

Upravljanje poslovnim procesima (UPP)

Nositelj: izv. prof. dr. sc. Darko Etinger

Asistent: Luka Blašković, mag. inf.

Ustanova: Sveučilište Jurja Dobrile u Puli, Fakultet informatike u Puli



Fakultet informatike u Puli

(7) Napredniji zadaci u procesno-orientiranom razvoju

#6

UPP

U ovoj skripti prolazimo kroz servisne i otpremne zadatke, događaje, potprocese te DMN u procesno-orientiranom razvoju poslovnih aplikacija. Dosad smo ove koncepte razmatrali u kontekstu modeliranja poslovnih procesa, a u nastavku ih povezujemo s praktičnom izgradnjom procesno-orientiranih poslovnih aplikacija koristeći Camunda 8 BPM platformu. Pritom nastavljamo razvoj procesne aplikacije za upravljanje narudžbama u trgovini na način da ćemo implementirati servisni zadatak koji će putem REST API-ja komunicirati s našim Express.js poslužiteljem za upravljanje narudžbama, kao i otpremni zadatak koji će slati email obavijesti korisnicima. Također, vidjet ćemo kako koristiti događaje i potprocese za modeliranje složenijih procesa te kako integrirati DMN odluke u procesnu aplikaciju kroz *Business rule taskove*.

Posljednje ažurirano: 25.1.2026.

Sadržaj

- [Upravljanje poslovnim procesima \(UPP\)](#)
- [\(7\) Napredniji zadaci u procesno-orientiranom razvoju](#)
 - [Sadržaj](#)
- [1. Servisni zadaci u procesnoj aplikaciji](#)
 - [1.1 Priprema Express.js REST API poslužitelja](#)
 - [Izmjena "order-confirmation" forme](#)
 - [1.2 Implementacija REST Outbound Connectora za servisne zadatke](#)
 - [Ručno rješavanje incidenata u Camunda Operate aplikaciji](#)
 - [Sprječavanje incidenta error boundary događajem](#)
- [2. Otpremni zadaci u procesnoj aplikaciji](#)
 - [2.1 Email.js konfiguracija](#)
 - [Priprema Email.js predloška](#)

- [2.2 Implementacija Email.js na Express.js poslužitelju](#)
- [2.3 Implementacija REST Outbound Connectora za otpremni zadatak](#)
- [3. Događaji i potprocesi u procesnoj aplikaciji](#)
 - [3.1 Potprocesi u procesnoj aplikaciji](#)
- [4. DMN u procesnoj aplikaciji](#)
- [Zadaci za Vježbu 7](#)

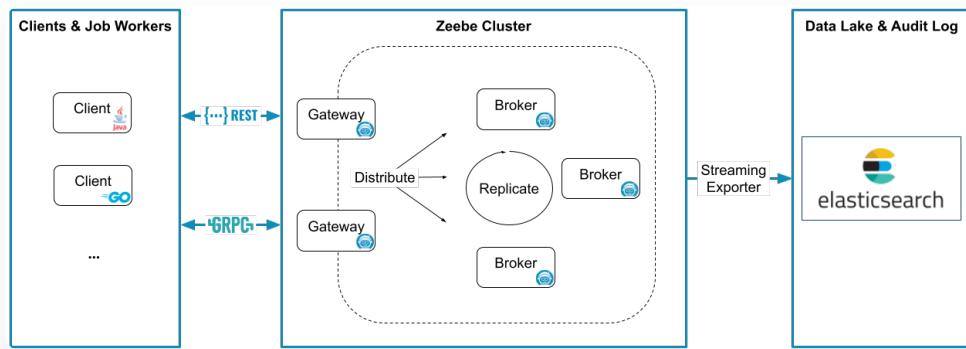
1. Servisni zadaci u procesnoj aplikaciji

U prošloj skripti vidjeli ste kako koristiti servisne zadatke na gotovom primjeru iz Camunda 8 self-managed distribucije. Prošli smo kroz implementaciju gotovih servisnih zadataka koji su bili implementirani u Node.js okruženju te su integrirani u procesnu aplikaciju kroz Camunda 8 SDK za Node.js.

Camunda8 SDK za Node.js omogućava jednostavnu integraciju servisnih zadataka u procesne aplikacije koristeći gRPC protokol za komunikaciju između procesnog *enginea* (Zeebe) i servisnih zadataka. Međutim, vidjeli smo da Zeebe ima i svoj REST API koji omogućava komunikaciju s procesnim *engineom* putem HTTP zahtjeva. Navedeno smo koristili za pokretanje procesne instance izvan Modeler alata.

Osim toga, Zeebe *engine* podržava i ponašanje kao HTTP klijent, što znači da servisni zadaci mogu slati HTTP zahtjeve prema vanjskim sustavima ili API-jima. Ovo je korisno kada želimo integrirati procesnu aplikaciju s drugim sustavima ili uslugama putem RESTful API-ja, a pritom ne želimo koristiti interni gRPC protokol niti Camunda 8 SDK.

[gRPC](#) (eng. gRPC Remote Procedure Calls) je open-source [RPC](#) (eng. Remote Procedure Call) razvojni okvir koji za serijalizaciju podataka koristi Protocol Buffers ([Protobuf](#)), a za prijenos podataka protokol HTTP/2. Omogućuje učinkovitu i skalabilnu komunikaciju između raspodijeljenih sustava te podržava rad u različitim programskim jezicima i na različitim platformama.



Slika 1. Arhitektura Camunda 8 Zeebe procesnog *enginea* (izvor: [Camunda Documentation](#))

Obzirom da se mi na kolegiju web aplikacije bavimo izradom Node.js poslužitelja koji komuniciraju putem HTTP protokola, nastojat ćemo povezati gradivo iz ovog poglavlja s našim dosadašnjim znanjem o izradi REST API poslužitelja u Node.js okruženju na način da implementiramo servisne zadatke koji će putem procesnog *enginea* slati HTTP zahtjeve prema našem postojećem Express.js poslužitelju za upravljanje narudžbama.

Na ovaj način, mi ustvari povezujemo dva koncepta koja smo na prošlim vježbama suprotstavili:

1. **Procesno-orientirani razvoj** poslovnih aplikacija (BPM) koristeći Camunda 8 platformu

2. Klasični backend razvoj - REST API poslužitelj koristeći Node.js i Express.js framework.

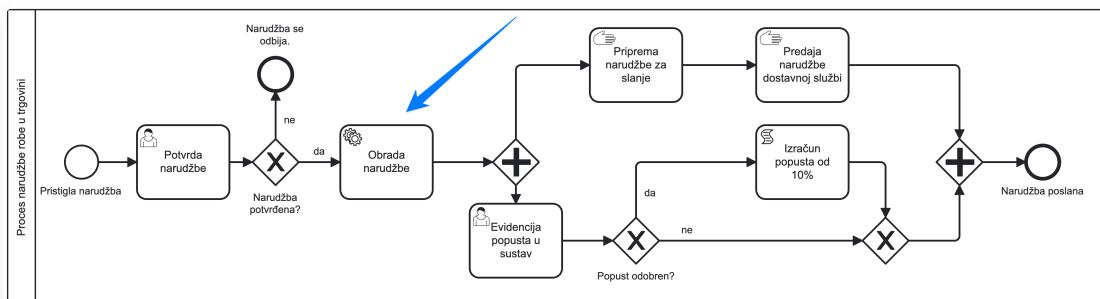
Međutim, poslovnu logiku naše aplikacije sada nastojimo odvojiti u procesnu aplikaciju, dok podatkovni sloj postaje Express.js REST API poslužitelj.

1.1 Priprema Express.js REST API poslužitelja

Implementirat ćemo jednostavan Express.js poslužitelj koji će obraditi dolazne narudžbe na način da ih pohrani u memoriju poslužitelja (za potrebe ove vježbe nećemo koristiti bazu podataka) te izvrši određene operacije nad njima (validacija podataka, izračun ukupne cijene narudžbe i dodavanje statusa i ID-a narudžbi).

Pozivanje ovih funkcija će se vršiti putem servisnog zadataka "Obrada narudžbe" koji će slati HTTP POST zahtjev prema našem Express.js poslužitelju koristeći REST Outbound Connector.

Otvorite Camunda Modeler i sve komponente procesne aplikacije koje smo izradili na prethodnim vježbama. Nakon uspješne potvrde narudžbe, dodajte novi servisni zadatak naziva "Obrada narudžbe" koji će izvršiti spomenute operacije nad podacima narudžbe iz procesne instance.



Slika 2. Servisni zadatak "Obrada narudžbe" dodajemo nakon "da" slijeda iz skretnica "Narudžba potvrđena?".

Prije nego što konfiguriramo [REST Outbound Connector](#), implementirat ćemo jednostavni Express.js poslužitelj koji će obrađivati dolazne narudžbe.

Inicijalizirajte novi Node.js projekt za naš Express.js poslužitelj i biblioteku za validaciju podataka `express-validator`:

```
→ mkdir order-management-api
→ cd order-management-api
→ npm init -y
→ npm install express express-validator
```

Nakon toga, implementirajmo osnovni Express.js poslužitelj u datoteci `index.js` i endpoint za obradu narudžbi `POST /orders`:

```
// order-management-api/index.js
import express from "express";
import { body, validationResult } from "express-validator";

const app = express();
```

```

const PORT = 3000;

app.use(express.json());
let orders = [];

app.post(
  "/orders",
  body("customerName").isString().notEmpty(), // validacija podataka koristeći express-
  validator middleware funkcije
  body("customerEmail").isEmail(),
  body("items").isArray({ min: 1 }),
  body("items.*.name").isString().notEmpty(),
  body("items.*.quantity").isInt({ min: 1 }),
  body("items.*.price").isFloat({ min: 0 }),
  (req, res) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
      return res.status(400).json({ errors: errors.array() }); // u slučaju greške,
  vraćamo složeni objekt sa svim validacijskim greškama
    }

    const { customerName, customerEmail, items } = req.body;
    // izračun ukupnog iznosa narudžbe
    const totalAmount = items.reduce(
      (sum, item) => sum + item.quantity * item.price,
      0,
    );
    const orderId = orders.length + 1;
    const newOrder = {
      id: orderId,
      customerName,
      customerEmail,
      items,
      totalAmount,
      status: "Processed",
    };

    orders.push(newOrder);
    console.log("Nova narudžba obrađena:", newOrder);
    res.status(201).json(newOrder); // vraćamo odgovor s podacima o obrađenoj narudžbi
  },
);

app.listen(PORT, () => {
  console.log(`Order Management API sluša na http://localhost:${PORT}`);
});

```

Definirali smo POST endpoint koji očekuje JSON tijelo s podacima o narudžbi, validira ih te izračunava ukupni iznos narudžbe i dodaje status "Processed" i generira ID na temelju broja prethodnih narudžbi.

