

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

(6) Servisna arhitektura procesne aplikacije

#6

UPP

Servisna arhitektura Camunda aplikacije obuhvaća dizajn i implementaciju raspodijeljenog sustava temeljenog na malim servisima (mikroservisi) koji komuniciraju preko REST API-ja. Camunda 7, kao jezgra procesne aplikacije pruža mogućnost izvođenja poslovnih procesa, njihovo upravljanje i praćenje, ali i integraciju s mikroservisima i vanjskim sustavima. Mikroservisi su ništa drugo nego neovisne aplikacije koje obavljaju specifične zadatke, u ovom kontekstu možemo ih zamisliti kao jednostavne REST API poslužitelje koji obavljaju određene radnje koje sami definiramo. U ovoj skripti naučit ćete integrirati jednostavne mikroservise s Camunda procesnim engineom kroz servisne zadatke i Express.js poslužitelje.

Posljednje ažurirano: 5.1.2025.

Sadržaj

- [Upravljanje poslovnim procesima \(UPP\)](#)
- [\(6\) Servisna arhitektura procesne aplikacije](#)
- [Sadržaj](#)
- [1. Servisni zadaci \(eng. Service Task\)
 - \[1.1 Priprema poslužitelja\]\(#\)
 - \[1.2 Slanje HTTP GET zahtjeva\]\(#\)
 - \[1.3 Dohvaćanje statusnog koda \\(`statusCode`\\)\]\(#\)
 - \[1.4 Dohvaćanje tijela odgovora \\(`response`\\)\]\(#\)
 - \[1.5 Slanje HTTP POST zahtjeva\]\(#\)](#)
- [2. Otpremni zadaci \(eng. Send Task\)
 - \[2.1 Priprema poslužitelja za automatsko slanje e-maila\]\(#\)
 - \[2.2 Email.js - priprema predloška\]\(#\)](#)

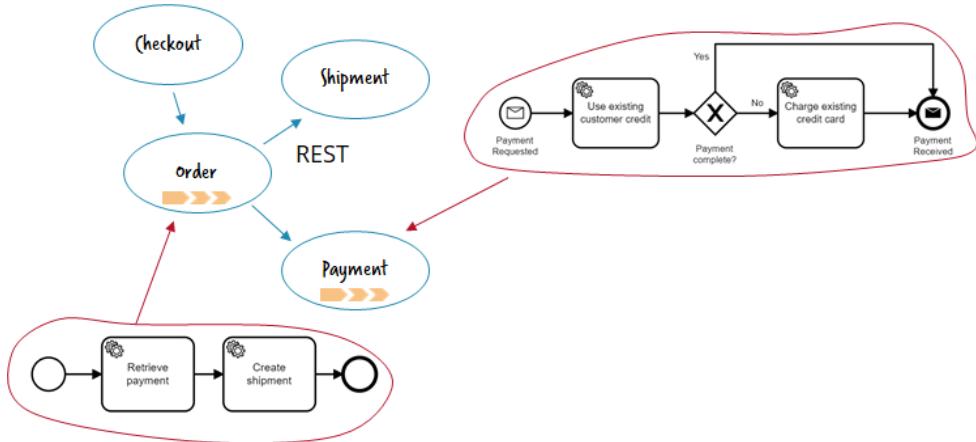
- [2.3 Implementacija slanja e-mail poruke](#)
- [2.4 Definiranje send Task aktivnosti](#)

1. Servisni zadaci (eng. Service Task)

Do sam smo iz vježbi najmanje govorili o servisnim zadacima (*eng. Service task*), a oni su zapravo jedan od najvažnijih elemenata procesnih aplikacija. Naučili ste da servisne zadatke koristimo za izvođenje vanjskih, automatiziranih operacija gdje imamo ulazne podatke, želimo napraviti nekakvu transformaciju ili akciju, te dobiti izlazne podatke.

Camunda podržava različite načine implementacije servisnih zadataka, npr. kroz JavaDelegate sučelje (Java), Expressione, Script Task (JavaScript, Groovy, Python, Ruby), ali i kroz REST API pozive (HTTP protokol). Mi ćemo se fokusirati na posljednji princip, odnosno na REST API pozive.

Za slanje HTTP zahtjeva koristit ćemo **Camunda7 Connectors API**, preciznije [http-connector](#) modul. Ovaj modul dolazi s Camunda platformom te ga nije potrebno naknadno instalirati.



Ilustracija servisne arhitekture procesne aplikacije

1.1 Priprema poslužitelja

Prije nego što krenemo s implementacijom servisnih zadataka, potrebno je pripremiti poslužitelj na koji ćemo slati HTTP zahtjeve. Poslužitelj može definirati u bilo kojem programskom jeziku/razvojnom okruženju, no mi ćemo koristiti Node.js i Express.js.

Izradite novi direktorij `express-server` i inicijalizirajte novi Node.js projekt:

```
mkdir express-server
cd express-server
npm init -y
```

Instalirajte Express.js:

```
npm install express
```

U `package.json` podesite `"type": "module"`, kako biste mogli koristiti `ES6` module te definirajte osnovni `Express.js` poslužitelj:

```
// express-server/index.js

import express from "express";

const PORT = 8000;

const app = express();
app.use(express.json());

app.get("/", (req, res) => {
  res.send("Pozdrav iz Express poslužitelja!");
});

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

Provjerite radi li poslužitelj slanjem GET zahtjeva na `http://localhost:8000`.

Kako bismo mogli nesmetano slati zahtjeve s Camunda platforme, potrebno je omogućiti CORS (Cross-Origin Resource Sharing) politiku. Instalirajte `cors` paket:

```
npm install cors
```

Dodajemo `cors` middleware u `Express.js` poslužitelj:

```
// express-server/index.js

import express from "express";
import cors from "cors";

const PORT = 8000;

const app = express();
app.use(express.json());
app.use(cors());

app.get("/", (req, res) => {
  res.send("Pozdrav iz Express poslužitelja!");
});

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

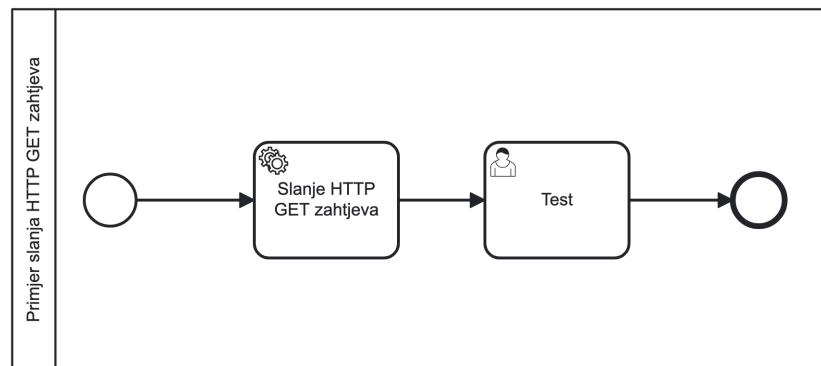
To je to! Za sada. Vraćamo se u Camunda Modeler 

1.2 Slanje HTTP GET zahtjeva

Definirat ćemo jednostavni proces koji se sastoji od jednog servisnog zadatka koji šalje HTTP GET zahtjev na naš Express.js poslužitelj.

Kako bismo vidjeli što se dešava, odnosno kako naša procesna instanca ne bi odmah završila, dodat ćemo i **User Task** neposredno nakon **Service Task** elementa.

Dodajte novo polje te u postavkama postavite osnovne podatke kako bi mogli izraditi procesnu instancu.



Jednostavna procesna definicija koja se sastoji od **Service Task** i **User Task** elemenata

Definirat ćemo i jednostavnu formu za **User Task** element:

USER TASK
Test

General

Name: Test

ID: Activity_1mzdr55

Documentation

Element documentation

User assignment

Forms

Type: Generated Task Forms

Form fields

+ 1

test

ID: test

Refers to the process variable name

Label: Želite li nastaviti?

