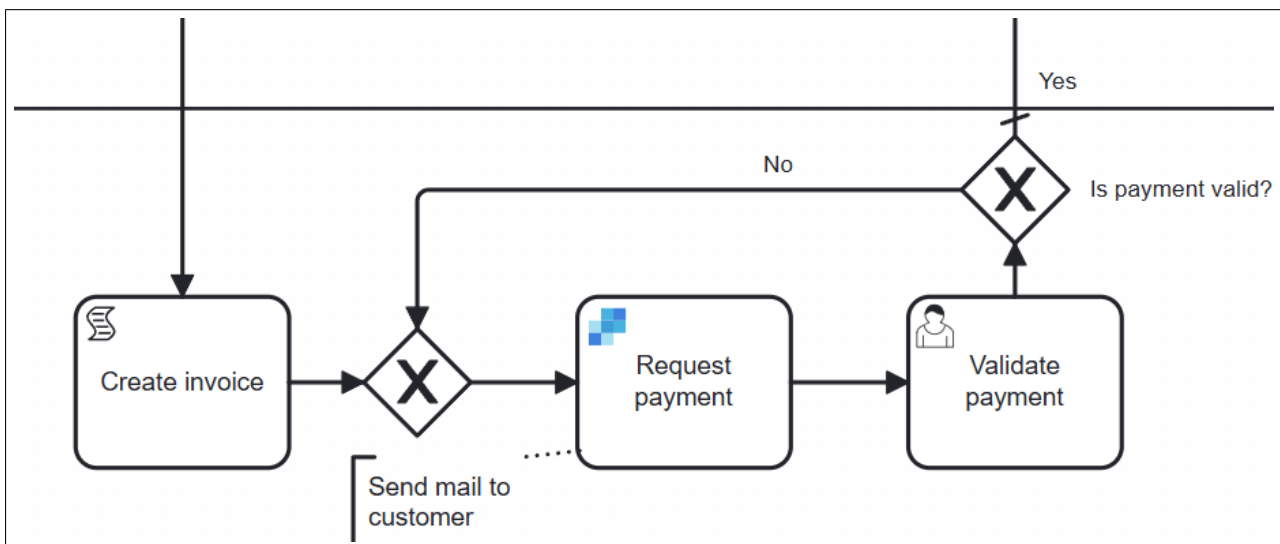


Figure 3: Subprocess - Handle payment (Screenshot from Camunda)

The external service sends an email from a custom template to the customer with the items of the order. The customer name and the item list are taken from the initial data of the customer's order. The order number is created at the beginning of the instance.

### 2.1.3 Handle order

The subprocess **Handle order** has been more simplified but is still a Multi-instance subprocess. This means that the subprocess is executed for each item in the items list. For this, the Input collection is set to "example\_order.items", the Input collection to "item", the Output collection to "items" and the Output element to "item". The output of the subprocess also includes the items\_to\_send variable which counts the number of items using "count(item)" which is necessary for the XOR-Gateway "Items to send?". Inside the subprocess **Handle order** we assume that the order only consists of items that we can deliver, we omit the first task of checking responsibility as well as the first XOR-gateway. The forwarding of the customer's order is therefore not necessary anymore. Also, we combine the two end events into one single "*Item handled*" end event, because we also assume that any stock, in case of insufficient stock, can be refilled. The task "Register unavailable items" from deliverabl 1 is therefore omitted. "Check Stock" is a manual task where the stock is checked. The XOR-Gateway "Product in stock?" checks the values of stockCount against the values of orderCount from the example.customer.items object. If the orderCount is greater than the stockCount, we have to refill the stock. This is implemented in the script task "Refill stock". The item.stockCount is set to "item.stockCount + item.orderCount + 100".