Ulagani podaci (podaci koji "dolaze" u procesnu aplikaciju pokretanjem instance) za ovaj endpoint trebaju imati sljedeću JSON strukturu:

```
{
  "customerName": "Ime Kupca",
  "customerEmail": "emailKupca@gmail.com",
  "items": [
    { "name": "Proizvod 1", "quantity": 2, "price": 50.0 },
    { "name": "Proizvod 2", "quantity": 1, "price": 30.0 }
  ]
}
```

curl naredba za testiranje endpointa:

```
→ curl -X POST http://localhost:3000/orders \
-H "Content-Type: application/json" \
-d '{
  "customerName": "John Doe",
  "customerEmail": "johndoe@gmail.com",
  "items": [
    { "name": "laptop", "quantity": 2, "price": 1000.0 },
    { "name": "TV", "quantity": 1, "price": 500.0 }
  ]
}'
```

Endpoint vraća u HTTP odgovoru JSON objekt s podacima o obrađenoj narudžbi, uključujući generirani ID, ukupni iznos i status narudžbe.

```
{
  "id": 1,
  "customerName": "John Doe",
  "customerEmail": "johndoe@gmail.com",
  "items": [
    { "name": "laptop", "quantity": 2, "price": 1000.0 },
    { "name": "TV", "quantity": 1, "price": 500.0 }
  ],
  "totalAmount": 2500.0,
  "status": "Processed"
}
```

Ideja je sljedeća: nakon što korisnik potvrdi narudžbu u procesnoj aplikaciji, servisni zadatak "Obrada narudžbe" će poslati HTTP POST zahtjev prema našem Express.js poslužitelju s podacima o narudžbi iz procesne instance. Poslužitelj će obraditi narudžbu i vratiti odgovor s podacima o obrađenoj narudžbi, koje ćemo zatim moći koristiti u dalnjem tijeku procesne instance (npr. za slanje email obavijesti korisniku).

Izmjena "order-confirmation" forme

Kako bismo ispravno prikazali djelatniku koji obrađuje narudžbu novi podatkovni format koji uključuje stavke narudžbe s atributima `name`, `quantity` i `price`, potrebno je izmijeniti formu `order-confirmation-form.form` u procesnoj aplikaciji.

Otvorite definiranu formu `order-confirmation-form.form` s prošlih vježbi i dodajte još jednu tablicu:

- tablica će nam prikazivati podatke o kupcu** (`customerName`, `customerEmail`)
- tablica će nam prikazivati stavke narudžbe** (`items`)

Dodajte odgovarajuće stupce, te izmijenite `Header items` ključeve kako bi odgovarali atributima stavki narudžbe. Podsjetnik: ključevi su proizvoljnog naziva.

Za upis podataka u **1. tablici**, pišemo sljedeći FEEL izraz:

```
[  
{  
    kupac_table_imeprezime: customerName,  
    kupac_table_email: customerEmail  
}  
]
```

- gdje su `kupac_table_imeprezime` i `kupac_table_email` **nazivi stupaca** u tablici za prikaz podataka o kupcu.
- `customerName` i `customerEmail` su nazivi **procesnih varijabli** koje sadrže podatke o kupcu.

The screenshot shows a 'Potvrda pristigle narudžbe' (Delivery confirmation) form on the left and its configuration panel on the right.

Form (Left):

- Section 'Podaci o kupcu':** Contains fields for 'Customer name' and 'Customer email'. A blue arrow points from the configuration panel's 'Header items' section to this section.
- Section 'Stavke narudžbe':** Contains columns for 'Naziv stavke', 'Jedinična cijena', and 'Naručena količina'.
- Text at the bottom:** 'Potvrđujem ispravnost narudžbe' (I confirm the order is correct) with a checkbox, and a note: 'Molimo vas da odaberete opciju iznad ako ste pregledali pristiglu narudžbu i potvrđujete ju za daljnju obradu.'

Configuration Panel (Right):

- General:** Table label: 'Podaci o kupcu'.
- Data source:** FEEL expression: `= [{ kupac_table_imeprezime: customerName, kupac_table_email: customerEmail }]`. A blue arrow points from this field to the 'Customer name' field in the 'Podaci o kupcu' section of the form.
- Pagination:** Number of rows per page: 10.
- Headers source:**
 - Header items:** Two entries:
 - Customer name: Label 'Customer name', Key 'kupac_table_imeprezime'
 - Customer email: Label 'Customer email', Key 'kupac_table_email'

Slika 3. Prikaz podataka o kupcu u formi za potvrdu narudžbe.

Dodajemo i **2. tablicu** za prikaz stavki narudžbe s FEEL izrazom. Možemo koristiti petlju `for item in items` kako bismo iterirali kroz sve stavke narudžbe iz procesne varijable `items`:

Form Definition

Potvrda pristigle narudžbe

Podaci o kupcu

| | |
|--------------------------|---------------------------|
| Customer name Content | Customer email Content |
|--------------------------|---------------------------|

Stavke narudžbe

| Naziv stavke | Jedinična cijena | Naručena količina |
|--------------|------------------|-------------------|
| Content | Content | Content |

Potvrđujem ispravnost narudžbe
Molimo vas da odaberete opciju iznad ako ste pregledali pristiglu narudžbu i potvrđujete ju za daljnju obradu.

FEEL izraz za iteriranje stavki narudžbi iz procesne varijable "items"

```
= for item in items
    return {
        stavke_table_naziv: item.name,
        stavke_table_kolicina: item.quantity,
        stavke_table_jed_cijena : item.price
    }
```

Form Definition

General

Data source **fx**

Specify the source from which to populate the table

Pagination

Number of rows per page: 10

Headers source

Type: List of items

Header Items

- Naziv stavke

Label: Naziv stavke
Key: stavke_table_naziv
- Jedinična cijena

Label: Jedinična cijena
Key: stavke_table_jed_cijena
- Naručena količina

Label: Naručena količina
Key: stavke_table_kolicina

Slika 4. Prikaz stavki narudžbe u formi za potvrdu narudžbe.

Deployati čemo procesnu definiciju nakon što završimo s izmjenama forme. Nakon toga, započnite instancu kako biste testirali ispravan prikaz podataka u formi za potvrdu narudžbe.

Kako biste izbjegli grešku sa servisnim zadatkom "Obrada narudžbe" (koji još nije konfiguriran), **možete ga privremeno pretvoriti u ručni zadatak** (eng. *manual task*) kako biste uspješno mogli *deployati* procesnu definiciju i započeti instancu procesa.

Otvorite Camunda Tasklist i provjerite ispravnost prikaza podataka o kupcu i stawkama narudžbe u formi:

Potvrda narudžbe
webshop-order-process

Unassigned Assign to me

Task Process

Potvrda pristigle narudžbe

Podaci o kupcu

| | |
|---------------------------|-------------------------------------|
| Customer name John Doe | Customer email johndoe@gmail.com |
|---------------------------|-------------------------------------|

Stavke narudžbe

| Naziv stavke | Jedinična cijena | Naručena količina |
|--------------|------------------|-------------------|
| laptop | 1000 | 2 |
| TV | 500 | 1 |

Potvrđujem ispravnost narudžbe
Molimo vas da odaberete opciju iznad ako ste pregledali pristiglu narudžbu i potvrđujete ju za daljnju obradu.

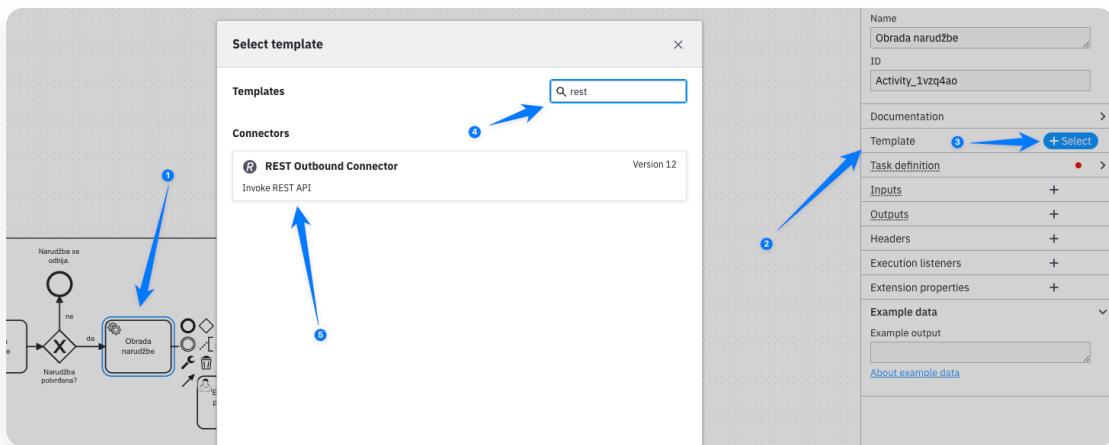
Slika 5. Prikaz forme za potvrdu narudžbe s podacima o kupcu i stawkama narudžbe u Camunda Tasklist aplikaciji.

1.2 Implementacija REST Outbound Connector-a za servisne zadatke

Sljedeći korak je implementacija **REST Outbound Connector-a** u servisnom zadatku "Obrada narudžbe" kako bismo poslali HTTP POST zahtjev prema našem Express.js poslužitelju.

Vratite se u Camunda Modeler i otvorite postavke servisnog zadatka "Obrada narudžbe".

U postavkama odaberite `Template -> Select`, a potom u izborniku pretražite pojam "rest" te odaberite **REST Outbound Connector**.



Slika 6. Odabir REST Outbound Connector-a u postavkama servisnog zadatka.

Primjetit ćete da je BPMN servisni zadatak promijenio ikonu kako bi označio da koristi REST Outbound Connector. **Mi i dalje koristimo servisni zadatak**, samo je njegova implementacija sada definirana putem REST Outbound Connector-a pa je to naznačeno i ikonografijom.

REST Outbound Connector predstavlja gotovu konfiguraciju servisnog zadatka koja nam omogućava slanje HTTP zahtjeva prema vanjskim sustavima ili API-jima bez potrebe za dodatnim kodiranjem.

Ako otvorite postavke, vidjet ćete uobičajene parametre za konfiguraciju REST zahtjeva:

- **Authentication** - autentikacija za REST API (ostavimo na None jer naš Express.js poslužitelj ne koristi autentikaciju; primjer: ako bismo slali JWT token, odabiremo ovdje Bearer Token)
- **HTTP endpoint** - odabiremo `Method`, `URL`, `Headers`, `Query Parameters`
- **Connection timeout** - vrijeme čekanja na odgovor od REST API-ja (našeg Express.js poslužitelja)
- **Payload** - tijelo HTTP zahtjeva (za POST, PUT, PATCH metode)

Osim toga, vidimo i **Error handling** te **Output mapping** sekcije koje možemo koristiti za rukovanje greškama i mapiranje odgovora iz REST API-ja na procesne varijable, slično kao što smo radili sa *script taskom* na prošlim vježbama.

Kako bismo poslali `POST /orders` zahtjev prema našem Express.js poslužitelju, konfiguriramo sljedeće parametre:

- **Authorization:** None
- **HTTP endpoint:**
 - **Method:** `POST`

- URL: `http://localhost:3000/orders`
- Headers: `{"Content-Type": "application/json", "Accept": "application/json"}`
- Query Parameters: ostavimo prazno
- **Connection timeout:** ostavimo zadane vrijednosti
- **Payload** navodimo istu JSON strukturu kao iz Postmana, ali sada koristimo procesne varijable kao vrijednosti:

```
= {
  "customerName": customerName,
  "customerEmail": customerEmail,
  "items": items
}
```

- možemo odabrati i `Ignore null values` checkbox kako bismo izbjegli slanje atributa s `null` vrijednostima (iako u našem slučaju to nije potrebno jer su sve varijable definirane).

To je to! *Deployajte* procesnu definiciju i započnite novu instancu procesa kako biste testirali ispravnost konfiguracije.

Nakon što potvrdite narudžbu u Camunda Tasklist aplikaciji, servisni zadatak "Obrada narudžbe" će se izvršiti i poslati HTTP POST zahtjev prema našem Express.js poslužitelju. Isto možete provjeriti u konzoli gdje je pokrenut Express.js poslužitelj:

```
Nova narudžba obrađena: {
  id: 3,
  customerName: 'John Doe',
  customerEmail: 'johndoe@gmail.com',
  items: [
    { name: 'laptop', price: 1000, quantity: 2 },
    { name: 'TV', price: 500, quantity: 1 }
  ],
  totalAmount: 2500,
  status: 'Processed'
}
```

Provjerimo sada stanje našeg procesa u Camunda Operate aplikaciji. Vidjet ćemo da je servisni zadatak "Obrada narudžbe" uspješno izvršen.