Type: boolean

Default value:

Constraints

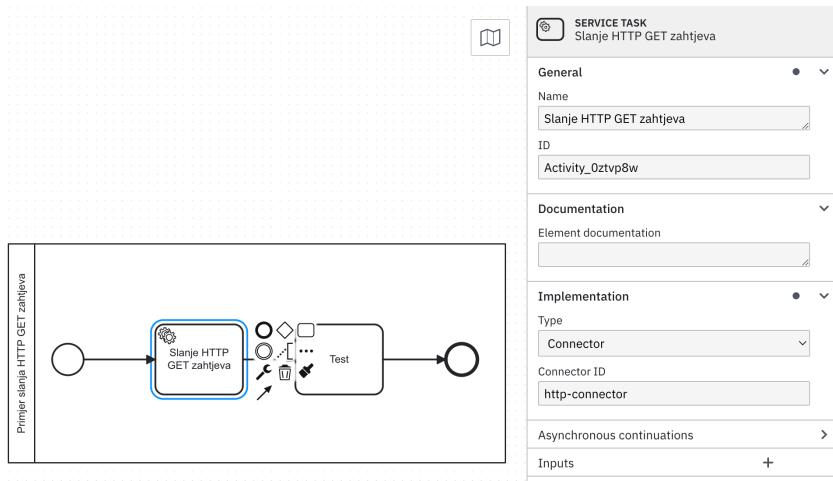
Properties

Dodavanje polja u **User Task** formu

Ako već niste, pokrenite **Camunda 7** preko Dockera:

```
docker run -d --name camunda -p 8080:8080 camunda/camunda-bpm-platform:latest
```

Za kraj, kako ne bi dobili grešku, moramo odabratи **connector** implementaciju servisnog zadatka, a za ID postaviti **http-connector**.



Deployajte procesnu definiciju na *Camunda Engine* i pokrenite novu procesnu instancu.

Vidimo da je *deployment* procesne definicije bio uspješan, ali procesnu instancu nije moguće pokrenuti. U konzoli Camunda Modelera vidjet ćete grešku:

```
HTCL-02005 Request url required. [ start-instance-error ]
```

Ova greška se javlja jer nismo definirali **URL** na koji će se poslati HTTP zahtjev.

Već iz web aplikacija znamo da su **obavezni dijelovi HTTP zahtjeva** `URL` i `method`, a **opcionarni dijelovi** su `payload` i `headers`. Isto primjenjujemo i ovdje:

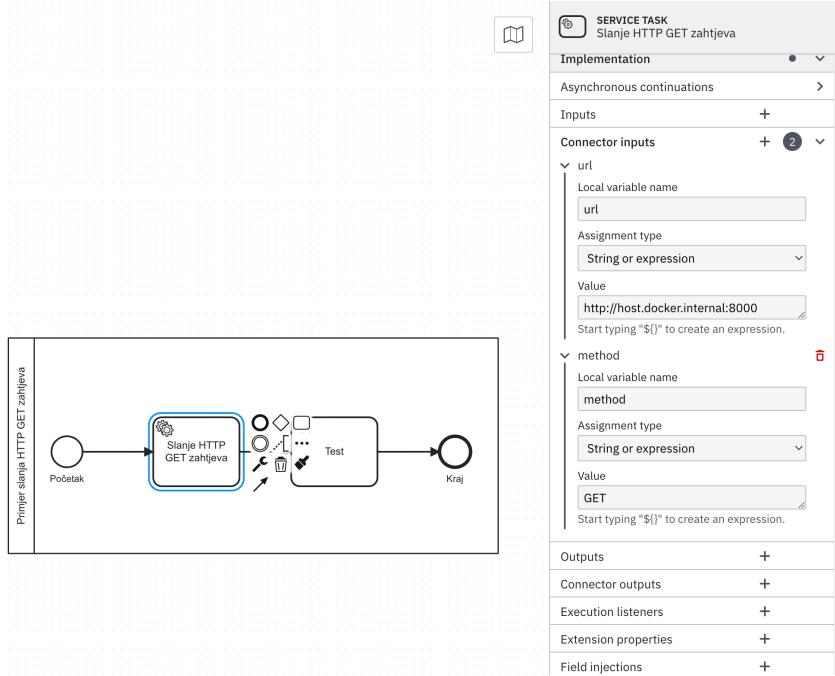
- `url`: ciljni URL gdje sluša naš poslužitelj
- `method`: HTTP metoda koju koristimo (GET, POST, PUT, DELETE, PATCH)
- `payload`: tijelo HTTP zahtjeva tj. *key-value* podaci koje šaljemo (JSON, XML)
- `headers`: dodatno zaglavlje HTTP zahtjeva (npr. Content-Type, Authorization)

Ove podatke navodit ćemo u **Connector inputs** polju servisnog zadatka.

- **Local variable name =** `url`, **Assignment type =** `String or expression`, **Value =**
`http://host.docker.internal:8000`
- **Local variable name =** `method`, **Assignment type =** `String or expression`, **Value =** `GET`

Razlog zašto koristimo `http://host.docker.internal:8000` umjesto `http://localhost:8000` je taj što se `localhost` u kontekstu kontejnera ne referencira na naše računalo, već na internu adresu docker kontejnera. Više informacija o tome na: <https://docs.docker.com/engine/network/>

`http://host.docker.internal` je adresa koja **omogućava pristup resursima na domaćinu iz Docker containera**.



Dodavanje `Connector inputs` polja (`url` i `method`) u servisni zadatak "Slanje HTTP GET zahtjeva"

Za kraj, dodat ćemo i jednostavan `console.log` unutar definicije rute na Express.js poslužitelju kako bismo se uvjerili da je zahtjev uspješno poslan.

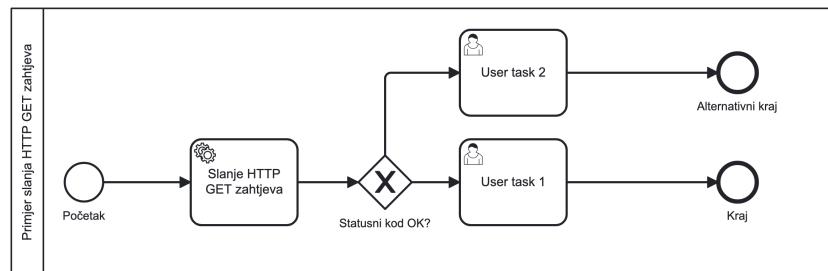
Redeployajte procesnu definiciju i pokušajte ponovno pokrenuti procesnu instancu. Pazite da je Express.js poslužitelj pokrenut.

Ako ste sve napravili ispravno, trebali biste vidjeti ispis u konzoli Express.js poslužitelja koji dokazuje da je GET zahtjev uspješno poslan 🚀

1.3 Dohvaćanje statusnog koda (statusCode)

Recimo da želimo preusmjeriti tok procesa koristeći XOR skretnicu temeljem statusnog koda koji dobijemo kao odgovor na HTTP zahtjev. Ukoliko je statusni kod 200, preusmjerit ćemo tok na `User Task 1`, a u suprotnom na `User Task 2`.

Dakle, želimo implementirati sljedeći sekvenčijalni tok:



Što se tiče `Connector outputs` polja, dostupne su sljedeće varijable:

- `response`: **tijelo odgovora** HTTP zahtjeva (`String`)
- `statusCode`: **statusni kod** HTTP odgovora (`Integer`)
- `responseHeaders`: **zaglavlje** HTTP odgovora (`Map<String, String>`)

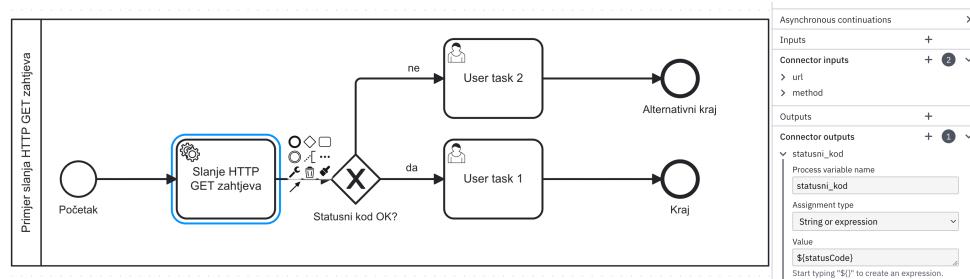
Kako bismo mogli dohvatiti statusni kod, potrebno je dodati novu varijablu u `connector outputs` polje. Međutim, uočite da je prvo polje zapisa naziva `Process variable name`, dakle varijablu možemo nazvati kako god želimo, ona će postati **procesna varijabla** dostupna u cijelom procesu (kao što smo ih definirali i u prethodnim vježbama).

Namjerno ćemo ju nazvati `statusni_kod` kako biste uočili razliku. Za dohvaćanje samog statusnog koda, moramo koristiti Camunda Expression `#{statusCode}`

- **Process variable name** = `statusni_kod`, **Type** = `String or expression`, **Value** = `#{statusCode}`

Na XOR skretnicu dodajte sljedeće uvjete:

- `#{statusni_kod == 200}` : nastavlja sekvencijalni flow prema `User Task 1`
- `#{statusni_kod != 200}` : nastavlja sekvencijalni flow prema `User Task 2`



Za kraj, unutar Express.js poslužitelja vratite statusni kod 200 kako biste preusmjerili tok na `User Task 1`.

```
// express-server/index.js

app.get("/", (req, res) => {
  console.log("Zahtjev primljen!");
  res.status(200).send("Pozdrav iz Express.js poslužitelja!");
});
```

Redeployajte procesnu definiciju i pokušajte ponovno pokrenuti procesnu instancu.