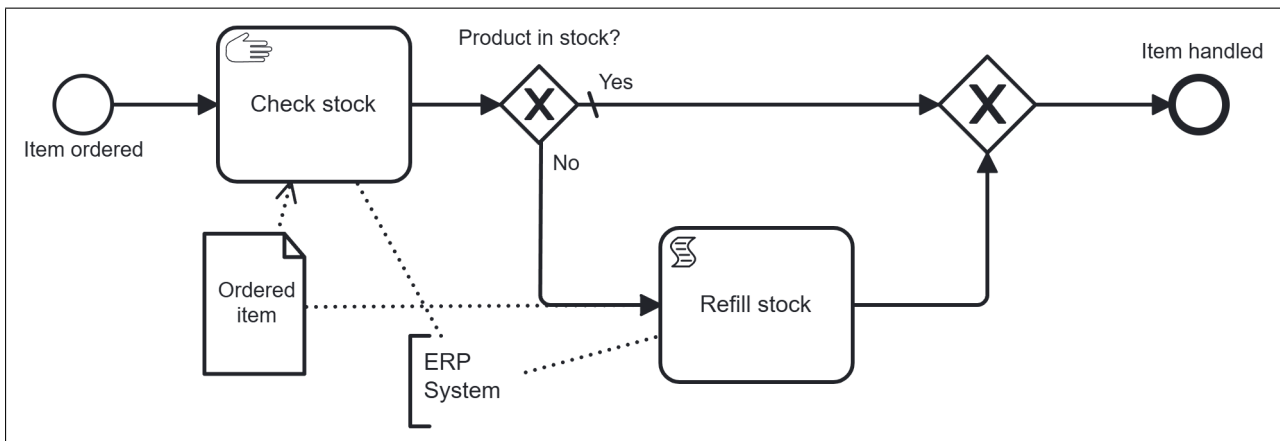


Figure 4: Subprocess - Handle order (Screenshot from Camunda)

### 2.1.4 Handle shipment

Inside the subprocess **Handle shipment** we still have a similar structure to before. We removed the XOR gateway which checked whether we had something to send by Amazon or someone else. Now we include other postal services into the "Arrange pickup" activity. Also, we changed the "Problem occurred?" to a loop which just checks whether the package is delivered. We assume that any problems would be handled inclusively in the "Update delivery status" activity. In our scenario we just have to wait 20 seconds, but this is of course just for demonstration purposes and in a real life scenario this waiting time would be longer.

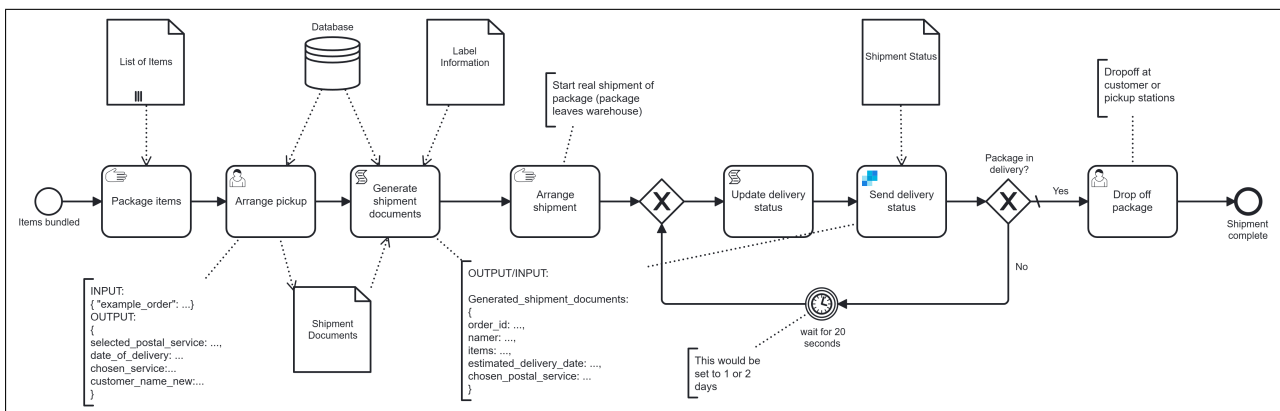


Figure 5: Subprocess - Handle shipment (Screenshot from Camunda)

In this subprocess, we included two user tasks, two script tasks and one external service of sending an email with the order's delivery status/confirmation. The email template is a similar one to the template used in subsection 2.1.2. The estimated delivery date is taken from the output variable of the script task in **Generate shipment documents** and is calculated by adding the "*duration*" to the "*date\_of\_delivery*". The date of the delivery is determined in the user task **Arrange pickup** and can be set by the user manually. Furthermore, the user should check whether the customer's credential are valid and could, in case of any errors, leave a note. The second user task **Drop off package** just confirms whether the recipient of the package is the customer or someone else in case the customer was not at home. In that case, the user can click the checkbox "Different recipient" and enter the credentials of that person.

## 2.2 Used variables and forms

This section lists all used variables and forms throughout our processes.

### 2.2.1 Variables

At different steps in our workflow, we generate variables for different tasks. In this list we show the input and output variables of all tasks which use variables.

- **Run Variables**

This variable has to be set in the Variable field when running the model. The following input is an example of how the input could look like. The values can be changed, but the keys have to be the same.

