Univerza v Mariboru

Fakulteta za elektrotehniko,
računalništvo in informatiko

Luka Mernik

Razvoj orodja za orkestracijo mikrostoritev v spletnih trgovinah

Diplomsko delo

Maribor, junij 2023

Univerza v Mariboru

Fakulteta za elektrotehniko,
računalništvo in informatiko

Luka Mernik

Razvoj orodja za orkestracijo mikrostoritev v spletnih trgovinah

Diplomsko delo

Maribor, junij 2023

Razvoj orodja za orkestracijo mikrostoritev v spletnih trgovinah

Diplomsko delo

Študent:	Luka Mernik
Študijski program:	Visokošolski strokovni program Računalništvo in informacijske tehnologije
Mentor:	izr. prof. dr. Tomaž Kosar, univ. dipl. inž. rač. in inf.
Somentor:	asist. Sandi Majninger, mag. inž. rač. in inf. tehnol.
Zunanji delovni somentor:	Aleš Flajšman, univ. dipl. inž. rač. in inf.
Lektorica:	dr. Ana Koritnik, prof. slov.

Zahvala

Zahvalil bi se mentorju dr. Tomažu Kosarju in asist. Sandiju Majningerju za vso pomoč, koordinacijo in svetovanje pri izdelovanju diplomskega dela.

Zahvalil bi se podjetju Comtron, d. o. o., ki mi je omogočilo izdelovanje produkta TRONintegration in uporabo le-tega za diplomsko delo. Zahvala gre tudi zunanjemu delovnemu somentorju Alešu Flajšmanu, ki mi je priskočil na pomoč v primeru težav in me usmeril na pravo pot.

Zahvalil bi se še svoji družini in dekletu, saj so me podpirali skozi celoten študij in mi omogočili nemoteno izdelovanje diplomskega dela.

Razvoj orodja za orkestracijo mikrostoritev v spletnih trgovinah

Ključne besede: integracija, diagram, korak, Angular, Syncfusion

UDK: 004.42:004.77(043.2)

Povzetek

V diplomskem delu smo opisali razvoj produkta TRONintegration. Produkt služi kot orodje za izdelavo integracije dveh sistemov (na primer TRONoffice in spletne trgovine Shopify). Orodje v osnovi temelji na podobnih funkcionalnostih obstoječega produkta Tray.io, vendar z veliko več možnostmi in dodelavami ter samostojno implementacijo. Opisali smo uporabljene tehnologije, raziskali smo knjižnico Syncfusion, ki nam je zelo pripomogla pri vizualnem prikazu diagrama. Ta najbolj pomembni del predstavlja potek same integracije in je sestavljen iz gradnikov: vozlišč in povezav. Raziskali smo področje orkestracije mikrostoritev in nato razvijali produkt po fazah ter ga na koncu testirali. S pomočjo dokumentacije smo uspešno izdelali orodje, ki omogoča izdelavo integracije dveh sistemov na intuitiven vizualen način in zanj ni potrebno veliko predznanja.

Development of a tool for microservices orchestration in online stores

Keywords: integration, diagram, step, Angular, Syncfusion

UDC: 004.42:004.77(043.2)

Abstract

In the thesis, we described the development of the TRONintegration product. The product serves as a tool for creating integration between two systems (for example, TRONoffice and the Shopify web store). The tool is primarily based on similar functionalities as the existing Tray.io product but with many more options, enhancements, and standalone implementation. We described the technologies used and explored the Syncfusion library, which greatly helped us with the visual representation of the diagram. This most important part presents the flow of the integration itself and is composed of components: nodes and connections. We explored the field of microservices orchestration and then developed the product in phases, testing it in the end. With the help of documentation, we successfully created a tool that allows the creation of integration between two systems in an intuitive visual manner, requiring minimal prior knowledge.

Univerza v Mariboru
Fakulteta za elektrotehniko, računalništvo in informatiko
Koroška cesta 46
2000 Maribor, Slovenija

FERI

IZJAVA O AVTORSTVU ZAKLJUČNEGA DELA

Ime in priimek študent-a/-ke: Luka Mernik
Študijski program: Računalništvo in informacijske tehnologije (VS)
Naslov zaključnega dela: Razvoj orodja za orkestracijo mikrostoritev v spletnih trgovinah

Mentor: izr. prof. dr. Tomaž Kosar, univ. dipl. inž. rač in inf.
Somentor: asist. Sandi Majninger, mag. inž. rač. in inf. tehnol.
Zunanji delovni somentor: Aleš Flajšman, univ. dipl. inž. rač. in inf.