Ako otvorite pregled procesne instance u **Cockpitu**, vidjet ćete da je tok preusmjeren prema `User Task 1` jer je statusni kod 200. Pohranjenu procesnu varijablu `statusni_kod` možete vidjeti u detaljima procesne instance.

- **Name** = `statusni_kod`, **Type** = `Integer`, **Value** = `200`, **Scope** = `connector_GET`

Name	Type	Value	Scope	Actions
statusni_kod	Integer	200	connector_GET	edit remove

Testirajte i drugi uvjet tako da promijenite statusni kod u Express.js poslužitelju na npr. 404.

1.4 Dohvaćanje tijela odgovora (response)

Ukoliko želimo dohvatiti tijelo odgovora HTTP zahtjeva, koristimo varijablu `response`. U ovom primjeru, želimo dohvatiti tijelo odgovora i pohraniti ga u procesnu varijablu kako bismo ga mogli koristiti u dalnjem toku procesa.

Možemo isto testirati odmah, budući da naš poslužitelj metodom `res.send` šalje odgovor koji je zapravo tijelo odgovora (nije JSON).

Dodajte novu procesnu varijablu `odgovor` koja će pohraniti tijelo odgovora HTTP zahtjeva. Varijablu dodajte na jednak način: kao **Connector output** varijablu s Expression izrazom `${response}` .

Redeployajte procesnu definiciju i pokušajte ponovno pokrenuti procesnu instancu.

Možemo vidjeti procesnu varijablu `odgovor` u detaljima procesne instance u **Cockpitu**.

Name	Type	Value	Scope	Actions
odgovor	String	Pozdrav iz Express.js poslužitelja!	connector_GET	edit remove
statusni_kod	Integer	200	connector_GET	edit remove

Međutim, iz web aplikacija znamo da nije uobičajeno slati tekstualne odgovore na ovaj način, već koristimo JSON. Vratit ćemo jednostavan objekt `key-value` parova kao JSON odgovor te ga zatim pohraniti u procesnu varijablu.

Recimo da naš API simulira dohvaćanje podataka o korisniku. Dodat ćemo novu rutu u Express.js poslužitelj koja vraća JSON odgovor s imenom i prezimenom korisnika.

```
app.get("/user", (req, res) => {
  console.log("Zahtjev primljen na /user");
  res.status(200).json({
    ime: "Marko",
    prezime: "Marić",
  });
});
```

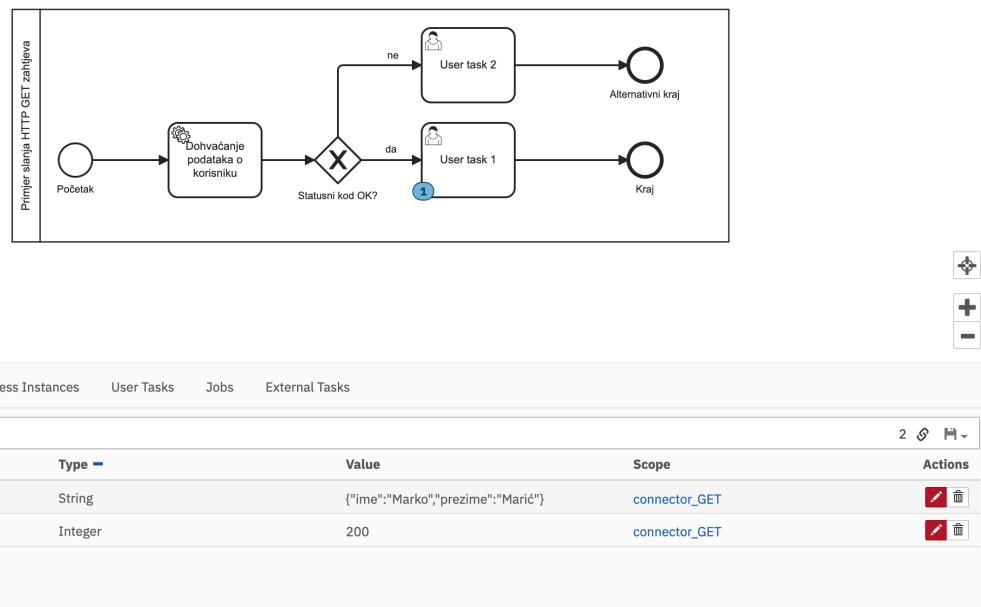
Service task ćemo sad preimenovati u "Dohvaćanje podataka o korisniku" te promijeniti URL na `http://host.docker.internal:8000/user`.

Redeployajte procesnu definiciju i pokušajte ponovno pokrenuti procesnu instancu.

Ako provjerite detalje procesne instance u **Cockpitu**, vidjet ćete da je tijelo odgovora pohranjeno u procesnu varijablu `odgovor`, međutim kao `String`.

- **Name** = `odgovor`, **Type** = `String`, **Value** = `{"ime": "Marko", "prezime": "Marić"}`, **Scope** = `connector_GET`

Ukoliko želimo koristiti taj odgovor kao objekt, moramo odraditi proces **deserializacije**.



Ideja je da pohranimo tijelo odgovora u dvije nove procesne varijable `ime` i `prezime`

U Camundi postoji mnogo načina za odraditi ovu transformaciju, primjerice isto je moguće postići koristeći **Script Task** element u kojeg ćemo direktno pisati JavaScript ili Java kod. Međutim, mi ćemo nastojati iskoristiti stvari rješavati na što jednostavniji način, koristeći **Expressione**.

[Camunda Spin](#) biblioteka nudi jednostavan API za rad s XML i JSON struktukrom. Međutim, jednako kao i Connector, nije ju potrebno naknadno instalirati jer dolazi s Camunda platformom.

Camunda Spin Expressione počinjemo pisati sa slovom `s`. Sintaksa je sljedeća:

```
 ${S("JSON").prop("key")}
```

Budući da je naš JSON odgovor pohranjen u procesnu varijablu `odgovor`, možemo upotrijebiti procesnu varijablu direktno kao argument Spin Expressiona.

```
 ${S(odgovor).prop("ime")}
```

Za kraj, sam proces deserijalizacije radimo naredbom `.value()`:

```
 ${S(odgovor).prop("ime").value()}
```

To je to! Sad možemo odraditi proces deserijalizacije u jednom Expression izrazu, bez potrebe za pisanjem koda.

Ostalo je jedino pitanje - gdje dodajemo ove Expressione?

Svaka aktivnost (zadatak) pa tako i `Service Task` ima dostupna polja za **Input/Output Mapping**:

- `Inputs`: **podaci koji se koriste kao ulazni parametri za aktivnost**. Ovdje možemo definirati koje procesne varijable će se koristiti kao ulazni parametri za tu aktivnost.
- `Outputs`: **rezultati aktivnosti**. Ovdje možemo definirati koje procesne varijable će se koristiti kao izlazni parametri za tu aktivnost.

U skripti `UPP5` već ste vidjeli kako koristimo ova polja, no mi ćemo sada iskoristiti `Outputs` polje kako bismo deserijalizirali tijelo odgovora.

Zašto `Outputs`? Jer želimo da **rezultat** aktivnosti "Dohvaćanje podataka o korisniku" budu procesne varijable `ime` i `prezime`, a ne `odgovor`.

Oprez: pazite da ne pomiješate Input s Connector inputima i Output s Connector outputima.

Connector inputi i outputi su vezani uz HTTP zahtjev Connector API-ja, dok su **Input/Output Mapping vezani uz samu aktivnost**.