– Input:

```
{
  "example_order": {
    "items": [
      {"product": "pencil", "orderCount": 10, "stockCount": 8, "cost": 1.0},
      {"product": "tablet", "orderCount": 1, "stockCount": 20, "cost": 180.49},
      {"product": "soccerball", "orderCount": 2, "stockCount": 96, "cost": 22.50} ],
    "customer": {
      "name": "Peter Parker",
      "address": "Rudower Chausee",
      "number": 26,
      "postal_code": 12489,
      "city": "Berlin",
      "email": "wei.jin@hu-berlin.de" },
    "duration":
      2
  }
}
```

- **Generate order ID** generates a random order ID for each running instance.

– Input: -

– Output: order\_id

- **Create invoice** generates the total cost from the input (example\_order) by multiplying the orderCount with the cost of each item from example\_order.items and then make the sum.

– Input: example\_order.items

– Output: total\_cost

- **Request payment** sends a mail to the customer with all important information. Order ID, list of items, number of each item, cost of each item per piece and total cost of the order.

– Input: order\_id, example\_order.items, total\_cost

– Output: E-Mail (see Figure 6)

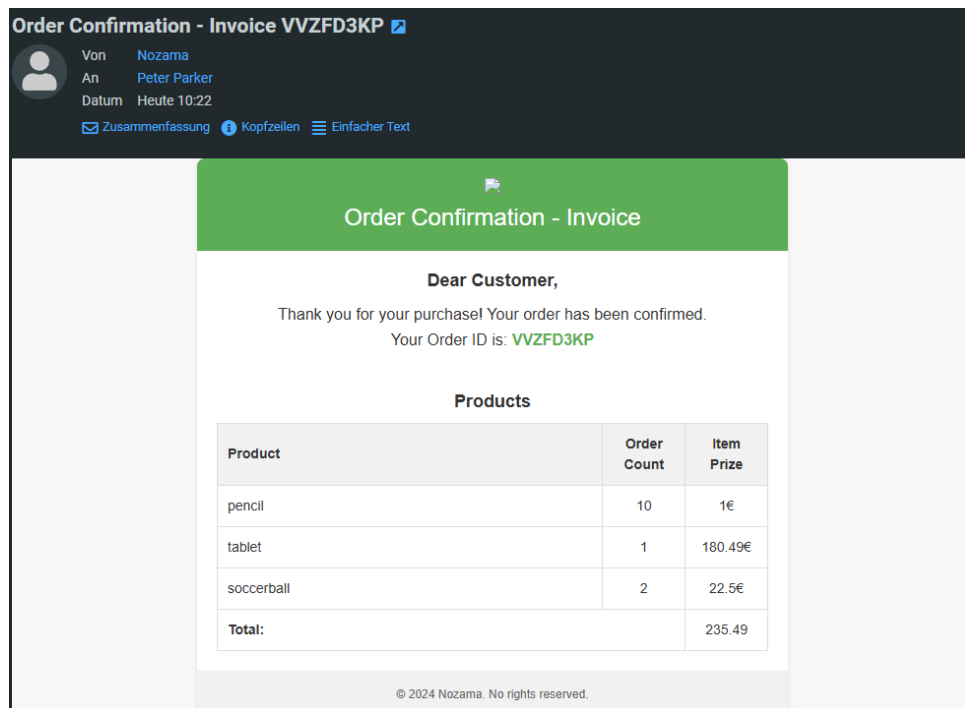


Figure 6: Example of the invoice which the customer receives.

- **Validate payment** is a user task where the user checks the received money and performs the validation of the payment (see Section 2.2.2). At the moment the received\_payment field is filled by the total\_cost since we do not really receive money from outside.

- Input: order\_id, total\_cost, received\_payment
- Output: is\_payment\_valid

- **Handle order**

This multi-instance subprocess takes example\_customer.items list as input and performs the subprocess for each item in the list. See **Refill Stock** (next bullet point).

- Input: example\_customer.items
- Output: (for each) item

- **Refill Stock**

This task can be found inside the subprocess **Handle order**. It takes the item as input and increases the stockCount of the item if it is smaller than the orderCount. We assume that each refill is successful.

- Input: item
- Output: item (but changed)

- **Arrange pickup**

This task can be found inside the subprocess **Handle shipment**. It takes the example\_order.customer as input and sets the values of the linked form (see Section 2.2.2) to these default values. The form values are then used to generate the output.

- Input: example\_order.customer
- Output: date\_of\_delivery, chosen\_service, package\_duration, customer\_name\_new

- **Generate shipment documents**

This task can be found inside the subprocess **Handle shipment**. This script task generates the shipment documents for the mail using the form values, items list and order ID.