Podpisan-i/-a študent/-ka Luka Mernik

- izjavljam, da je zaključno delo rezultat mojega samostojnega dela, ki sem ga izdelal/-a ob pomoči mentor-ja/-ice oz. somentor-ja/-ice;
- izjavljam, da sem pridobil/-a vsa potrebna soglasja za uporabo podatkov in avtorskih del v zaključnem delu in jih v zaključnem delu jasno in ustrezno označil/-a;
- na Univerzo v Mariboru neodplačno, neizključno, prostorsko in časovno neomejeno prenašam pravico shranitve avtorskega dela v elektronski obliki, pravico reproduciranja ter pravico ponuditi zaključno delo javnosti na svetovnem spletu preko DKUM; sem seznanjen/-a, da bodo dela deponirana/objavljena v DKUM dostopna široki javnosti pod pogoji licence Creative Commons BY-NC-ND, kar vključuje tudi avtomatizirano indeksiranje preko spleta in obdelavo besedil za potrebe tekstovnega in podatkovnega rudarjenja in ekstrakcije znanja iz vsebin; uporabnikom se dovoli reproduciranje brez predelave avtorskega dela, distribuiranje, dajanje v najem in priobčitev javnosti samega izvirnega avtorskega dela, in sicer pod pogojem, da navedejo avtorja in da ne gre za komercialno uporabo;
- dovoljujem objavo svojih osebnih podatkov, ki so navedeni v zaključnem delu in tej izjavi, skupaj z objavo zaključnega dela.

Uveljavljam permisivnejšo obliko licence Creative Commons: CC BY-NC.ND 4.0 (navedite obliko)

Kraj in datum: Maribor, 30.05.2023

Podpis študent-a/-ke:

Luka Mernik Digitally signed by Luka Mernik Date: 2023.05.30 19:32:09 +02'00'

KAZALO VSEBINE

KAZALO SLIK

KAZALO IZSEKOV KODE

Uporabljene kratice

ACID – atomarnost, doslednost, izoliranost in trajnost (angl. Atomicity, Consistency, Isolation and Durability)

API – vmesnik uporabniškega programa (angl. Application Programming Interface)

CRUD – ustvarjanje, branje, posodabljanje in brisanje (angl. Create, Read, Update and Delete)

CSS – predloge, ki določajo izgled spletnih strani (angl. Cascading Style Sheets)

DTO – objekt za prenos podatkov (angl. Data transfer object)

GUID – globalni unikatni identifikator (angl. Globally unique identifier)

HTML – jezik za označevanje nadbesedila (angl. HyperText Markup Language)

HTTP – protokol za prenos hiperteksta (angl. Hypertext Transfer Protocol)

HTTPS – protokol za varen prenos hiperteksta (angl. Hypertext Transfer Protocol Secure)

JSON – objektna notacija za JavaScript (angl. JavaScript Object Notation)

npm – upravitelj paketov za Node (angl. Node Package Manager)

REST – aktualni prenos stanja (angl. REpresentational State Transfer)

SQL – strukturiran povpraševalni jezik za delo s podatkovnimi bazami (angl. Structured Query Language)

ts – TypeScript (angl. TypeScript)

1 UVOD

1.1 Opis problema

Podjetje Comtron, d. o. o., v nadaljevanju Comtron, razvija produkt, imenovan TRONoffice. Ta strankam omogoča upravljanje z zalogo, financami, avtomatizacijo poslovnih funkcij in ostalih potrebnih stvari za delovanje podjetja. Obstoječe stranke, ki so uporabljale produkt zgolj za fizična podjetja, so imele željo po vzpostavitvi spletne trgovine, ki bi jo podjetje ustvarilo s pomočjo ene izmed platform za ustvarjanje spletne trgovine, na primer Shopify. Na tem mestu se je pojavila potreba po vzpostavitvi komunikacije med že uporabljenim produktom TRONoffice in na novo ustvarjeno spletno trgovino, saj je za uspešno delovanje treba imeti ažurirane podatke na obeh straneh. Zaradi raznih razlik med produktom in spletno trgovino ter zaradi ločene podatkovne baze se je pojavila potreba po izmenjavi podatkov, kjer se pojavi težava. Ena izmed rešitev je bila integracija produkta TRONoffice in spletne trgovine. Rešitve ni potrebovala zgolj ena stranka, temveč veliko več. Pojavila se je ideja o orodju, ki smo ga razvili in omogoča izgradnjo integracije. Pri tem nismo zgolj omejeni na določeni produkt ali platformo, s pomočjo katere je ustvarjena spletna trgovina. Znatno pridobimo, saj ni treba za vsako stranko posebej prilagajati in izdelovati integracije, temveč je možna pouporaba s prilagoditvami. Po opravljeni raziskavi smo ugotovili, da obstaja kar nekaj obstoječih produktov, ki omogočajo izgradnjo integracije (Tray.io [22], CDATA [6], API2CART [6]). Ugotovili smo, da je zelo malo takih, ki bi omogočali prilagajanje glede na potrebe oziroma gre za dokaj omejene produkte in cenovno zelo drage. Po opravljeni raziskavi je najprimernejši produkt bil Tray.io, ki je za uporabo dokaj enostaven, vendar cenovno zelo drag in omejen. Zaradi težav in omejitev smo sami razvili produkt, ki služi kot orodje za izdelavo integracij v osnovi za povezavo produkta TRONoffice in spletne trgovine. Podprli smo tudi povezavo med dvema produktoma, ki nista določena in sta neodvisna drug od drugega.