Napravit ćemo mali rezime prije nego nastavimo:

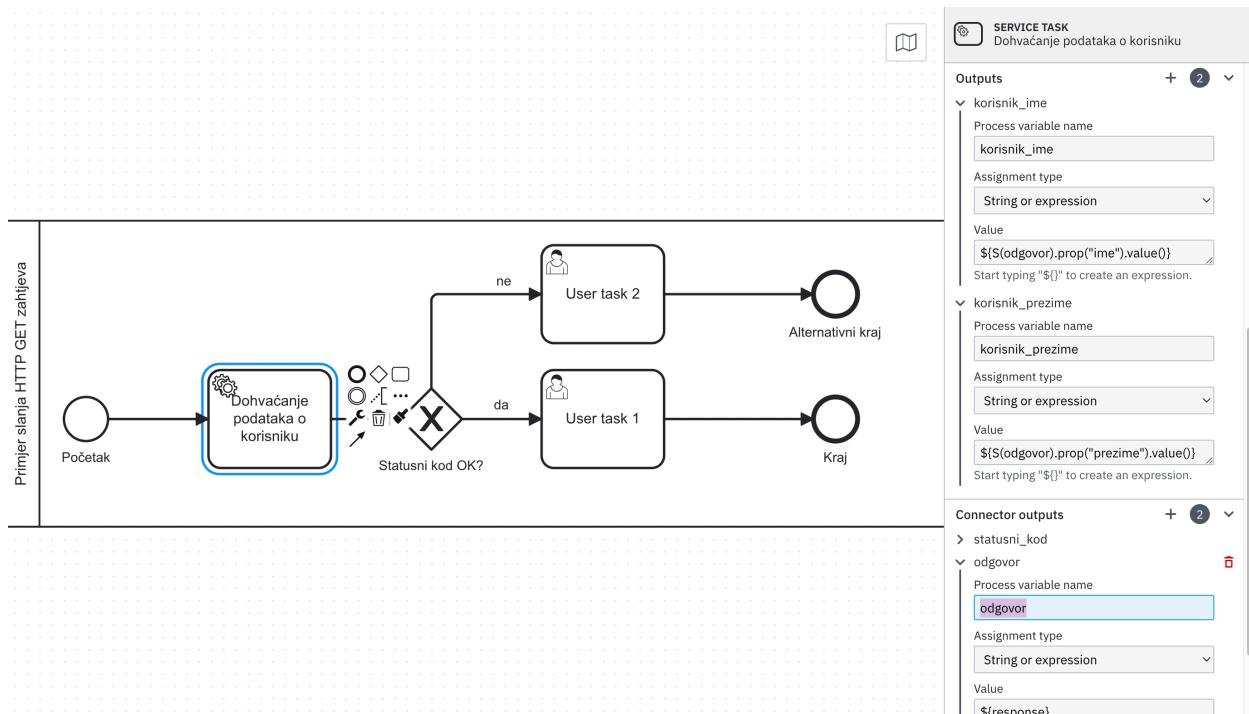
1. Procesni *engine* nailazi na `Service Task` "Dohvaćanje podataka o korisniku"
2. Navedeni `Service Task` implementiran je kao `Connector`, preciznije `http-connector` jer smo rekli da koristimo tu implementaciju za slanje HTTP zahtjeva (iako ih ima više)
3. U `Connector inputs` polju definirali smo URL i metodu HTTP zahtjeva, konkretno to je slanje GET zahtjeva koji dohvaca podatak o korisniku s našeg Express.js poslužitelja
4. U `Connector outputs` polju definirali smo da želimo pohraniti **statusni kod i tijelo odgovora** u procesne varijable
5. Kako je tijelo odgovora JSON, želimo ga deserijalizirati i pohraniti u procesne varijable `ime` i `prezime`. Sad već imamo taj JSON u procesnoj varijabli `odgovor`, pa koristimo Spin Expressione za deserijalizaciju unutar polja `Outputs` za `Service Task` element.

Dakle, **S Expressioni** koje ćemo koristiti u `Outputs` polju su sljedeći:

- `ime: ${S(odgovor).prop("ime").value()}`
- `prezime: ${S(odgovor).prop("prezime").value()}`

Na isti način možemo deserijalizirati bilo koji drugi primitivni tip podatka.

Redeployajte procesnu definiciju i pokušajte ponovno pokrenuti procesnu instancu.



Deserijalizacija JSON podataka iz tijela odgovora u procesne varijable `ime` i `prezime`

Provjerite vrijednosti svih procesnih varijabli u Cockpitu.

1.5 Slanje HTTP POST zahtjeva

Naučili smo kako poslati HTTP GET zahtjev, dohvatiti statusni kod i tijelo odgovora, te deserijalizirati JSON odgovor. Sada ćemo vidjeti kako poslati HTTP POST zahtjev, odnosno **kako poslati vrijednosti procesnih varijabli na poslužitelj**.

Idemo implementirati jednostavni proces gdje korisnik unosi svoje prezime, a servisni zadatak šalje to prezime na poslužitelj kako bi dohvatio ukupne podatke o korisniku (`ime`, `prezime`, `username`, `email`). Ako poslužitelj ne pronađe korisnika, vraća statusni kod 404. U suprotnom vraća statusni kod 200 i JSON objekt koji predstavlja korisnika.

Podatke, iako bi trebali biti pohranjeni u bazi podataka, ćemo pohraniti samo *in-memory*.

Dodat ćemo novu rutu u Express.js poslužitelj koja simulira dohvaćanje korisnika **na temelju prezimena**.

Iako prema standardu REST protokola, ovu radnju bi modelirali GET metodom i **parametrom rute** (npr. `/user/Marić`) ili **query parametrima** (npr. `/user?prezime=Marić`), mi ćemo koristiti **POST metodu i poslati prezime u tijelu zahtjeva**, budući da ova distinkcija nije bitna u kontekstu ovog kolegija:

Definirat ćemo nekoliko korisnika u memoriji:

```
// express-server/index.js
```

```
// legendarni trio
let korisnici = [
  {
    ime: "Marko",
    prezime: "Marić",
    username: "marko.maric",
    email: "mmaric@gmail.com"
  },
  {
    ime: "Pero",
    prezime: "Perić",
    username: "ppppp.pero",
    email: "pero123@gmail.com"
  },
  {
    ime: "Ana",
    prezime: "Anić",
    username: "ana.anic",
    email: "aanic@gmail.com"
  }
]
```

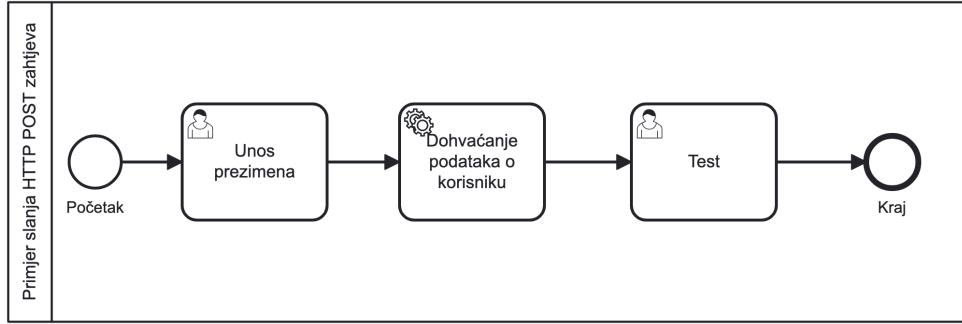
Dodajemo definiciju POST rute na poslužitelju:

```
// express-server/index.js

app.post("/user", (req, res) => {
  console.log("Zahtjev primljen na /user");
  let prezime = req.body.prezime;
  let korisnik = korisnici.find((korisnik) => korisnik.prezime === prezime); // pronalazak
  korisnika
  if (!korisnik) {
    return res.status(404).json({ message: "Korisnik nije pronađen." });
  } else {
    return res.status(200).json(korisnik);
  }
});
```

Implementirat ćemo sljedeći proces: "Primjer slanja HTTP POST zahtjeva":

1. Korisnik unosi svoje prezime preko **User Task** elementa "Unos prezimena"
2. **Service Task** "Dohvaćanje podataka o korisniku" **šalje prezime na poslužitelj HTTP POST metodom** i vraća objekt korisnika
3. **User Task** "Test" samo stopira proces kako bi mogli vidjeti rezultate u Cockpitu



Procesna definicija sa servisnim zadatkom koji šalje HTTP POST zahtjev na poslužitelj

Dalje, definirat ćemo jednostavnu formu za unos prezimena u **User Task** elementu "Unos prezimena".

```

graph LR
    Start((Početak)) --> Unos[Unos prezimena]
    Unos --> Dohv[Dohvaćanje podataka o korisniku]
    Dohv --> Test[Test]
    Test --> Kraj((Kraj))

```

USER TASK
Unos prezimena

General	•	▼
ID	Activity_1tec2gw	
Documentation	Element documentation	
User assignment	>	
Forms	•	▼
Type	Generated Task Forms	

Form fields

+ 1	▼
prezime	
ID	prezime
Refers to the process variable name	
Label	Molimo da unesete prezime korisnika
Type	string
Default value	

Dodavanje polja za unos prezimena u **User Task** formu

Za **Service Task** element, postavite sljedeće **Connector inputs** vrijednosti:

- **Local variable name** = `url`, **Assignment type** = `String or expression`, **Value** = `http://host.docker.internal:8000/user`
- **Local variable name** = `method`, **Assignment type** = `String or expression`, **Value** = `POST`

Tijelo zahtjeva definiramo u varijabli `payload`. Međutim, moramo ga poslati u onom obliku koji poslužitelj očekuje, a to je JSON format s ključem `prezime` i vrijednošću koju korisnik unese (procesna varijabla `prezime`).