Ipak, nismo još pohranili **HTTP odgovor** (eng. *HTTP response*) koji je vratio naš Express.js poslužitelj.

Odgovor se pohranjuje u `response` objekt koji je dostupan unutar servisnog zadatka, a sastoji se od atributa:

- `statusCode` - HTTP statusni kod odgovora (npr. 200, 201, 400, 500)
- `headers` - zaglavlja HTTP odgovora
- `body` - tijelo HTTP odgovora (u našem slučaju, JSON objekt s podacima o obrađenoj narudžbi)
- `document` - opcionalno možemo pohraniti cijeli odgovor kao dokument ako odaberemo opciju `Store response`

Odaberite **Output mapping** sekciju u postavkama servisnog zadatka. Možemo navesti sljedeće:

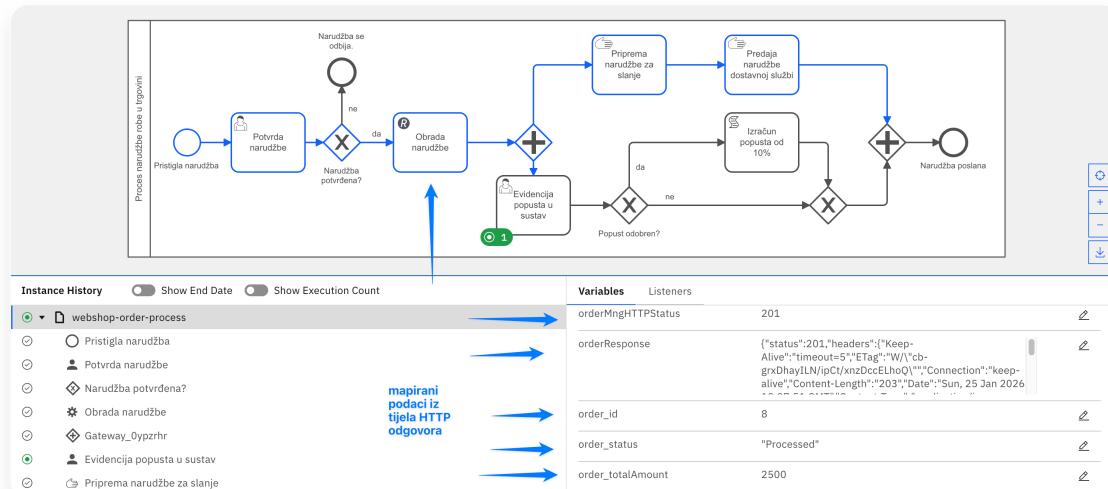
- **Variable name:** `orderResponse` (ili bilo koji drugi naziv procesne varijable gdje pohranjujemo objekt HTTP odgovora)
- **Result expression:** navodimo FEEL izraz koji će mapirati željene atribute iz `response` objekta.

Pohranit ćemo samo dodatne podatke o narudžbi koje je generirao naš poslužitelj: `id`, `totalAmount`, `status` te ćemo pohraniti HTTP statusni kod odgovora kako bismo mogli obraditi validacijsku grešku ako do nje dođe.

```
{  
    order_totalAmount: response.body.totalAmount,  
    order_status: response.body.status,  
    order_id: response.body.id,  
    orderMngHttpStatus: response.status  
}
```

- gdje su `order_totalAmount`, `order_status`, `order_id` i `orderMngHttpStatus` nazivi procesnih varijabli koje ćemo koristiti u dalnjem tijeku procesa.

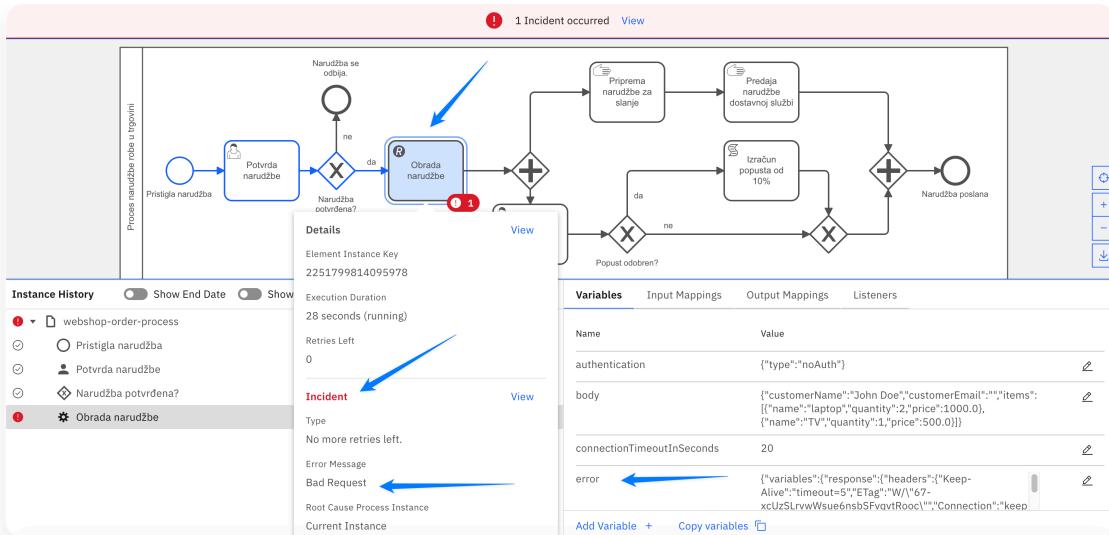
Deployajte procesnu definiciju i započnite novu instancu procesa kako biste testirali ispravnost konfiguracije s mapiranjem izlaznih podataka.



Slika 7. Unutar Camunda Operate aplikacije možemo vidjeti mapirane podatke iz HTTP odgovora pohranjene u procesne varijable nakon izvršenja servisnog zadatka "Obrada narudžbe".

Ako pošaljemo neispravne podatke (npr. izostavimo email adresu kupca), naš Express.js poslužitelj će vraćati validacijske greške s HTTP statusnim kodom `400`.

Pošaljite jedan takav zahtjev i pogledajte što će se dogoditi u Camunda Operate aplikaciji.



Slika 8. U slučaju greške (npr. nedostaje email adresa kupca), servisni zadatak "Obrada narudžbe" će pasti s greškom jer nismo implementirali rukovanje greškama.

Ručno rješavanje incidenata u Camunda Operate aplikaciji

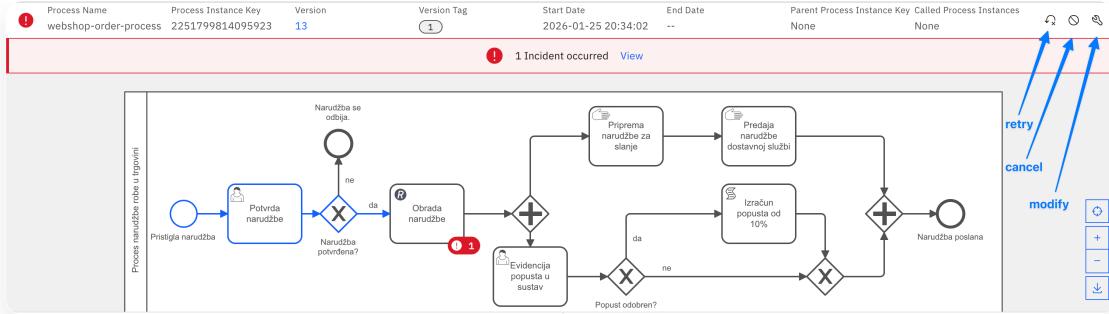
Dobit ćemo grešku odnosno **Incident** tijekom izvođenja naše procesne instance. Ovo se događa zato što nismo implementirali nikakvo rukovanje greškama u slučaju da naš Express.js poslužitelj vratí HTTP statusni kod **400** (Bad Request) ili neki drugi kod koji označava grešku. Samim time, **procesna instanca ovdje ne može nastaviti dalje** jer servisni zadatak nije uspio izvršiti svoju funkciju.

Zamislite sljedeću poslovnu situaciju. Nakon što djelatnik trgovine potvrđuje narudžbu, dođe do greške prilikom obrade narudžbe na poslužitelju, ali ovaj put dođe do greške koju djelatnik ne može riješiti (npr. zbog nedostupnosti baze podataka ili nekog drugog tehničkog problema). Tada bi poslužitelj trebao rezultirati statusnim kodom **500** (Internal Server Error), a procesna instanca bi trebala biti pauzirana dok se greška ne riješi. Jedna od velikih prednosti procesno-orientiranog razvoja je što poslužitelj (Camunda 8 engine) pamti stanje procesa, samim time djelatnik trgovine može **poništiti instancu, ponovno je pokrenuti** (eng. retry) ili pak **preusmjeriti token na neku drugu aktivnost**. Naravno, najbolja situacija bi bila kada nikad ne bi došlo do greške, ili kada bi *flow* automatski preusmjerio, ali u stvarnom svijetu to nije uvijek moguće, a i **vrlo je teško predvidjeti sve moguće scenarije dok se ne dogode**.

Ne moramo ništa implementirati, već direktno incident možemo riješiti kroz Camunda Operate aplikaciju.

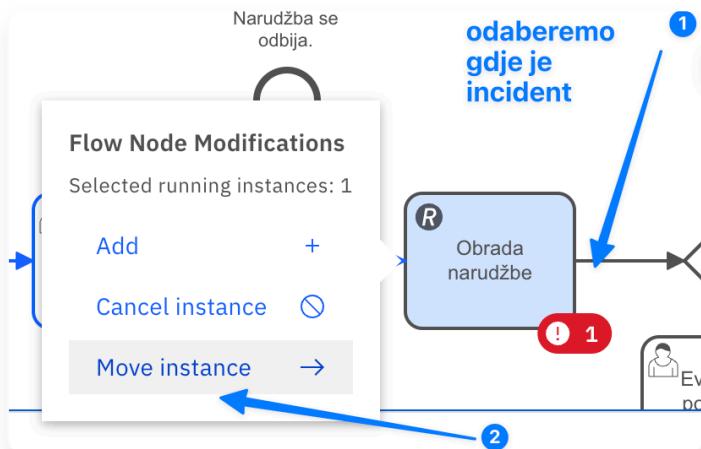
Kada dođe do incidenta, u gornjem desnom kutu imate 3 opcije za ručno upravljanje incidentom:

- **Retry instance:** ponovno pokretanje instance od točke gdje je došlo do greške (pokušava ponovno izvršiti servisni zadatak); ovo može biti korisno ako je greška bila privremena (npr. mrežni problem, nedostupnost poslužitelja)
- **Cancel instance:** poništavanje instance (prekida se daljnje izvršavanje procesa)
- **Modify instance:** nudi niz mogućnosti, uključujući preusmjeravanje toka procesa na drugu aktivnost, dodavanje ili izmjenu procesnih varijabli, itd.

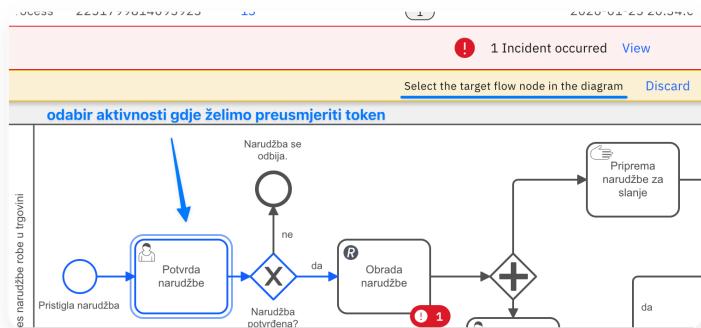


Slika 9. Opcije za upravljanje incidentom u Camunda Operate aplikaciji.