1.2 Cilji

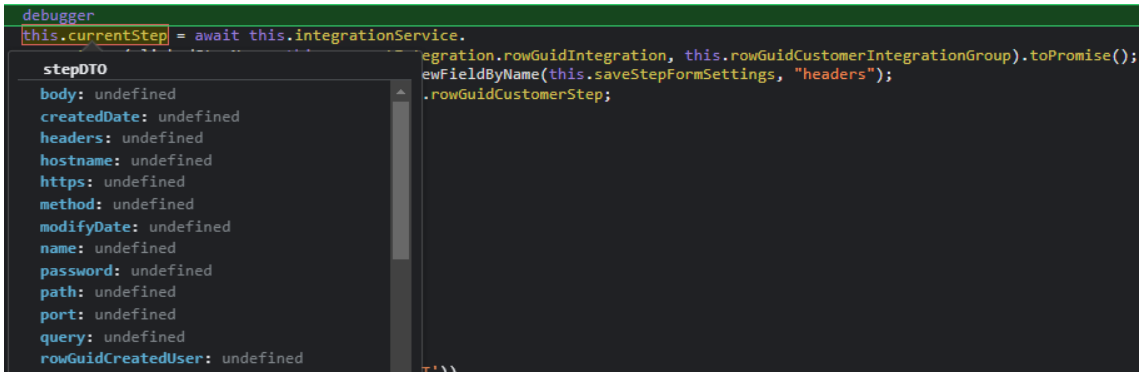
Razvili smo produkt, ki služi kot orodje za izgradnjo integracije med dvema sistemoma (na primer integracija med produktom TRONoffice in spletno trgovino Shopify). Orodje je prijazno za uporabo in ni omejeno zgolj na integracijo med produktom TRONoffice in spletno trgovino Shopify, temveč je možno izdelati integracijo med produktom TRONoffice in spletno trgovino ne glede na izbrano platformo, ki omogoča vzpostavitev spletne trgovine. Podprli smo tudi izgradnjo integracije med dvema produktoma, ki nista vnaprej definirana. Dodatno smo podprli vse možne stvari, ki se lahko pojavijo kot potreba pri izgradnji integracije. Razvito orodje je sila preprosto za uporabo in je namenjeno skoraj vsakomur. V osnovi ga uporablja oddelek za podporo strankam, ki izdelujejo integracije za le-te. Strankam je omogočeno, da lahko tudi same brez velikega predznanja oziroma s pomočjo jasnih in točnih navodil izdelajo potrebno integracijo. Orodje je izdelano tako, da je vizualno pregledno za uporabo. V primeru kakršnih koli napak se pojavi pomoč v obliki pojavnih sporočil, kar uporabniku pomaga pri obravnavi danih napak, ki jih lahko enostavno popravi oziroma odpravi. Skozi sam razvoj produkta smo spoznali sodobne pristope in postopke, ki so trenutno aktualni. V glavnem smo dosegli zastavljene cilje in izdelali produkt TRONintegration, ki se bo v nadaljevanju še razvijal.