Prije slanja, dodat ćemo u implementaciji POST rute na poslužitelju `console.log` kako bismo ispisali tijelo zahtjeva.

```
// express-server/index.js

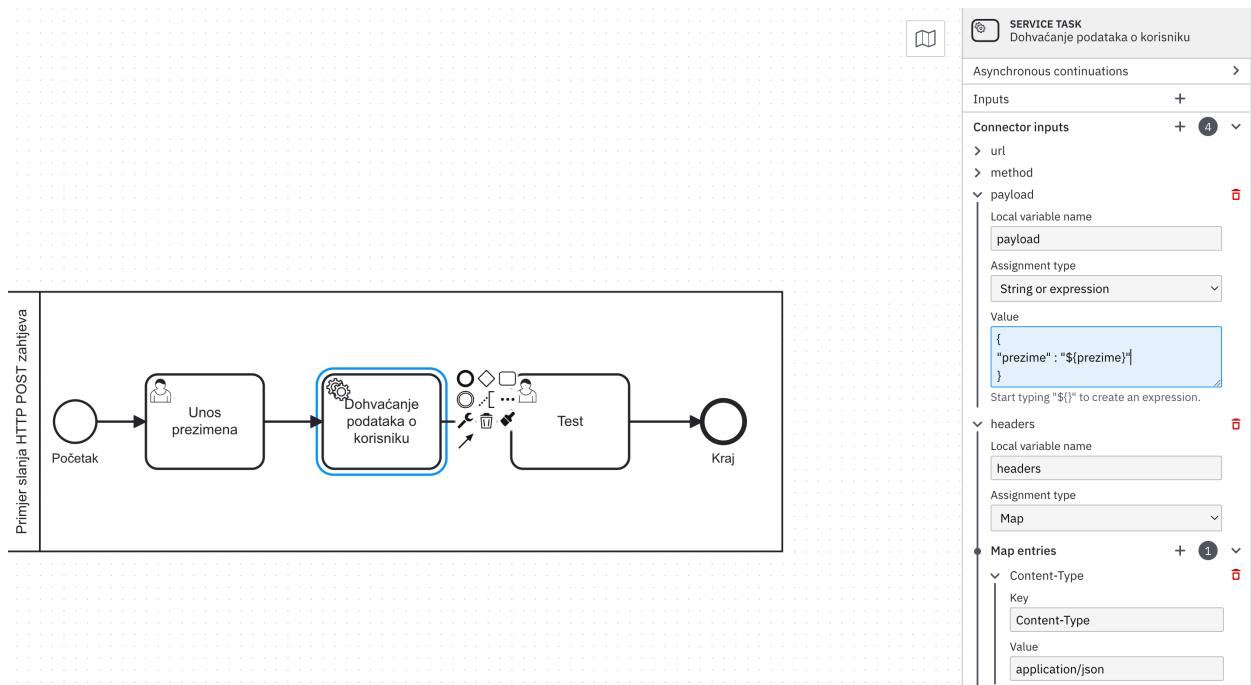
app.post("/user", (req, res) => {
  console.log("Zahtjev primljen na /user");
  console.log(req.body); // ispis tijela zahtjeva
  let prezime = req.body.prezime;
  let korisnik = korisnici.find((korisnik) => korisnik.prezime === prezime);
  if (!korisnik) {
    return res.status(404).json({ message: "Korisnik nije pronađen." });
  } else {
    return res.status(200).json(korisnik);
  }
});
```

U `Connector inputs` polju dodajemo novu varijablu `payload`.

- **Local variable name** = `payload`, **Assignment type** = `String or expression`, **Value** = `{"prezime": "${prezime}"}`

Dodatno, moramo poslati i zaglavje (`header`) kako bi poslužitelj znao da se radi o JSON formatu.

- **Local variable name** = `headers`, **Assignment type** = `Map`, **Map entries** (Key = `Content-Type`, Value = `application/json`)



Obavezno je potrebno proslijediti i `Content-Type` zaglavje kako bi poslužitelj ispravno interpretirao tijelo zahtjeva. Tada u `payload` možemo jednostavno pisati JSON objekt kao string.

Budući da svaki korisnik sadrži podatke o imenu, prezimenu, korisničkom imenu i e-mailu, dodat ćemo **preostale izlazne procesne varijable** u Outputs polju `Service Task` elementa.

- `ime: ${S(odgovor).prop("ime").value()}`
- `username: ${S(odgovor).prop("username").value()}`
- `email: ${S(odgovor).prop("email").value()}`

Redeployajte procesnu definiciju i pokušajte ponovno pokrenuti procesnu instancu.

Ako ste sve napravili ispravno, nakon unosa prezimena putem Tasklist aplikacije, vidjet ćete ispis tijela zahtjeva na poslužitelju, a zatim i dohvaćene podatke o korisniku u Cockpitu.

Camunda Cockpit Processes Decisions Human Tasks More ▾

Dashboard > Processes » connector_POST : 3d7593c6-cabd-11ef-b5d6-0242ac110002:Runtime

Information Filter

Instance ID: 3d7593c6-cabd-11ef-b5d6-0242ac1...

Business Key: null

Definition Version: 8

Definition ID: connector_POST:8:3b81b305-cabd-1...

Definition Key: connector_POST

Definition Name: connector_POST

Tenant ID: null

Deployment ID: 3b77c7f3-cabd-11ef-b5d6-0242ac1...

Super Process Instance ID: null

Diagram:

```
graph LR; Start((Početak)) --> Unos[Unos prezimena]; Unos --> Dohv[Dohvadanje podataka o korisniku]; Dohv --> Test[Test]; Test --> End((Kraj));
```

Primer slanja HTTP POST zahtjeva

Variables

Name	Type	Value	Scope	Actions
email	String	ppero123@gmail.com	connector_POST	[edit] [refresh]
ime	String	Pero	connector_POST	[edit] [refresh]
odgovor	String	{"ime":"Pero","prezime":"Perić","username": "pppp.pero", "statusni_kod": 200}	connector_POST	[edit] [refresh]
prezime	String	Perić	connector_POST	[edit] [refresh]
statusni_kod	Integer	200	connector_POST	[edit] [refresh]
username	String	pppp.pero	connector_POST	[edit] [refresh]

2. Otpremni zadaci (eng. Send Task)

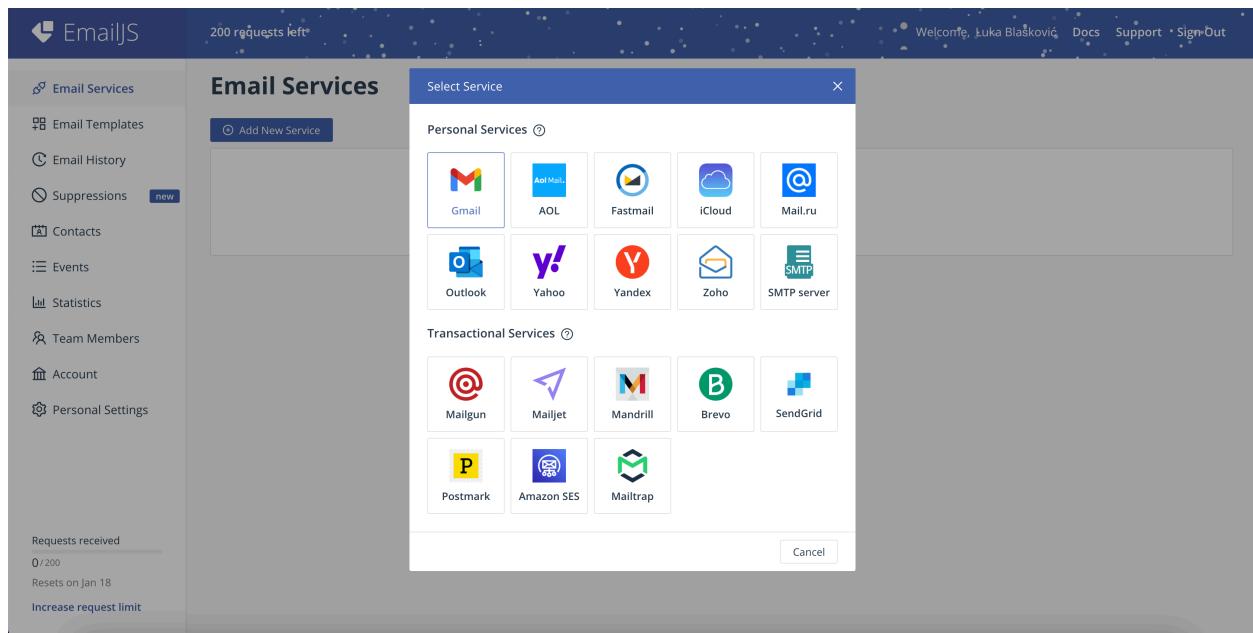
Otpremni zadaci (end. *Send Task*) predstavljaju aktinosti koje uključuju slanje poruka vanjskim dionicima, ali ih možemo koristiti i za pokretanje novih procesa.