Ako odaberemo **Modify instance**, možemo preusmjeriti tok procesa na neku drugu aktivnost. Primjerice, pokušajmo preusmjeriti tok na zadatku "Potvrda narudžbe" kako bismo "lopticu" prebacili natrag djelatniku trgovine koji sada može odbiti narudžbu i time završiti proces.



Slika 10. Unutar Camunda Operate aplikacije odaberite Modify Instance, a potom Move instance da biste preusmjerili tok procesa na drugu aktivnost.



Slika 11. Odaberite aktivnost "Potvrda narudžbe" kao novu točku na koju želite preusmjeriti tok procesa.

Odaberite **Apply Modifications** kako bi se promjene primijenile.

Trebali biste vidjeti da je instanca uspješno preusmjerena na zadatku "Potvrda narudžbe", a incident je riješen. Međutim, **Camunda pamti da je došlo do incidenta**, pa će se on i dalje prikazivati na aktivnosti "Obrada narudžbe" gdje je incident prvotno nastao.

Procesne varijable također se vraćaju na stanje prije nego što je došlo do incidenta - ovo je vrlo važno kako bi se osiguralo konzistentno stanje procesa.

Sprječavanje incidenta *error boundary* događajem

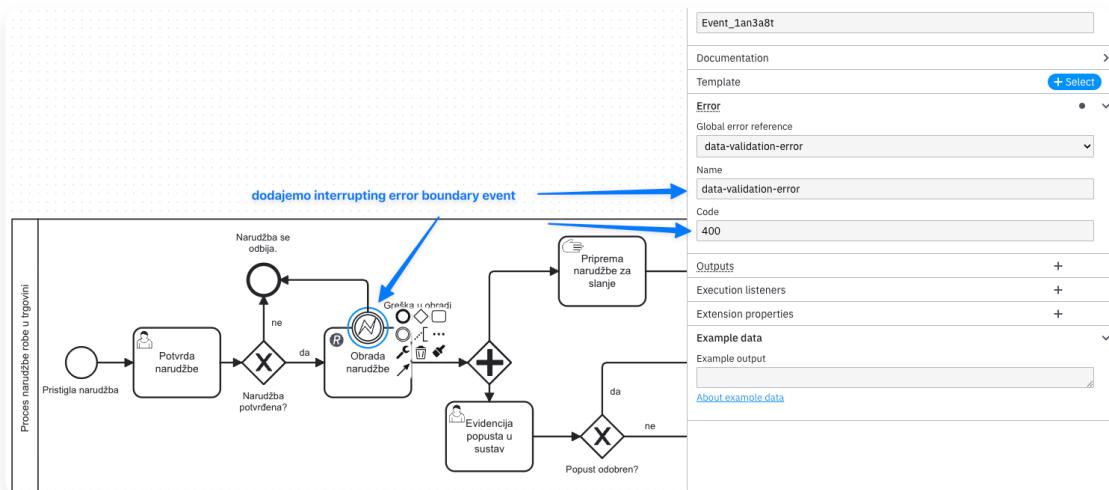
Kako bismo spriječili da dođe do incidenta u slučaju greške prilikom obrade narudžbe, možemo koristiti *interrupting error boundary event* priključen na servisni zadatak "Obrada narudžbe". Na ovaj način, ako dođe do greške (npr. HTTP statusni kod 400 ili 500), tok procesa će se preusmjeriti na definiranu aktivnost unutar *boundary eventa* umjesto da se generira incident.

Mi ćemo završiti s procesnom instancom u slučaju greške.

Kako bismo ovo implementirali, vratite se u Camunda Modeler i dodajte ***interrupting error boundary event*** na servisni zadatak "Obrada narudžbe".

Pojedini boundary event može "uhvatiti" više različitih tipova grešaka. Primjerice, možemo uhvatiti greške vezane uz HTTP statusni kod 400 (Bad Request).

Pod **Errors** dodajte globalnu referencu greške (na taj način da možemo koristiti istu grešku na više mesta u procesu ako je potrebno). Nazovimo grešku `data-validation-error` s kodom 400.



Slika 12. Konfiguracija *interrupting error boundary eventa* za hvatanje greške s kodom 400.

Nakon toga, moramo u postavkama servisnog zadatka "Obrada narudžbe" definirati kada će se ova greška aktivirati. Otvorite postavke servisnog zadatka i idite na **Error handling** sekciju.

Moramo unijeti FEEL izraz za **Error expression**. Definirat ćemo izraz koji će provjeriti je li HTTP statusni kod odgovora jednak 400, ako je će aktivirati *boundary event* funkcijom `bpmnError`.

Sintaksa:

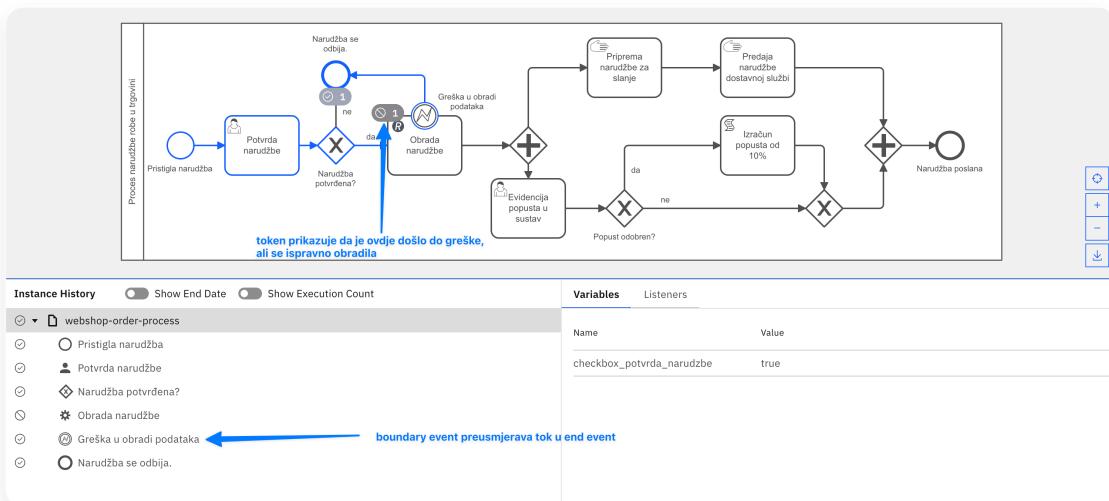
```
if response.status = 400 then bpmnError("statusni_kod", "poruka") else null
```

Naš izraz:

```
if error.code = "400" then
    bpmnError("400", "Greška u obradi narudžbe na poslužitelju")
else null
```

- gdje `error.code` predstavlja HTTP statusni kod iz odgovora REST Outbound Connector-a.
- `bpmnError("400", "Greška u obradi narudžbe na poslužitelju")` aktivira *boundary event* s kodom greške 400 i porukom.

Deployajte procesnu definiciju i započnite novu instancu procesa bez podataka i odobrite narudžbu. Unutar Operate aplikacije vidjet ćete da je instanca završila budući da se tok procesa preusmjerio na završni događaj preko boundary eventa **umjesto da je došlo do incidenta**.



Slika 13. U Camunda Operate aplikaciji vidimo da je procesna instanca završila bez incidenta zahvaljujući implementaciji *interrupting error boundary eventa*.

Više o event handlingu za servisne zadatke možete pročitati [ovdje](#) i [ovdje](#).

2. Otpremni zadaci u procesnoj aplikaciji

Otpremni zadaci (*eng. send tasks*) su specijalizirani tip servisnih zadataka koji se koriste za slanje poruka ili obavijesti iz procesne aplikacije prema vanjskim sustavima ili korisnicima. U kontekstu naše procesne aplikacije za upravljanje narudžbama, **implementirat ćemo otpremni zadatak koji će slati email obavijesti korisnicima nakon što je njihova narudžba obrađena i spremna za isporuku.**

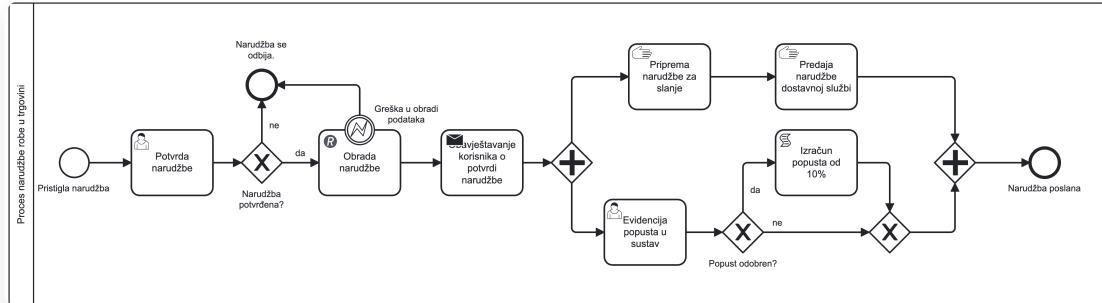
Kao i kod servisnih zadataka, možemo koristiti Camunda 8 SDK za Node.js za implementaciju otpremnih zadataka, koristiti gotove predloške te spojiti se na vanjske email servise poput *Sendgrida*, slanje obavijesti na *Slack* kanal, slanjem poruke na *RabbitMQ*, *Microsoft Teams*, itd. Mogućnosti su neograničene.

Kako nemamo [SMTP server](#) (*eng. Simple Mail Transfer Protocol*) za slanje emailova, iskoristit ćemo popularni Node.js paket **Email.js** koji omogućava slanje emailova putem različitih email servisa (Gmail, Outlook, Yahoo, itd.) bez potrebe za vlastitim SMTP serverom.

Ako imate neki SMTP server, ili želite koristiti drugi servis, možete slobodno prilagoditi implementaciju prema vašim potrebama.

Sve što trebamo je podesiti Email.js s našim email računom, implementirati ga na našem Express.js poslužitelju, te koristiti REST Outbound Connector u otpremnom zadatku za slanje email obavijesti korisnicima.

Dodat ćemo novi *send task* naziva "Obavještavanje korisnika o potvrdi narudžbe" nakon servisnog zadatka "Obrada narudžbe" u našem BPMN modelu.



Slika 14. Dodavanje otpremnog zadatka "Obavještavanje korisnika o potvrdi narudžbe" nakon servisnog zadatka "Obrada narudžbe".

U sljedećoj sekciji ćemo implementirati Email.js na našem Express.js poslužitelju kako bismo mogli slati email obavijesti korisnicima.