2 UPORABLJENE TEHNOLOGIJE

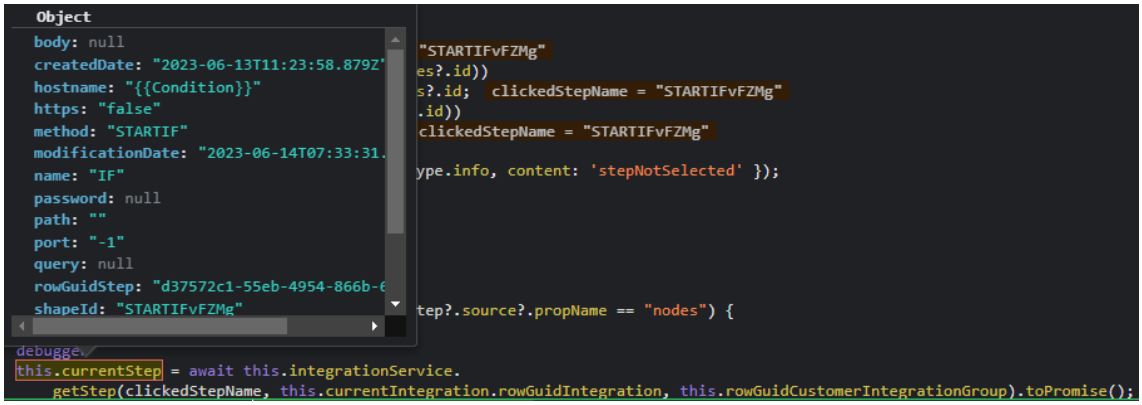
Vse uporabljene tehnologije so bile izbrane skrbno in namenoma. Podjetje trenutno uporablja vse navedene tehnologije, razen knjižnice Syncfusion. Ta knjižnica je bila dodana zaradi potrebe po vizualnem prikazu in izgradnji diagramov. Izkazala se je kot odlična izbira.

2.1 Angular

Čelni del produkta TRONintegration smo razvijali s pomočjo ogrodja Angular [4]. Gre za odprtokodno ogrodje, ki je namenjeno razvoju spletnih aplikacij in je v zadnjem času zelo pogosto uporabljeno pri razvoju sodobnih spletnih aplikacij. Njegova priljubljenost in pogostost uporabe sta seveda odvisni od številnih dejavnikov: vrste projekta, zahteve stranke, ekipe in znanja razvijalcev. Ogrodje omogoča razvoj kompleksnih aplikacij in igra pomembno vlogo pri razvoju hibridnih ter mobilnih aplikacij s pomočjo platforme Ionic [8]. V osnovi gre za razširjen JavaScript, ki omogoča razvoj enostranskih kot tudi hibridnih aplikacij. Posamezni delčki aplikacije so sestavljeni iz komponent, ki so med seboj povezane. Prednost uporabe je, da omogoča povezavo logike (TypeScript), prikaza (HTML) in stilizacijo (CSS) kot dinamične funkcionalnosti. Največja prednost ogrodja je, da omogoča dvosmerno vezanje podatkov (angl. two-way data binding). Slednje omogoča samodejno posodabljanje brez posredovanja. To lahko naredimo s pomočjo @Input in @Output ukaza, glede na to, ali želimo prejemati ali pošiljati podatke. V primeru napak lahko s pomočjo ukaza »debugger« (Slika 2.1), ki ga zapišemo v kodi, preprosto testiramo in preverimo delovanje komponent in elementov aplikacije (Slika 2.2).



Slika 2.1: Uporaba ukaza »debugger«



Slika 2.2: Odpravljanje napak in pregled vrednosti spremenljivke s pomočjo ukaza »debugger«

2.2 Knjižnica Kendo UI

Podjetje Telerik je razvilo knjižnico Kendo UI [10], ki ponuja širok nabor uporabniških komponent za vizualizacijo (gumbi, grafikoni, obrazci, meniji, koledarji in še veliko različnih komponent), ki so vizualno zelo dodelane in lahke za uporabo. Zraven teh ponuja še funkcionalnosti komponent, ki služijo za upravljanje uporabniškega vmesnika. Knjižnica omogoča preprostejšo in hitrejšo izgradnjo sodobnih spletnih aplikacij. Podpira avtomatsko prilagajanje velikosti aplikacije glede na velikost naprave, na kateri je prikazana, in omogoča podporo za različne brskalnike. Zgrajena je s pomočjo jezika HTML, CSS in JavaScript. Uporaba knjižnice je mogoča in podprta na sledečih ogrodjih: Angular, ASP.NET, React in Vue.js. V osnovi gre za plačljivo knjižnico, vendar obstaja tudi brezplačna verzija, ki ponuja zgolj osnovne funkcionalnosti in komponente, primerne za

osnovno rabo. Za profesionalno rabo in posledično velik nabor različnih komponent je treba izbrati eno izmed plačljivih različic. Velika prednost knjižnice je, da razvijalcem precej olajša delo, saj je za lep in estetski izgled treba vložiti občutno manj truda in časa. Posledično spletna aplikacija izgleda izvrstno in privlačno za uporabnika.