Kao i kod servisnih zadataka, `Send Task` implementacija može se realizirati na različite načine, ali i korištenjem Connector API-ja za slanje HTTP zahtjeva koji smo već naučili. Prema tome, iskoristit ćemo istu implementaciju za slanje HTTP zahtjeva na poslužitelj (mikroservis) koji služi za slanje e-mail poruka.

Servis za slanje e-mail poruka može biti bilo koji, primjerice *Nodemailer*, *SendGrid*, *Mailgun*, itd. U našem slučaju, koristit ćemo [Email.js](#) servis. Bez obzira što je *Email.js* primarno namijenjen za korištenje u frontend aplikacijama, možemo ga koristiti i u backend aplikacijama preko REST API-ja.

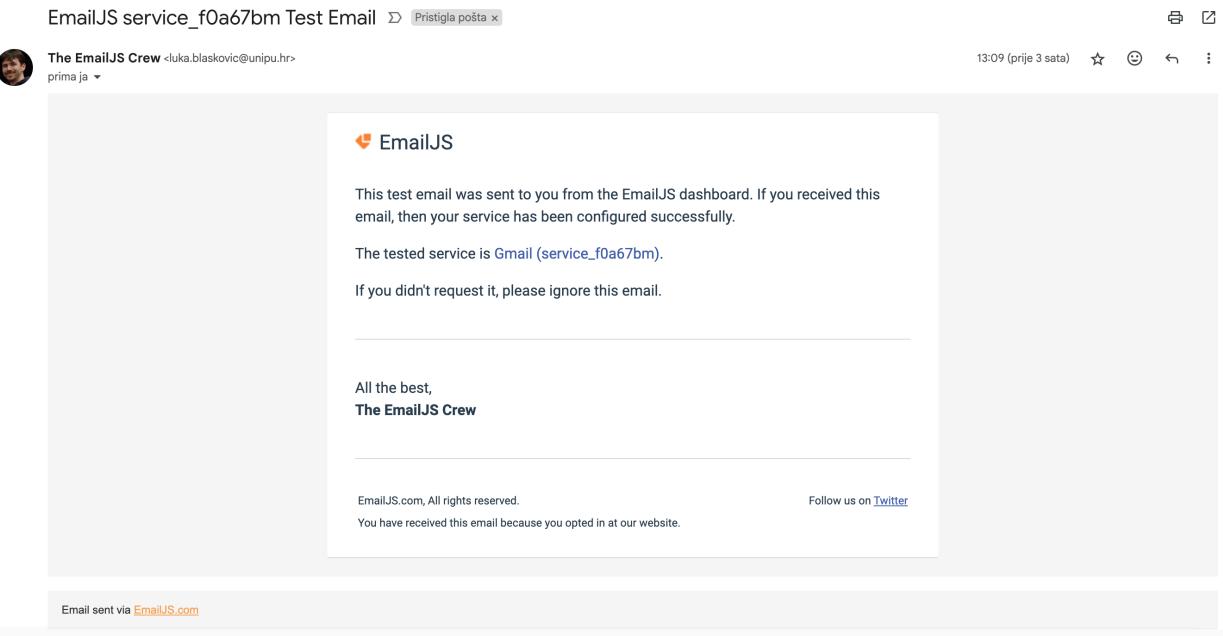
Registrirat ćemo novi račun na *Email.js* servisu, a nakon toga povezati naš Gmail račun putem kojeg ćemo slati e-mail (najjednostavniji način). U produkciji, vjerojatno nećete htjeti koristiti osobni Gmail račun za slanje e-mail poruka, već **SMTP** (eng. *Simple Mail Transfer Protocol*) poslužitelj vaše organizacije ili neki od gore navedenih servisa.

Izradite novi račun na [Email.js](#) servisu i povežite svoj **Gmail** račun.



Prilikom povezivanja morate dozvoliti pristup vašem Gmail računu i omogućiti **Slanje e-poruka u vaše ime**. Jednom kad povežete račun, možete poslati testnu poruku kako biste se uvjерili da je sve ispravno konfiguirano.

Trebali biste dobiti sljedeći email u vašem sandučiću koji ste poslali "sami sebi":



2.1 Priprema poslužitelja za automatsko slanje e-maila

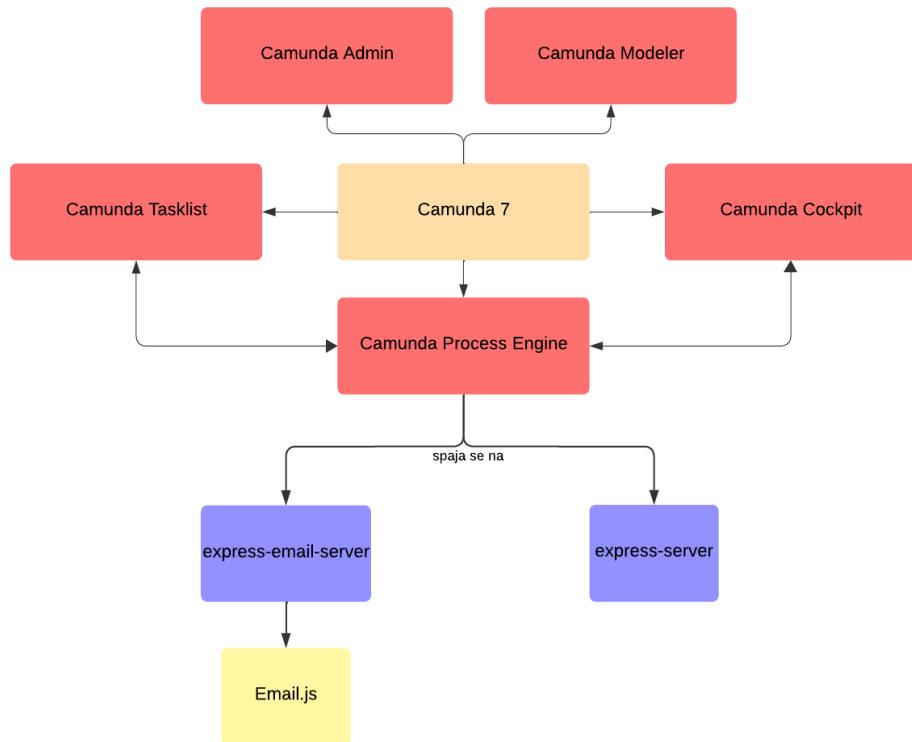
Sljedeći korak je implementacija Express.js poslužitelja koji će služiti kao **posrednik između Camunda Enginea i Email.js servisa**. Ideja je sljedeća:

1. Camunda Engine šalje HTTP POST zahtjev na naš Express.js poslužitelj s podacima o korisniku
2. Express.js poslužitelj obrađuje zahtjev i šalje e-mail poruku korisniku putem *Email.js* servisa
3. *Email.js* servis šalje e-mail poruku korisniku

Izradite novi direktorij `express-email-server`, inicijalizirajte novi Node.js projekt i instalirajte Express.js:

Naravno, moguće je iskoristiti postojeći Express.js poslužitelj koji smo definirali ranije, međutim kako su same procesne aplikacije bazirane na raspodijeljenoj arhitekturi, nije loše držati se te paradigme i odvojiti poslužitelje.

Prije nego nastavimo, nije loše pogledati kako do sada izgleda **raspodijeljena arhitektura naše procesne aplikacije**:



Ilustracija raspodijeljene arhitekture procesne aplikacije bazirane na Camundi 7

Naš `express-email-server` poslužitelj slušat će na portu `3000`:

```
// express-email-server/index.js

import express from "express";
import cors from "cors";

const PORT = 3000;

const app = express();
app.use(express.json());
app.use(cors());

app.get("/", (req, res) => {
  console.log("Zahtjev primljen!");
  res.status(200).send("Pozdrav iz express-email-server poslužitelja!");
});

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

Kako koristimo Email.js servis na poslužiteljskoj strani, isto moramo dozvoliti u postavkama Email.js-a.

Otvorite sljedeću poveznicu: <https://dashboard.emailjs.com/admin/account/security>

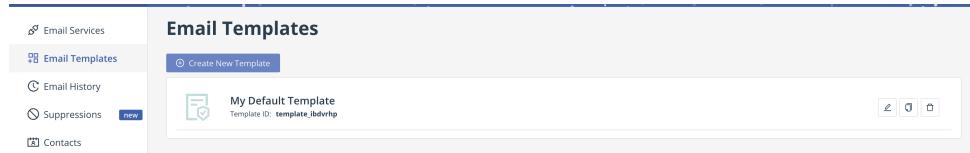
U odjeljku **API Settings** omogućite: "Allow EmailJS API for non-browser applications" i spremite promjene.

2.2 Email.js - priprema predloška

Prije nego krenemo s pisanjem koda, kreirat ćemo predložak e-mail poruke koji ćemo koristiti za slanje e-mail poruke korisniku.

Predložak (*eng. Template*) možete kreirati na sljedećoj povezničkoj stranici: <https://dashboard.emailjs.com/admin/templates> ili odabirom **Email Templates** u glavnom izborniku.

Kliknite na **Create New Template** i izradite novi predložak. Nazovite ga `process-app-template`.



Email.js - kreiranje novog predloška e-mail poruke (**Email Templates**)

Otvorite postavke predloška (*eng. Settings*) i promijenite njegov naziv, potom kopirajte negdje **ID predloška** (*eng. Template ID*) jer će nam trebati kasnije. Pohranite promjene.



U **Content** odjeljku možete definirati sadržaj e-mail poruke, uključujući naslov (*eng. Subject*) te sadržaj emaila (*eng. Content*).

Osim vizualnog editora, možete koristiti i HTML editor za preciznije uređivanje sadržaja e-mail poruke.

Odaberite **Edit Content** i uredite sadržaj emaila. Poslat ćemo korisniku jednostavni email s podacima o korisniku (ime, prezime, email, username) koje smo dohvatili iz procesa.

Kako su podaci o korisniku pohranjeni u procesnim varijablama, koje se proslijeđuje u tijelu HTTP POST zahtjeva na ovaj poslužitelj, moramo ih dohvatiti i koristiti u predlošku e-mail poruke. Za to koristimo tzv. **placeholders** koje pišemo duplim vitičastim zagradama `{ }`.

Dakle, naše varijable možemo koristiti na sljedeći način:

```
Ime: {{ime}}
Prezime: {{prezime}}
Korisničko ime: {{username}}
E-mail: {{email}}
```

Primjer predloška:

Pozdrav!

Šaljemo Vam podatke o korisniku:

Ime: {{ime}}
Prezime: {{prezime}}
Email: {{email}}
Username: {{username}}

Lijepi pozdrav i ugodan dan,

Vaša Camunda!

Kao naslov emaila možemo postaviti: "Camunda: Podaci o korisniku"

Za pošiljatelja navodite vaš email koji ste koristili prilikom registracije Gmail servisa, a kao naziv pošiljatelja možete staviti "Moja Camunda" ili sl.

Sve skupa trebalo bi izgledati ovako:

process-app-template

[Playground](#) [Test It](#)

[Save](#)

[Content](#) [Auto-Reply](#) [Attachments](#) [Contacts](#) [Settings](#)

Subject *

Camunda: Podaci o korisniku

Content *

Desktop Mobile [Edit Content](#)

Pozdrav!

Šaljemo Vam podatke o korisniku:

Ime: {{ime}}

Prezime: {{prezime}}

Email: {{email}}

Username: {{username}}

Lijepi pozdrav i ugodan dan,
Vaša Camunda!

To Email *

lukablasovic2000@gmail.com

From Name

Moja Camunda

From Email *

Use Default Email Address [?](#)

Reply To

Bcc

Cc

Primjer Email.js predloška e-mail poruke

Spremite promjene.

2.3 Implementacija slanja e-mail poruke

Sada kada smo pripremili predložak e-mail poruke, možemo implementirati Express.js poslužitelj koji će služiti kao posrednik između *Camunda Enginea* i *Email.js* servisa.

Provjerite još jednom jeste li omogućili **Allow EmailJS API for non-browser applications** u postavkama *Email.js* servisa kako bi stvari radile kako treba.

Definirajte POST rutu `/send-email` koja će obrađivati zahtjeve za slanje e-mail poruka.

```
app.post("/send-email", async (req, res) => {
  res.send("Zahtjev primljen!");
});
```

Kako šaljemo zahtjeve na REST API *Email.js* servisa, moramo koristiti neku HTTP klijentsku biblioteku. U ovom slučaju, koristit ćemo [Axios](#).

Instalirajte **Axios** biblioteku:

```
npm install axios
```

Instalirat ćemo i `dotenv` biblioteku kako bismo mogli koristiti `.env` datoteku za pohranu osjetljivih podataka kao što su **API ključevi** *Email.js* servisa i ID predloška.

```
npm install dotenv
```

Izrađujemo `.env` datoteku u korijenskom direktoriju projekta i dodajemo sljedeće 4 varijable:

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

Environment varijable učitavmao koristeći `dotenv` biblioteku:

```
// express-email-server/index.js

import dotenv from "dotenv";
dotenv.config();
const { SERVICE_ID, TEMPLATE_ID, PUBLIC_KEY, PRIVATE_KEY } = process.env;
```

Unutar `/send-email` rute definirat ćemo `try-catch` blok te unutar njega definirati Axios kod za slanje POST zahtjeva na *Email.js* servis te obradu eventualnih grešaka.

URL endpointa gdje šaljemo POST zahtjev je sljedeći: <https://api.emailjs.com/api/v1.0/email/send>

```
// express-email-server/index.js
```

```

app.post("/send-email", async (req, res) => {
  try {
    const response = await axios.post(
      "https://api.emailjs.com/api/v1.0/email/send", // Email.js REST API endpoint
      {
        service_id: SERVICE_ID,
        template_id: TEMPLATE_ID,
        user_id: PUBLIC_KEY,
        accessToken: PRIVATE_KEY,
      },
      {
        headers: {
          "Content-Type": "application/json", // uključujemo Content-Type zaglavlje
        },
      }
    );
    // obrada uspješnog odgovora
    res.status(200).json({
      message: "Email uspješno poslan!",
      data: response.data,
    });
  } catch (error) {
    // obrada greške
    console.error(
      "Greška prilikom slanja emaila: ",
      (error.response && error.response.data) || error.message
    );
    res.status(500).json({
      error: "Greška prilikom slanja emaila!",
      details: (error.response && error.response.data) || error.message,
    });
  }
});

```

Za kraj, **moramo proslijediti podatke o korisniku u tijelu POST zahtjeva**. Kako su podaci o korisniku pohranjeni u procesnim varijablama, moramo ih dohvatiti i koristiti u tijelu zahtjeva.

```

// express-email-server/index.js

app.post("/send-email", async (req, res) => {
  try {
    const { ime, prezime, email, username } = req.body; // dohvaćanje podataka o korisniku
    const response = await axios.post(
      "https://api.emailjs.com/api/v1.0/email/send",
      {
        service_id: SERVICE_ID,
        template_id: TEMPLATE_ID,
        user_id: PUBLIC_KEY,
        accessToken: PRIVATE_KEY,
        // podaci koji se koriste u predlošku e-mail poruke
      }
    );
    res.status(200).json({
      message: "Email uspješno poslan!",
      data: response.data,
    });
  } catch (error) {
    console.error("Greška prilikom slanja emaila: ", error.message);
    res.status(500).json({
      error: "Greška prilikom slanja emaila!",
      details: error.message,
    });
  }
});

```

```

    template_params: {
      ime: ime,
      prezime: prezime,
      email: email,
      username: username,
    },
  },
  {
    headers: {
      "Content-Type": "application/json",
    },
  }
);
res.status(200).json({
  message: "Email uspješno poslan!",
  data: response.data,
});
} catch (error) {
  console.error(
    "Greška prilikom slanja emaila: ",
    (error.response && error.response.data) || error.message
  );
res.status(500).json({
  error: "Greška prilikom slanja emaila!",
  details: (error.response && error.response.data) || error.message,
});
}
});

```

Testirajte ovu POST rutu korištenjem nekog HTTP klijenta (npr. [Postman](#), [Insomnia](#), [ThunderClient](#)). Primjer tijela zahtjeva:

```
{
  "ime": "Pero",
  "prezime": "Perić",
  "email": "pperic@gmail.com",
  "username": "pperic123"
}
```

Ako ste sve točno napravili, email bi se trebao poslati na vašu e-mail adresu, a kao rezultat dobiti ćete sljedeće tijelo odgovora i statusni kod 200:

```
{
  "message": "Email uspješno poslan!",
  "data": "OK"
}
```

Primjer zaprimljenog e-maila:

Camunda: Podaci o korisniku ✉ Pristigla pošta

 Moja Camunda <luka.blaskovic@unipu.hr>
 prima ja ▾

Pozdrav!

Šaljemo Vam podatke o korisniku:

Ime: Pero
Prezime: Perić
Email: pperic@gmail.com
Username: pperic123

Lijepi pozdrav i ugodan dan,
Vaša Camunda!

Email sent via [EmailJS.com](#)

✉ Odgovor ✉ Proslijedi ✉

Primjer e-mail poruke koju smo poslali koristeći *Email.js* servis

To je to! Uspješno smo implementirali Express.js poslužitelj za slanje e-mail poruka putem *Email.js* servisa

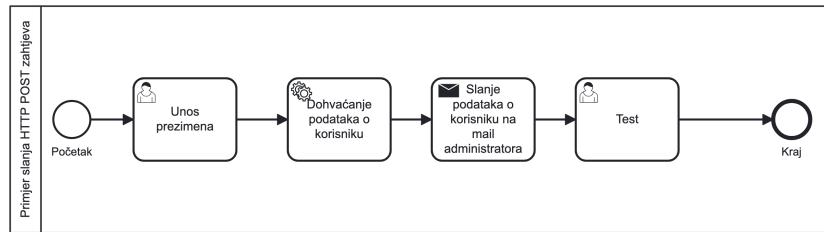


Sljedeći korak je integracija ovog poslužitelja s *Camunda Engineom*, preciznije [Connector](#) implementacija na [Send Task](#) elementu.

2.4 Definiranje Send Task aktivnosti

Postupak definiranja `Send Task` elementa je identičan kao i kod `Service Task` elementa. Koristeći Connector API implementirat ćemo na jednak način slanje POST zahtjeva na poslužitelj `express-email-server`.

Nadogradit ćemo proces "Primjer slanja HTTP POST zahtjeva" `Send Taskom` - "Slanje podataka o korisniku na mail administratora", gdje prepostavljamo da smo administrator mi, odnosno email koji smo definirali na Email.js servisu.



Procesna definicija s dodanim `Send Task` elementom

Odabiremo `Send Task` element te jednako kao i kod servisnog zadatka, pod **Implementation** odabiremo `Connector` te kao **Connector Id** unosimo `http-connector`.

Dalje, dodajemo sljedeće `Connector inputs`:

- **Local variable name** = `url`, **Assignment type** = `String or expression`, **Value** = `http://host.docker.internal:3000/send-email`
- **Local variable name** = `method`, **Assignment type** = `String or expression`, **Value** = `POST`
- **Local variable name** = `headers`, **Assignment type** = `Map`, **Map entries** (Key = `Content-Type`, Value = `application/json`)
- **Local variable name** = `payload`, **Assignment type** = `String or expression`, **Value** = `{"ime": "${ime}", "prezime": "${prezime}", "email": "${email}", "username": "${username}"}`

Pazite da se imena procesnih varijabli podudaraju s imenima varijabli koje referenciramo Expressionom u `payload` varijabli!

The screenshot shows the Camunda BPMN editor interface. On the left is the process diagram with the 'Slanje podataka o korisniku na mail administratora' task highlighted. On the right is the 'SEND TASK' configuration panel, which includes fields for 'Connector inputs' (Map, Content-Type: application/json), 'payload' (Local variable name: payload, Assignment type: String or expression, Value: {"ime": "\${ime}", "prezime": "\${prezime}", "email": "\${email}", "username": "\${username}"}), and a note to start typing "\${}" to create an expression.

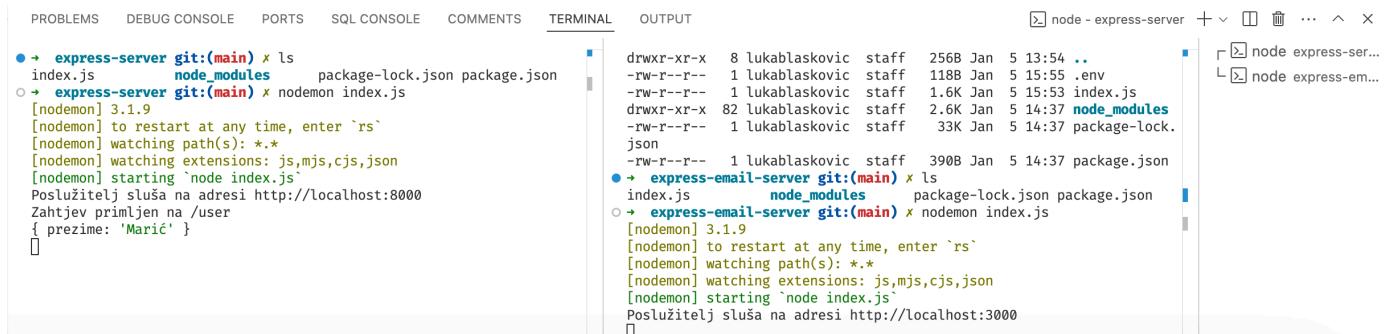
Dodavanje `Connector inputs` za `Send Task` element, primjer definiranja tijela zahtjeva i zaglavlja

Kao odgovor, dovoljno nam je samo pohraniti statusni kod odgovora u **izlaznu procesnu varijablu**

Connectora (`Connector outputs`):

- **Local variable name** = `emailStatus`, **Assignment type** = `String or expression`, **Value** =
 `${statusCode}`

Prije nego testirate procesnu definiciju, provjerite da ste pokrenuli oba poslužitelja (`express-server` i `express-email-server`) te da su dostupni na odgovarajućim portovima. Ako koristite VS Code, oba poslužitelja možete pokrenuti u zasebnim terminalima, najpregleđnije je odvojiti ih u zasebne radne prozore (`Terminal -> Split Terminal`).



```
PROBLEMS DEBUG CONSOLE PORTS SQL CONSOLE COMMENTS TERMINAL OUTPUT node - express-server + ▾ ▷ node express-ser... ▾ node express-em...  
● ➔ express-server git:(main) ✘ ls  
index.js node_modules package-lock.json package.json  
○ ➔ express-server git:(main) ✘ nodemon index.js  
[nodemon] 3.1.9  
[nodemon] to restart at any time, enter `rs`  
[nodemon] watching path(s): *.*  
[nodemon] watching extensions: js,mjs,cjs,json  
[nodemon] starting `node index.js`  
Poslužitelj sluša na adresi http://localhost:8000  
Zahtjev primljen na /user  
{ prezime: 'Marić' }  
○ ➔ express-email-server git:(main) ✘ ls  
index.js node_modules package-lock.json package.json  
○ ➔ express-email-server git:(main) ✘ nodemon index.js  
[nodemon] 3.1.9  
[nodemon] to restart at any time, enter `rs`  
[nodemon] watching path(s): *.*  
[nodemon] watching extensions: js,mjs,cjs,json  
[nodemon] starting `node index.js`  
Poslužitelj sluša na adresi http://localhost:3000
```

Redeployajte procesnu definiciju i pokušajte ponovno pokrenuti procesnu instancu.

Ako ste sve ispravno definirali, nakon što unesete prezime u `User Task` elementu "Unos prezimena", dešava se sljedeće:

1. Poziva se servisni zadatak koji pronađe korisnika na temelju prezimena u servisu `express-server`
2. Dohvaćeni podaci se pohranjuju u odgovarajuće procesne varijable te se koristeći `Send Task` element šalju na poslužitelj `express-email-server` koji šalje e-mail poruku korisniku
3. Ako je poslužitelj ispravno implementiran, trebali biste primiti e-mail poruku na vašu e-mail adresu

Primjerice, unijeli smo prezime "Marić" i dohvatali podatke o korisniku Marku Mariću. Nakon toga, podaci o korisniku su poslati na našu e-mail adresu.

Camunda Cockpit Processes Decisions Human Tasks More ▾ Demo Der

Dashboard » Processes » connector_POST : 58469790-cb78-11ef-945f-0242ac110002 : Runtime

Information Filter

Instance ID: 58469790-cb78-11ef-945f-0242ac1...

Business Key: null

Definition Version: 1

Definition ID: connector_POST:1:568ba11f-cb78-1...

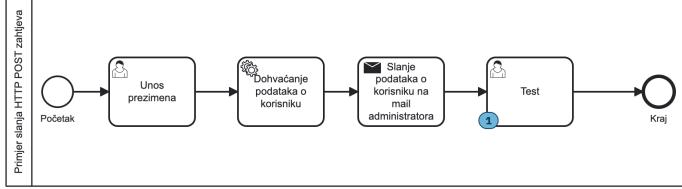
Definition Key: connector_POST

Definition Name: connector_POST

Tenant ID: null

Deployment ID: 56711dfd-cb78-11ef-945f-0242ac11...

Super Process Instance ID: null

Diagram: 

```

graph LR
    Start((Početak)) --> Unos[Unos prezimena]
    Unos --> Dohvacanje[Dohvaćanje podataka o korisniku]
    Dohvacanje --> Slanje[Slanje podataka o korisniku na mail administratora]
    Slanje --> Test[Test]
    Test --> Kraj((Kraj))
    
```

Variables Incidents Called Process Instances User Tasks Jobs External Tasks

Add criteria

Name	Type	Value	Scope	Actions
email	String	mmaric@gmail.com	connector_POST	
emailStatus	Integer	200	connector_POST	
ime	String	Marko	connector_POST	
odgovor	String	{"ime": "Marko", "prezime": "Marić", "use...}	connector_POST	
prezime	String	Marić	connector_POST	
statusni_kod	Integer	200	connector_POST	
username	String	marko.maric	connector_POST	

Primjer rezultata procesne instance u Cockpitu