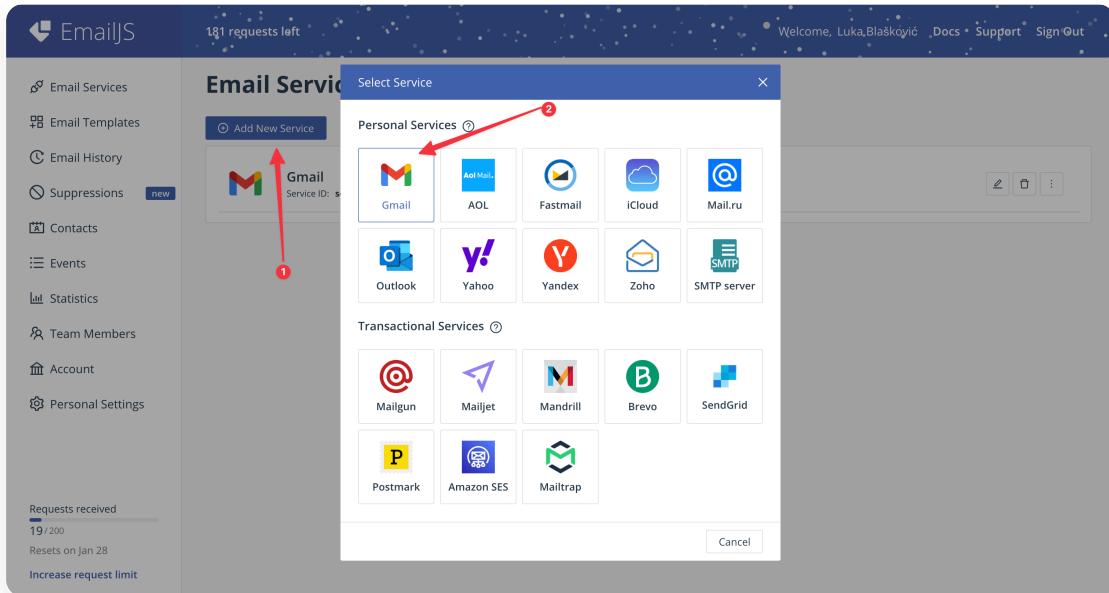
2.1 Email.js konfiguracija

Email.js je popularni servis za slanje emailova direktno iz JavaScript aplikacija bez potrebe za vlastitim SMTP serverom. Pritom ga je moguće koristiti i na klijentskoj (npr. Vue.js) i na serverskoj strani (Node.js).

Izradite novi Email.js račun na [Emailjs.com](#) i slijedite upute za postavljanje servisa za slanje emailova (npr. *Gmail*, *Outlook*, *Yahoo*, itd.).

Napomena: Možete slobodno koristiti i studentski *unipu* račun za ovu vježbu koristeći **Gmail servis**.

Odaberite `Add New Service` i slijedite upute za autorizaciju vašeg email računa. Odabirom servisa, vi ustvari konfigurirate Email.js da koristi taj servis (vaš email račun) za slanje emailova.

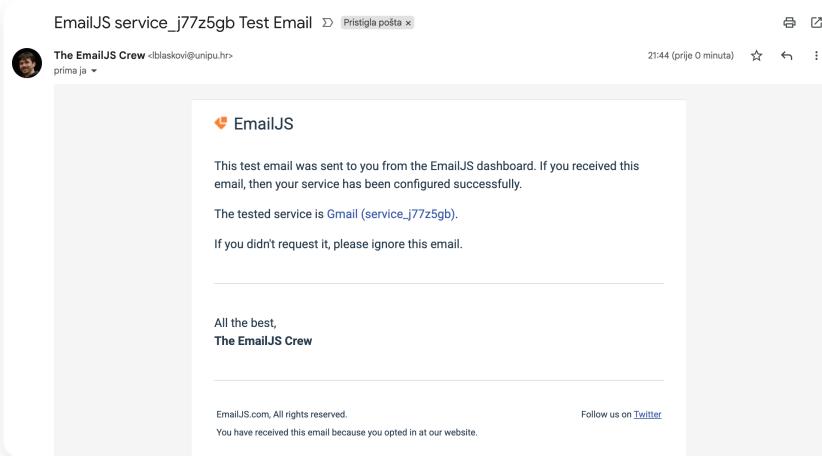


Slika 15. Odabir Gmail servisa u Email.js sučelju.

Ovo nije najbolje produkcijsko rješenje - u pravilu želite izbjegavati korištenje osobnih email računa za slanje emailova iz poslovnih aplikacija. Bolje je koristiti namjenske email servise poput *Sendgrid*, *Mailgun*, *Amazon SES*, itd. koji su dizajnirani za slanje velikog broja emailova i imaju bolje performanse i pouzdanost. Ipak, za male aplikacije ili potrebe testiranja i učenja, korištenje osobnog email računa je u redu.

Odaberite **Connect account** i autorizirajte vaš email račun. Pripazite da **omogućite Email.js aplikaciji slanje emailova** u vaše ime prilikom autorizacije. Nakon vježbe, ovo možete jednostavno opozvati brisanjem servisa u Email.js sučelju.

Ako je autorizacija uspješna, vidjet ćete vaš servis na Email.js nadzornoj ploči i **dobit ćete email potvrde u vaš sandučić**.

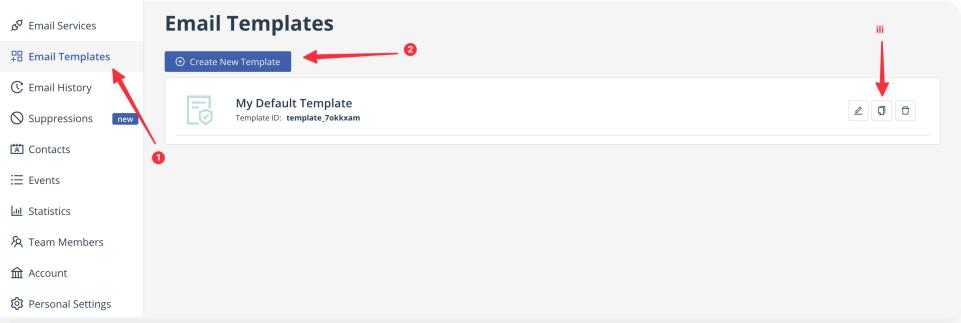


Slika 16. Primjer email potvrde koju šalje Email.js nakon uspješne autorizacije email računa.

Priprema Email.js predloška

Prije nego krenemo s pisanjem koda, izradit ćemo novi **Email.js predložak** koji će se koristiti za slanje email obavijesti korisnicima. Predložak možemo raditi kroz Email.js sučelje.

Odaberite **Email Templates** u lijevom izborniku, a potom **Create New Template**.



Slika 17. Izrada novog Email.js predloška.

Odaberite `Order Confirmation` kao vrstu predloška. Ako hoćete, možete i ručno definirati predložak koristeći HTML i CSS, ali za ovu vježbu iskoristit ćemo gotovi predložak.

Uredite predložak prema vašim preferencijama. Možete prilagoditi boje, fontove, slike, itd. Predložak je moguće urediti za **Desktop** i **Mobile** prikaz. Ako odaberete `Edit Content`, možete birati između **Email.js Design uređivača** ili ručnog uređivanja HTML/CSS koda.

Slika 18. Uređivanje Email.js predloška za potvrdu narudžbe.

Uočit ćete *placeholder* za varijable poput `{{order_id}}`, `{{orders}}`, `{{price}}`, itd. Ideja je da ove podatke popunimo dinamički prilikom slanja emaila iz našeg Express.js poslužitelja. Podatke na poslužitelj poslat ćemo iz procesne aplikacije putem REST Outbound Connectora u otpremnom zadatku.

U Email.js, *placeholderi* za varijable definiraju se dvostrukim vitičastim zagradama `{}{naziv_varijable}{}{}`.

Napravit ćemo sljedeće izmjene u predlošku:

- Promijenit ćemo sliku u zaglavju na neku prikladniju - stavit ćemo logotip našeg Fakulteta/Sveučilišta.
- Promijenit ćemo `From Name` varijablu (desno) - stavit ćemo `UPP Procesna aplikacija`.
- Otvorite HTML kod predloška i pronađite početak i kraj `orders` sekcije. Zamijenit ćemo taj dio varijablom `orders_html` koja će se generirati na poslužitelju kao HTML tablica s podacima o stavkama narudžbe.

Isječak iz predloška koji treba zamijeniti:

...

```

<div
  style="
    text-align: left;
    font-size: 14px;
    padding-bottom: 4px;
    border-bottom: 2px solid #333;
  ">
  <strong>Order # {{order_id}}</strong>
</div>
{{orders_html}}
<div style="padding: 24px 0">
  <div style="border-top: 2px solid #333"></div>
</div>
...

```

4. Promijenit ćemo nazive sljedećih varijabli:

- `cost.shipping` mijenjamo u `cost_shipping`
- `cost.total` mijenjamo u `cost_total`
- `customer.tax` mijenjamo u `customer_tax`

Gotovi HTML kod predloška možete pronaći u `UPP7/order_confirmation.html` datoteci unutar repozitorija za ovu vježbu.

Jednom kada ste završili predložak, spremite ga i zabilježite njegov **Template ID** koji će nam trebati za slanje emailova iz našeg Express.js poslužitelja.

Template ID možete pronaći na nadzornoj ploči predložaka pod `settings`. Primjer: `template_vj22ava`.

Dodatno, možete testirati ispravnost predloška tako da odaberete `Test it` i unesete vašu email adresu i ostale parametre predloška. Email.js će vam poslati testni email koristeći definirani predložak. Ako dobijete `200 OK` poruku, **predložak je ispravan**.

2.2 Implementacija Email.js na Express.js poslužitelju

Otvorite vaš Express.js poslužitelj iz prethodne sekcije i instalirajte `axios` i `dotenv` pakete.

Kako je Email.js servis namijenjen za korištenje na klijentskoj strani, samo za nju postoji službena [Email.js SDK biblioteka](#). Mi ga koristimo na poslužiteljskoj strani, stoga ćemo mu pristupiti preko HTTP REST API-ja koristeći `axios` za slanje HTTP zahtjeva.

Da dobro ste čuli, slati ćemo HTTP zahtjev s poslužitelja. 

```
→ npm install axios dotenv
```

Za to nam je potrebno nekoliko sigurnosnih podataka (API par ključeva, Service ID, Template ID) koje ćemo pohraniti u `.env` datoteku kako bismo ih mogli koristiti unutar našeg Express.js poslužitelja.

Izradite `.env` datoteku u korijenskom direktoriju vašeg Express.js projekta s sljedećim sadržajem:

```

SERVICE_ID= service_xxxxxxx (kopirati iz Email Services, odabir Gmail servisa)
TEMPLATE_ID= template_xxxxxxx (kopirati iz Email Templates/Settings, odabir predloška za
potvrdu narudžbe)
PUBLIC_KEY=Public Key (kopirati iz postavka Email.js servisa - Account - General)
PRIVATE_KEY=Private key (kopirati iz postavka Email.js servisa - Account - General)

```

- `SERVICE_ID` - ID servisa koji smo konfigurirali u Email.js (kopirajte iz Email Services nadzorne ploče)
- `TEMPLATE_ID` - ID predloška koji smo izradili za potvrdu narudžbe (kopirajte iz Email Templates/Settings nadzorne ploče)
- `PUBLIC_KEY` - javni ključ za autentikaciju prema Email.js API-ju (kopirajte iz postavki Email.js servisa - Account - General)
- `PRIVATE_KEY` - privatni ključ za autentikaciju prema Email.js API-ju (kopirajte iz postavki Email.js servisa - Account - General)

Jednom kad ste unijeli sve potrebne varijable okruženja, možemo implementirati novi endpoint na našem Express.js poslužitelju za slanje email obavijesti korisnicima.

Implementirat ćemo endpoint `POST /order-confirmation-email` koji će primati podatke o narudžbi i slati email obavijest korisniku koristeći Email.js REST API.