- Input: order\_id, example\_order.items, date\_of\_delivery, chosen\_service, example\_order.duration, customer\_name\_new
- Output: generated\_shipment\_documents



- **Update delivery status** decrements the duration by one each time the loop is executed. In our example input the update is done twice. This script task can be found inside the subprocess **Handle shipment**.
  - Input: order\_id, example\_order.items, date\_of\_delivery, chosen\_service, example\_order.duration, customer\_name\_new
  - Output: generated\_shipment\_documents
- **Send delivery status** is a SendGrid outbound connector to send a delivery status via e-mail and can be found inside the subprocess **Handle shipment**. It takes the shipment documents as input and send the mail to the customer.
  - Input: generated\_shipment\_documents
  - Output: E-Mail

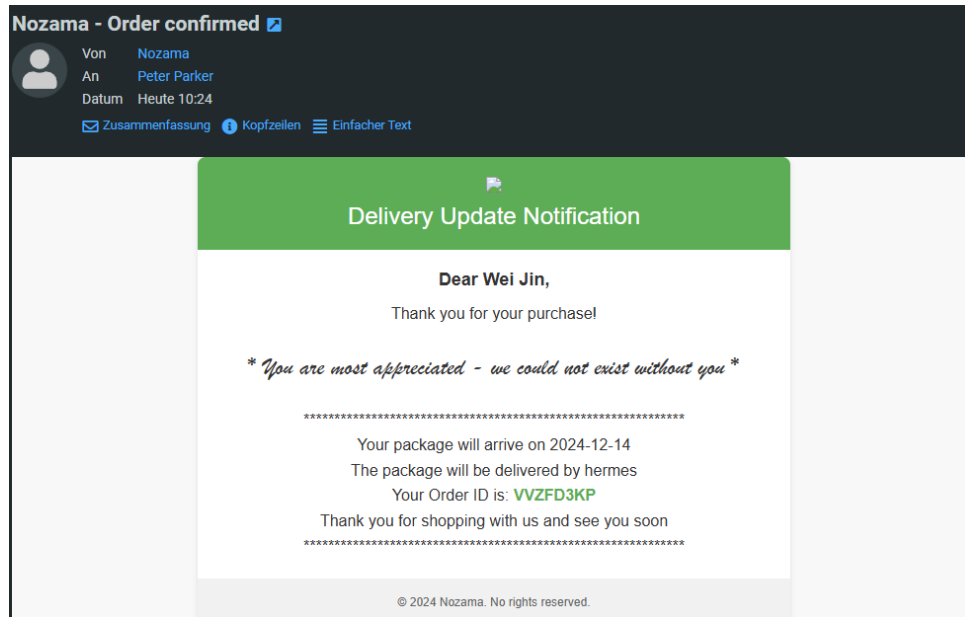


Figure 7: Example of the invoice which the customer receives.

- **Drop off package**  
This user task can be found inside the subprocess **Handle shipment**. It uses the form "Drop-off-package" which is explained in 2.2.2.
  - Input: customer\_name\_new
  - Output: -

### 2.2.2 Model Forms

For the two user tasks inside **Handle shipment** we created two forms which have to be filled. The user task "Arrange pickup" has the "Arrange pickup"-form linked to it, while the second user task "Drop off package" has the "Drop off package"-form linked to it.

- **Arrange pickup form** When the process is simulated, the user has to select a postal service from a drop down list and enter the expected date of delivery. The Name, City, Postal Code, Address and Number have default values from the input data, but the user can still change them. Additionally the user can input additional notes. All fields aside from the notes are required and validated by regular expressions.
- **Drop off package form** The "Drop off package"-form is similar to the "Arrange pickup"-form. When simulated, the user has to enter the Delivery data, while the Name is read from the input data and can't be changed. The user further has the option to click the checkbox, if the package is delivered to a different recipient. If this is the case, new textfields appear, where the user has to put the Name, City, Postal Code, Address and Number. Once again, a Notes textfield is provided, if the user wants to add notes.
- **Check payment received form** The "Check payment received"-form is just for validating that the customer paid the correct requested cost of their order. The user can check via the checkbox whether the two amounts are equal.

## Appendix

The link to the GitHub repository: [github project](#).


The link to the HU Box with the screencast video: [HU Box](#).

Full name*	Delivery date*	
<input type="text"/>	<input type="text" value="dd.mm.yyyy"/>	
<input type="checkbox"/> different recipient		
Full name of recipient	City	Postal Code
<input type="text"/>	<input type="text"/>	<input type="text"/>
Address	Number	
<input type="text"/>	<input type="text"/>	
Note	<input type="text"/>	

Figure 8: Form for the arrange pickup.

Full name\*

Delivery date\*



dd.mm.yyyy

☐ different recipient

Full name of recipient

City

Postal Code

Address

Number

Note

Figure 9: Form for the dropoff package.

ORDER ID\*

Total Cost\*

Received Payment\*

☐ Payment received

Figure 10: Form for the check receive payment.