Ukupna implementacija endpointa može izgledati ovako:

- nemojte zaboraviti uključiti `dotenv` i `axios` na početku `index.js` datoteke

```

// order-management-api/index.js
// random slike proizvoda
let product_images = [
  {
    name: "laptop",
    url: "https://png.pngtree.com/png-vector/20250304/ourmid/pngtree-sleek-modern-laptop-
with-high-resolution-display-png-image_15711292.png",
  },
  {
    name: "TV",
    url: "https://static.vecteezy.com/system/resources/thumbnails/038/015/883/small/ai-
generated-modern-tv-isolated-on-transparent-background-free-png.png",
  },
];

app.post(
  "/order-confirmation-email",
  body("customerEmail").isEmail(),
  body("orderId").isInt({ min: 1 }),
  body("customerName").optional().isString(),
  body("totalAmount").optional().isFloat({ min: 0 }),
  body("status").optional().isString(),
  body("items").isArray({ min: 1 }),
  body("items.*.name").isString().notEmpty(),
  body("items.*.quantity").isInt({ min: 1 }),
  body("items.*.price").isFloat({ min: 0 }),

```

```

async (req, res) => {
  const errors = validationResult(req);
  if (!errors.isEmpty()) {
    return res.status(400).json({ errors: errors.array() });
  }

  const { customerEmail, orderId, items } = req.body;

  // Mapiramo slike proizvoda na stavke narudžbe
  const orders = items.map((item) => {
    const product = product_images.find(
      (p) => p.name.toLowerCase() === item.name.toLowerCase(),
    );
    return {
      name: item.name,
      units: item.quantity,
      price: (item.price * item.quantity).toFixed(2),
      image_url: product ? product.url : "",
    };
  });

  // Generiranje HTML sadržaja za narudžbe (ovo se primjenjuje u predložak na
  {{orders_html}} )
  const ordersHtml = orders
    .map(
      (order) => `
        <table style="width: 100%; border-collapse: collapse">
          <tr style="vertical-align: top">
            <td style="padding: 24px 8px 0 4px; display: inline-block; width: max-content">
              
            </td>
            <td style="padding: 24px 8px 0 8px; width: 100%">
              <div>${order.name}</div>
              <div style="font-size: 14px; color: #888; padding-top: 4px">QTY:<br/>
                ${order.units}</div>
            </td>
            <td style="padding: 24px 4px 0 0; white-space: nowrap">
              <strong>${order.price}</strong>
            </td>
          </tr>
        </table>
      `,
    )
    .join(" ");

  // Dummy izračun troškova dostave i poreza
  const subtotal = items.reduce(
    (sum, item) => sum + item.quantity * item.price,
    0,
  );
  const shipping = 10.0;
  const tax = subtotal * 0.1;
}

```

```

const total = subtotal + shipping + tax;

const emailData = {
  service_id: process.env.SERVICE_ID,
  template_id: process.env.TEMPLATE_ID,
  user_id: process.env.PUBLIC_KEY,
  accessToken: process.env.PRIVATE_KEY,
  template_params: {
    order_id: orderId,
    orders_html: ordersHtml,
    cost_shipping: shipping.toFixed(2),
    cost_tax: tax.toFixed(2),
    cost_total: total.toFixed(2),
    email: customerEmail,
  },
};

try {
  const response = await axios.post(
    "https://api.emailjs.com/api/v1.0/email/send",
    emailData,
    {
      headers: {
        "Content-Type": "application/json",
      },
    },
  );

  console.log("Email poslan:", response.data);
  res.status(200).json({ message: "Email poslan uspješno." });
} catch (error) {
  console.error("Greška pri slanju emaila:", error.response.data);
  res.status(500).json({ error: "Došlo je do greške pri slanju emaila." });
}
},
);

```

Ukratko što je implementirano u ovom kodu:

1. Validiramo ulazne podatke koristeći `express-validator`.
2. Mapiramo slike proizvoda na stavke narudžbe.
3. Generiramo HTML sadržaj za stavke narudžbe koji će se koristiti u predlošku.
4. Pripremamo podatke za slanje emaila koristeći Email.js REST API.
5. Šaljemo HTTP POST zahtjev prema Email.js API-ju koristeći `axios`.
6. Vraćamo odgovarajući HTTP odgovor ovisno o ishodu slanja emaila.

Prije nego implementiramo otpremni zadatak u procesnoj aplikaciji, testirajmo ispravnost novog endpointa koristeći Postman.

The screenshot shows the Postman interface with the following details:

- URL:** https://webshop_process_app / new email
- Method:** POST
- Path:** {{ADDRESS_ORDER_MNG}} /order-confirmation-email
- Body:** JSON (selected)


```

1
2   "customerEmail": "luka.blaskovic@unipu.hr",
3   "orderId": 4,
4   "customerName": "John Doe",
5   "items": [
6     {
7       "name": "laptop",
8       "quantity": 2,
9       "price": 1000
10    },
11    {
12      "name": "TV",
13      "quantity": 1,
14      "price": 500
15    }
16  ],
17  "totalAmount": 2500,
18  "status": "Processed"
19
      
```
- Response Status:** 200 OK
- Response Time:** 1.34 s
- Response Size:** 272 B
- Response Body:** JSON

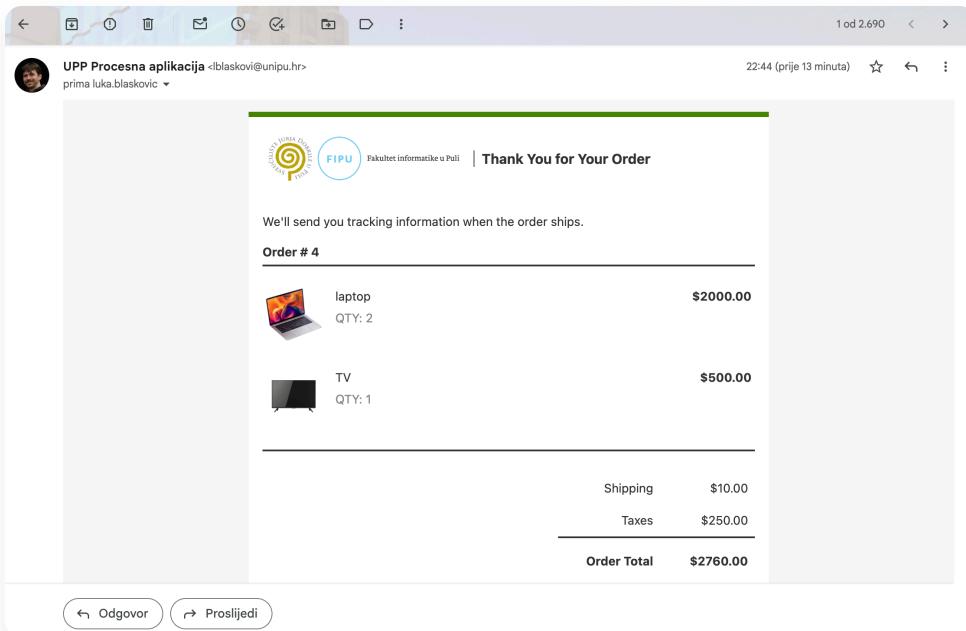

```

1 {
2   "message": "Email poslan uspješno."
3 }
```

Slika 19. Testiranje slanja emaila putem Postmana koristeći novi endpoint `POST /order-confirmation-email`.

Ako nema grešaka, trebali biste dobiti `200 OK` odgovor, a email bi trebao stići u sandučić korisnika koji ste naveli pod `customerEmail`.

Možete staviti svoj vlastiti email kako biste testirali i na taj način sami sebi poslali potvrdu narudžbe.



Slika 20. Primjer email obavijesti o potvrdi narudžbe primljene u Gmail sandučiću.

2.3 Implementacija REST Outbound Connectora za otpremni zadatak

Sada kada imamo funkcionalan endpoint za slanje email obavijesti, možemo implementirati otpremni zadatak u našoj procesnoj aplikaciji koristeći REST Outbound Connector.

Odabrat ćemo **REST Outbound Connector** kao implementaciju otpremnog zadataka "Obavještavanje korisnika o potvrdi narudžbe". Navedeno možemo implementirati na isti način kao i servisni zadatak u prethodnoj sekciji.

Konfigurirajmo sljedeće parametre:

- **Authorization:** None
- **HTTP endpoint:**
 - Method: `POST`
 - URL: `http://localhost:3000/order-confirmation-email`
 - Headers: `{"Content-Type": "application/json", "Accept": "application/json"}`
 - Query Parameters: ostavimo prazno
- **Connection timeout:** ostavimo zadane vrijednosti
- **Payload** navodimo istu JSON strukturu kao iz Postmana, ali sada koristimo ispravne procesne varijable kao vrijednosti:

```
= {  
  "customerEmail": customerEmail,  
  "orderId": order_id,  
  "customerName": customerName,  
  "totalAmount": order_totalAmount,  
  "status": order_status,  
  "items": items  
}
```

Pripazite da koristite **ispravne procesnih varijabli** koje smo definirali i mapirali u prethodnom servisnom zadataku - u suprotnom, otpremni zadatak neće imati ispravne podatke za slanje emaila.

Također, **pripazite da koristite ispravne nazive ključeva** koji se očekuje na Express.js poslužitelju.

The screenshot shows a BPMN process diagram and its corresponding REST connector configuration.

BPMN Process Diagram:

- Starts with a task "Obavještavanje korisnika o potvrdi narudžbe".
- Then "Prepreka narudžbu za slanje".
- Then "Predaja narudžbe dostavničkoj službi".
- Decision diamond "Izračun popusta od 10%": If "da" (yes), it goes to "Izračun popusta od 10%" and then "Narudžba poslana". If "ne" (no), it goes to "Popust odobren?" and then "Narudžba poslana".
- Task "Evidencija popusta u sustav".
- Decision diamond "Popust odobren?": If "da", it goes to "Narudžba poslana". If "ne", it goes back to "Izračun popusta od 10%".
- End event.

REST Outbound Connector Configuration:

- Headers:** Map of HTTP headers to add to the request. Headers = {"Content-Type": "application/json", "Accept": "application/json"}
- Query parameters:** Map of query parameters to add to the request URL. Query parameters =
- Connection timeout:** Connection timeout in seconds. Value: 20
- Payload:** Request body = {
 "customerEmail": customerEmail,
 "orderId": orderId,
 "customerName": customerName,
 "totalAmount": totalAmount,
 "status": orderStatus,
 "items": items
 }
- Output mapping:** Result variable: emailConfirmationResponse. Result expression = {
 "emailConfirmationHttpStatus": response.status
 }

A blue arrow points from the "Popust odobren?" decision diamond to the "Ignore null values" checkbox in the payload configuration.

Slika 21. Postavke *Send Taska* s REST Outbound Connectorom za slanje email obavijesti putem Express.js poslužitelja.

Deployajte procesnu definiciju i započnite novu instancu procesa kako biste testirali ispravnost konfiguracije otpremnog zadatka.

Ako ste sve napravili ispravno, nakon što potvrdite narudžbu u Camunda Tasklist aplikaciji, servisni zadatak "Obrada narudžbe" će se izvršiti, a potom i otpremni zadatak "Obavještavanje korisnika o potvrdi narudžbe" koji će poslati email obavijest korisniku. **Sve možete pratiti i u konzoli gdje je pokrenut Express.js poslužitelj.**

The screenshot shows the Camunda Operate application interface with a running process instance.

Process Details:

- Process name: workshop-order-process
- Instance ID: 1
- Start date: 2023-09-18T10:00:00Z
- Last update date: 2023-09-18T10:00:00Z

Variables:

| Name | Value |
|-----------------------------|---|
| checkbox_potvrda_narudzbe | true |
| customerEmail | "luka.blaskovic@unipu.hr" |
| customerName | "Luka Blaskovic" |
| emailConfirmationHttpStatus | 200 |
| emailConfirmationResponse | {"status":200,"headers":{"Keep-Alive":"timeout=5","ETag":"W/125-..."}, "body": "..."} (partial JSON output) |

Listeners:

- None

Slika 22. U Camunda Operate aplikaciji možemo vidjeti da je otpremni zadatak "Obavještavanje korisnika o potvrdi narudžbe" uspješno izvršen.

3. Događaji i potprocesi u procesnoj aplikaciji

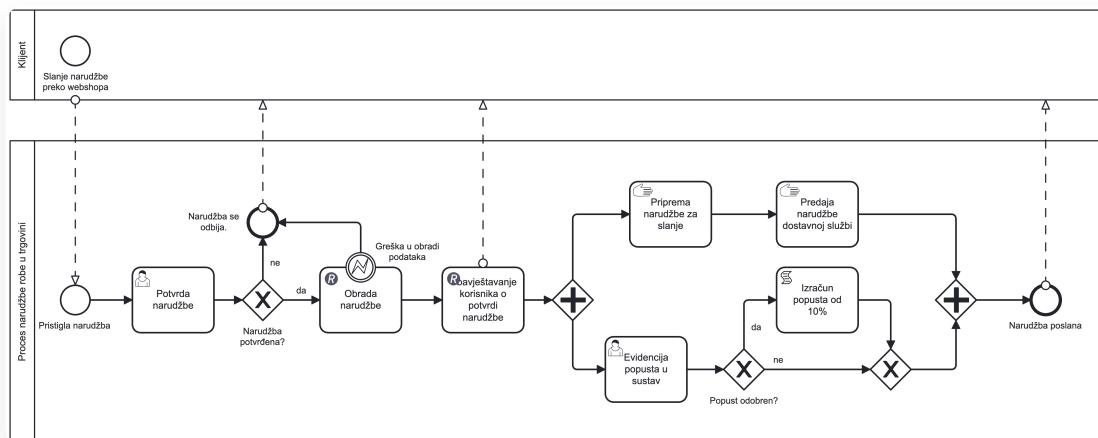
Procesni *engine* Camunda 8 podržava implementaciju gotovo svih BPMN elemenata poslovne logike koji su definirani standardom. Nećemo naravno pokazati sve što smo radili na prethodnim vježbama kada smo učili modelirati, ipak, malo ćemo još "začiniti" našu procesnu aplikaciju dodatnim BPMN (i DMN) elementima.

Primjerice, možemo iskoristiti i one neke elemente modeliranja kojima smo se bavili na prethodnim vježbama, poput **dodavanja apstraktnih polja** i korištenja komunikacije informacijskim tokovima (*eng. message flows*) između različitih procesnih aplikacija.

Također, možemo koristiti **događaje** (*eng. events*) poput *timer eventa* za implementaciju vremenski uvjetovanih aktivnosti, ili *message eventa* za implementaciju komunikacije između različitih procesa.

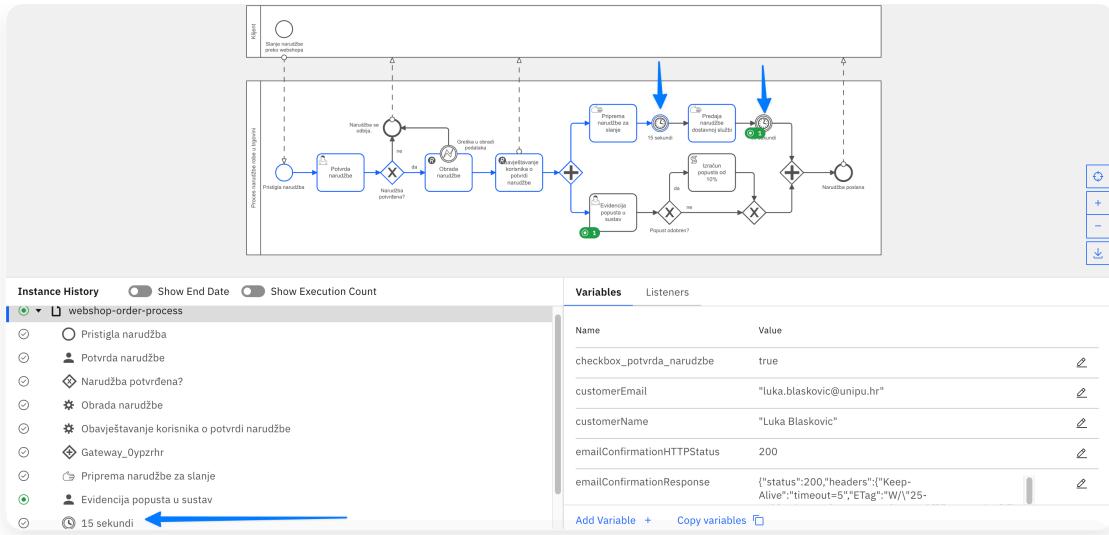
Pokazat ćemo osnovne stvari, međutim ako vas zanimaju detalji znate gdje možete pronaći više informacija - [Camunda 8 dokumentacija](#).

Primjerice, možemo **dodati apstraktno polje** "Klijent" i pokazati informacijske tokove između naše procesne aplikacije i klijenta kao nekog vanjskog entiteta. Mi nećemo izrađivati procesnu aplikaciju za klijenta (nema puno smisla - to bi bila obična web aplikacija za naručivanje proizvoda), ali ćemo pokazati kako bi to izgledalo na BPMN dijagramu radi boljeg razumijevanja modeliranog procesa.



Slika 23. Dodavanje apstraktnog polja "Klijent" i informacijskih tokova između procesne aplikacije i klijenta.

Također, možemo "simulirati" trajanje aktivnosti koristeći *timer evente*. Primjerice, možemo dodati *intermediate timer evente* nakon ručnih aktivnosti "Priprema narudžbe za slanje" i "Predaja narudžbe dostavnoj službi" kako bismo simulirali vrijeme potrebno za pripremu i predaju narudžbe. Naravno, u stvarnoj procesnoj aplikaciji, ove aktivnosti ne bi bile simulirane.



Slika 24. Dodavanje *intermediate timer* eventa za simulaciju trajanja aktivnosti nakon ručnih zadataka.

3.1 Potprocesi u procesnoj aplikaciji

Možemo dodati i potprocese u našu procesnu aplikaciju kako bismo grupirali povezane aktivnosti unutar većeg procesa. Primjerice, možemo izraditi potproces "Dostava narudžbe" gdje ćemo simulirati aktivnosti vezane uz dostavu narudžbe.

Samo ćemo simulirati aktivnosti unutar potprocesa koristeći ručne zadatke i *timer evente* - **ideja je pokazati kako se potproces ponaša unutar procesne aplikacije**.



Slika 25. Implementacija simulacije potprocesa "Dostava narudžbe kupcu" unutar glavnog procesa upravljanja narudžbama.

Potproces dodajemo u glavni proces nakon ručnog zadatka "Predaja narudžbe dostavnoj službi".

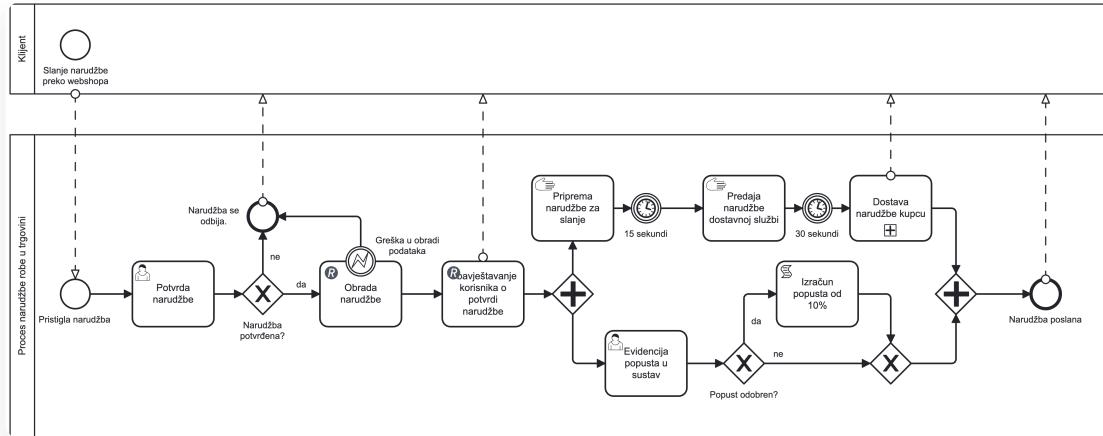
Unutar potprocesa imamo sljedeći događaja i aktivnosti:

- "Pickup proizvoda" (*start event*)
- "15 sekundi" (*intermediate timer event*) - simulira vrijeme potrebno za preuzimanje proizvoda od dostavne službe
- "Dostava proizvoda u sortirnicu" (ručni zadatak)
- "30 sekundi" (*intermediate timer event*) - simulira vrijeme potrebno za dostavu proizvoda u sortirnicu
- "Sortiranje proizvoda" (ručni zadatak)
- "Utovar u dostavno vozilo" (ručni zadatak)
- "Dostava naručenih proizvoda" (ručni zadatak)
- "30 sekundi" (*intermediate timer event*) - simulira vrijeme potrebno za dostavu proizvoda kupcu
- "Proizvodi dostavljeni" (*end event*)

Kod dodavanja timer eventa, pripazite da koristite ispravan format trajanja vremena u ISO 8601 formatu. Primjerice, za 15 sekundi koristimo `PT15S`, a za 30 sekundi `PT30S` i odabiremo **Duration** opciju.

Također, moguće je odabrat i **Date** opciju ako želite postaviti točan datum i vrijeme kada se događaj treba aktivirati.

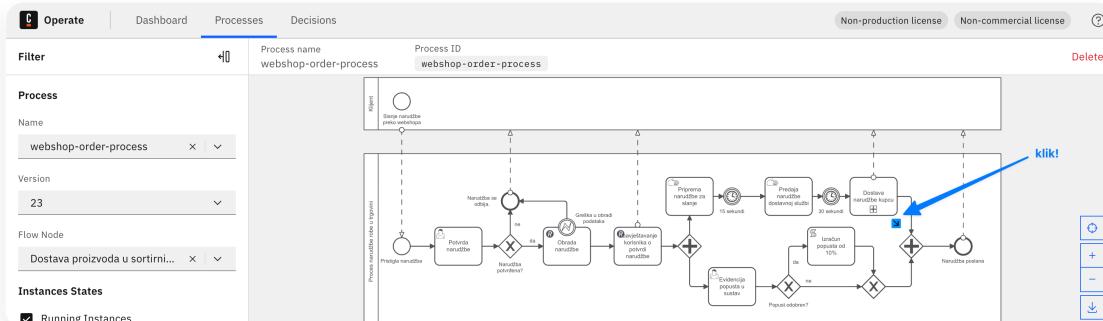
Više o vremenskim formatima možete pronaći [ovdje](#).



Slika 26. Cjelokupni BPMN dijagram procesne aplikacije za upravljanje narudžbama s dodanim potprocesom "Dostava narudžbe kupcu".

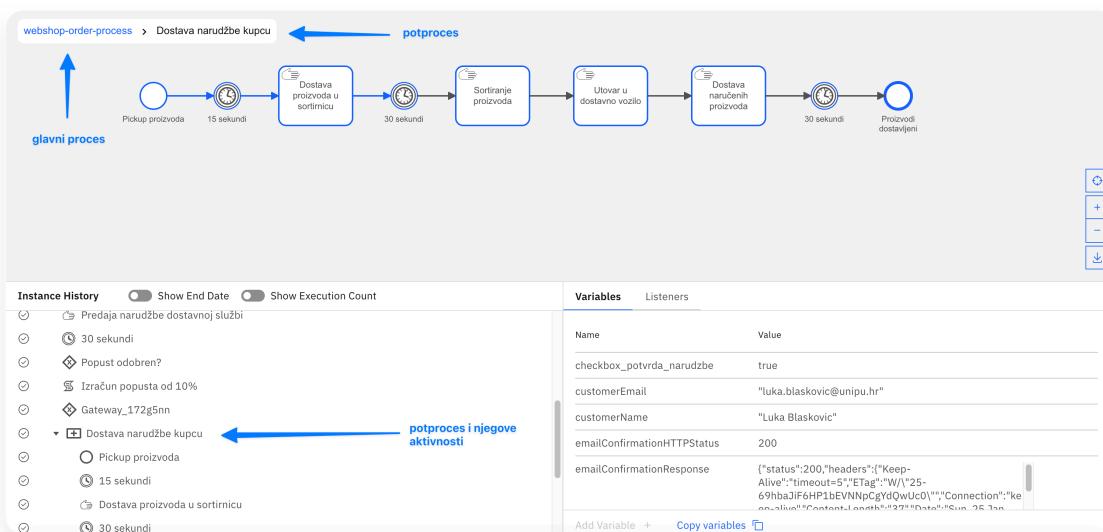
To je to! Možete *deployati* procesnu definiciju i isprobati kako će se potproces ponašati unutar glavnog procesa u Camunda Operate aplikaciji.

I prije nego pokrenete instancu, unutar Camunda Operate aplikacije vidjet ćete plavu strelicu pored potprocesa koja označava da je to potproces. Možete ju stisnuti i otvorit ćete vam ugniježđeni prikaz potprocesa.



Slika 27. Pregled potprocesa unutar Camunda Operate aplikacije.

Potproces se ponaša kao samostalni proces unutar glavnog procesa, ali je logički povezan s njim.



Slika 28. Praćenje izvršavanja instance potprocesa unutar Camunda Operate aplikacije.

4. DMN u procesnoj aplikaciji

Za kraj, možemo integrirati **DMN (Decision Model and Notation)** tablice odluka unutar naše procesne aplikacije kako bismo implementirali složeniju poslovnu logiku odlučivanja.

Umjesto fiksног iznosa popusta od 10% na cijelu narudžbu, koju djelatnik može ili ne mora primijeniti, možemo koristiti DMN tablicu odluka za dinamičko određivanje visine popusta na temelju ukupnog iznosa narudžbe.

Izradite novu DMN tablicu odluka naziva `izracunavanje_popusta.dmn` i pohranite ju u direktorij poslovne aplikacije.

Tablica ima jedan ulazni uvjet: **Vrijednost narudžbe** (`number`) i jedan izlazni rezultat: **Ukupni Popust (%)** (`number`).

Odaberite **unique hit policy** za tablicu i definirajte sljedeća poslovna pravila:

- ako je ukupni iznos narudžbe manji ili jednak `500`, popust je `0%`
- ako je ukupni iznos narudžbe između `501` i `1500`, popust je `10%`
- ako je ukupni iznos narudžbe veći od `1500`, popust je `15%`

| Izračunavanje popusta | | Hit policy: | Unique | |
|-----------------------|-------------------------------|-------------|-----------------------------|--|
| When | Vrijednost narudžbe number | Then | Ukupni popust (%) number | Annotations |
| 1 | <=500 | 0 | | Ako je vrijednost narudžbe <= 500\$, nema popusta |
| 2 | [501..1500] | 0.10 | | Ako je vrijednost narudžbe u rasponu 501 - 1500\$, popust je 10% |
| 3 | >1500 | 0.15 | | Ako je vrijednost narudžbe veća od 1500\$, popust je 15% |
| | + | - | | |

Slika 29. DMN tablica odluka za izračunavanje popusta na temelju ukupnog iznosa narudžbe.

Sljedeći korak je ispravno povezati naše podatke iz procesne aplikacije s DMN tablicom odluka.

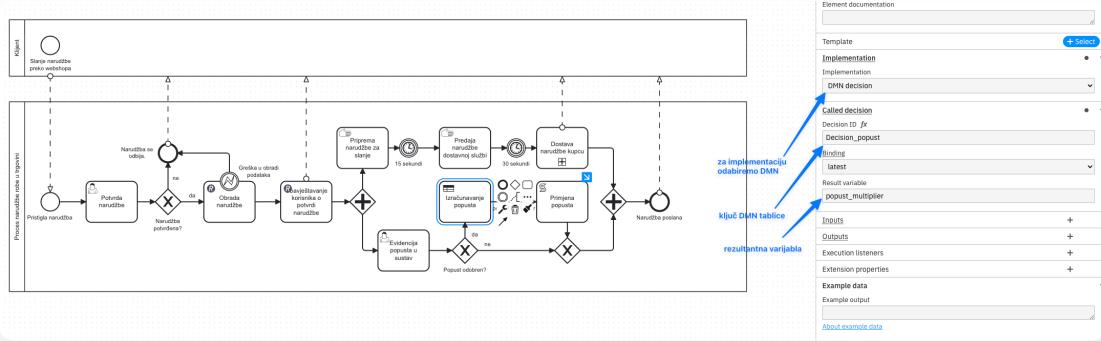
Vrijednost narudžbe pohranjena nam je u varijabli `order_totalAmount`, **stoga ćemo tu varijablu koristiti kao ulazni uvjet za DMN tablicu.**

- Odaberite uvjet `vrijednost narudžbe` i pod **Expression** unesite: `order_totalAmount` (bez znaka `=` jer je to već FEEL izraz).
- Odaberite izlazni rezultat `Ukupni popust (%)` i pod **Output name** unesite `popust_multiplier`. Ova varijabla će pohraniti rezultat DMN tablice, tj. iznos popusta koji ćemo koristiti u dalnjem toku procesa.

Script task za primjenu popusta ćemo preimenovati u "Primjena popusta" i izmijeniti njegovu implementaciju:

```
order_totalAmount - order_totalAmount * popust_multiplier;
```

Prije njega, dodajemo novi **Business rule task** koji ćemo povezati na našu DMN tablicu odluka `izracunavanje_popusta.dmn` - možemo ga nazvati "Izračunavanje popusta".

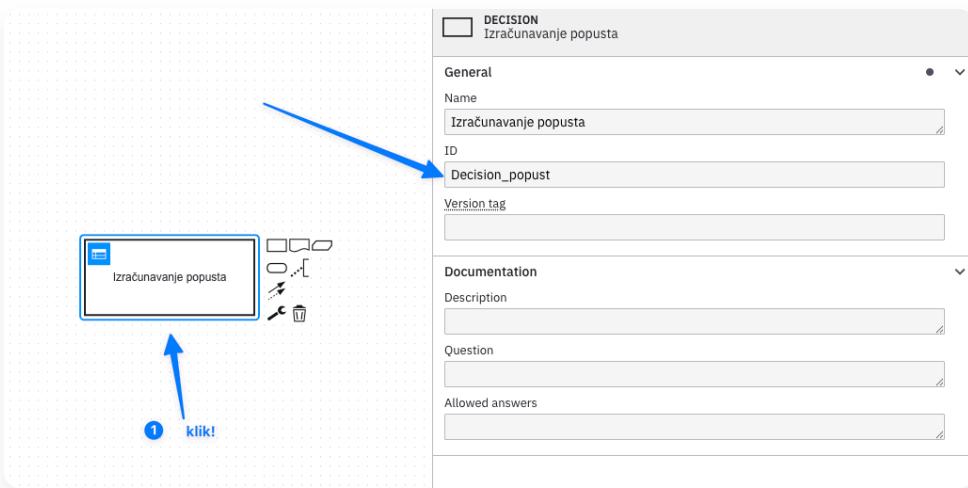


Slika 30. Dodavanje *Business Rule Taska* za povezivanje s DMN tablicom odluka.

Povežite tok procesa tako da *Business rule task* dolazi nakon "da" slijeda na XOR skretnici grananja "Popust odobren?" i prije *script taska* "Primjena popusta".

Kako biste povezali DMN i Business rule task, odaberite DMN i otvorite **DRD prikaz**. Unutar DRD prikaza, vidjet ćete ID vaše DMN tablice odluke. Mi ćemo ga promijeniti u `Decision_popust` radi lakšeg prepoznavanja.

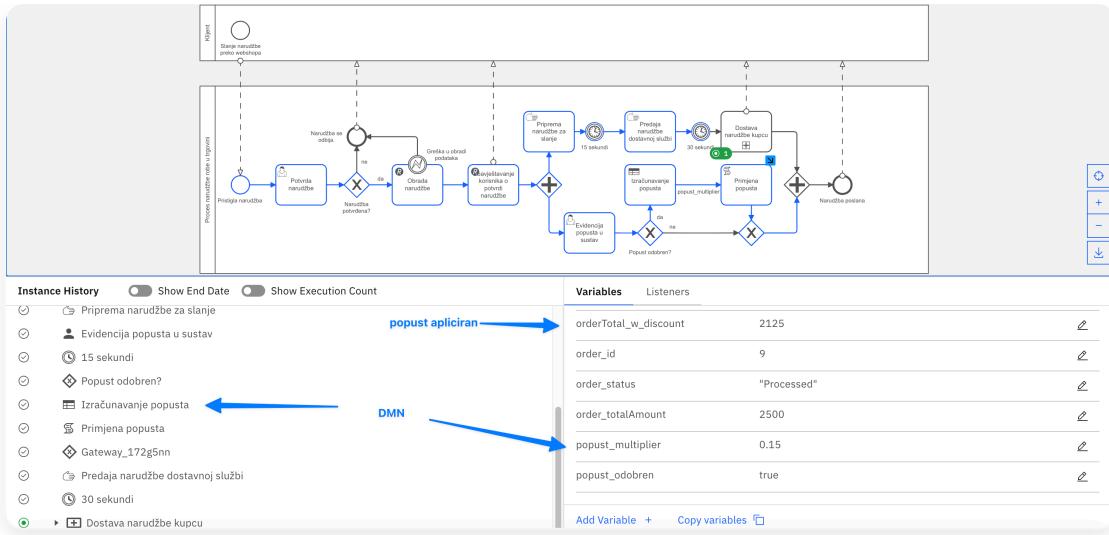
Možete još ažurirati formu za odobrenje popusta kako bi bila jasnija korisnicima - npr. promijeniti tekst pitanja u "Želite li primijeniti popust na ovu narudžbu?" ili slično.



Slika 31. Promjena ID-a DMN tablice odluka unutar DRD prikaza.

To je to! Pokrenite novu instancu procesa i isprobajte kako DMN tablica odlučuje o visini popusta na temelju ukupnog iznosa narudžbe.

Unutar Camunda Operate aplikacije, možemo pratiti izvršavanje procesa i vidjeti kako je DMN tablica odlučila o visini popusta. Uočite promjene u procesnim varijablama i primjenu Business rule taska iz *Instance History* prikaza.



Slika 32. Primjer izvršavanja procesa u *Operateu* gdje je DMN tablica odlučila o visini popusta na temelju ukupnog iznosa narudžbe.

Možemo iz procesnih varijabli u Operate aplikaciji uočiti da se na našu narudžbu od `2500$` uspješno primijenio popust od `15%`, te je konačni iznos narudžbe sada `2125$`.

Zadaci za Vježbu 7

Zadatak je proći kroz cijelu skriptu i nadograditi procesnu aplikaciju za upravljanje narudžbama koristeći naučene naprednije koncepte.

Nakon što završite, izmijenite procesni model i poslovnu aplikaciju na način da se poruka o potvrdi narudžbe šalje tek aktivnosti "Evidencija popusta u sustav", tada je potrebno u predlošku emaila korisniku poslati stvarni iznos narudžbe nakon primjene popusta.

Također, implementirajte dodatnu DMN tablicu i Business rule task koji računaju troškove dostave na temelju ukupne narudžbe i porez na temelju lokacije kupca (pošaljite ove vrijednosti kao dodatne procesne varijable prilikom instanciranja procesa).

Nakon što završite, izmijenite kod Express poslužitelja na način da ti podaci više ne kodiraju, već šalju iz procesne aplikacije kod aktivnosti "Obrada narudžbe".

Predajete zip datoteku koja sadrži sve potrebne datoteke procesne aplikacije uključujući screenshotove iz Operate i Tasklist aplikacija koji dokazuju ispravno izvođenje procesa.