2.3 PostgreSQL

PostgreSQL [16] temelji na relacijskem podatkovnem modelu in je namenjen upravljanju s podatkovno zbirko podatkov. Kot že samo ime pove, implementira standardni jezik SQL in ponuja številne napredne funkcije. Podatki so shranjeni v vnaprej definiranih tabelah, ki so med seboj povezane prek relacij oziroma povezav med entitetami. Omogoča več fleksibilnosti in prilagajanja napram ostalim podatkovnim zbirkam podatkov. ACID-lastnost bistveno doprinese pri zagotavljanju zanesljivega in varnega shranjevanja ter obdelave podatkov. Gre za izredno zmogljiv, odporen na napake in predvsem robusten odprtokodni sistem. Podpira dodajanje novih podatkovnih tipov in funkcij za procedure ter omogoča ustvarjanje lastnih razširitev v primeru potreb naprednih uporabnikov. Na voljo je za operacijske sisteme macOS, Linux in Windows. Namenjen je tako manjšim kot tudi večjim projektom. Podpira napredne funkcije, kot so transakcije in sprožilci. Replikacija podatkov prinaša veliko prednost, saj je v primeru napak možna obnovitev podatkov. Med drugim ponuja zelo dobro in obsežno dokumentacijo, ki je na voljo na spletu in tudi v tiskani obliki.

2.4 TypeORM

TypeORM [23] prinaša zelo enostavno uporabo skupaj z relacijskimi podatkovnimi bazami: MySQL, SQLite, MSSQL, PostgreSQL, in drugimi podatkovnimi bazami, ki omogočajo objektno-relacijsko preslikavo s pomočjo jezikov JavaScript in TypeScript. Namesto pisanja SQL-poizvedb omogoča upravljanje s podatkovno bazo s pomočjo objektov in metod. Veliko prednost prinaša vzpostavitev ujemanja med shemo podatkovne baze in objektnim modelom. Podpira preslikavo objektov v vrstice in

obratno, kar opazno olajša delo razvijalca. Omogoča enostavno zamenjavo podatkovne baze brez znatnega spreminjanja kode. Nadgradnja podatkovne sheme brez poseganja v podatkovno bazo ali z minimalnimi spremembami omogoča enostavno migracijo in občutno prihrani čas. Izdelava kompleksnejših poizvedb je mogoča s pomočjo sestavljanja enostavnih izrazov, ki skupaj tvorijo kompleksno poizvedbo. Transakcije omogočajo izvajanje več operacij hkrati, ki se lahko izvedejo vse ena za drugo ali se v primeru napake prekličejo. Posledično ne pride do delne transakcije, česar si tudi ne smemo dovoliti. Namenjen je predvsem razvijalcem, ki si želijo poenostaviti delo z relacijsko bazo podatkov.

2.5 Express

Za zaledni del produkta smo uporabili priljubljeno ogrodje, ki je namenjeno razvoju spletnih aplikacij s pomočjo platforme Node.js. Preprosta sintaksa omogoča razvoj spletnih strežnikov, ki učinkovito obravnavajo in odgovarjajo na HTTP-zahteve. Omogoča tako izdelavo enostranskih kot tudi večstranskih spletnih aplikacij. Enostavnost, zmogljivost in podpora za ustvarjanje RESTful API-klicev zelo doprinese k priljubljenosti ogrodja za razvoj spletnih aplikacij v Node.js. Izgradnja spletnih aplikacij s pomočjo ogrodja Express [7] zagotavlja razumljivo in hitro učljivo ogrodje, s pomočjo katerega je mogoče zgraditi robustne in enostavne spletne aplikacije. Omogoča dodajanje razširitev, kot so dodatne knjižnice in različni moduli. Med obdelovanjem zahtev omogoča izvajanje kode, zato se uporablja tudi kot vmesna programska oprema (angl. middleware). Razvijalci se radi odločajo za uporabo, saj je preprost, hiter in, kar je najbolj pomembno, prilagodljiv.

3 KNJIŽNICA SYNCFUSION

Knjižnico Syncfusion [20] je razvilo istoimensko podjetje in omogoča razvoj uporabniškega vmesnika, podobno kot knjižnica Kendo UI. Razlikuje se predvsem v drugače implementiranih komponentah in raznih specifikah. Ena izmed njih je komponenta diagram, ki je bila odločilna, da smo se odločili za izbiro knjižnice in jo dodali v produkt. Knjižnica omogoča hiter razvoj zmogljivih in vizualno dodelanih spletnih aplikacij. Podprta ogrodja in programski jeziki: .NET, JavaScript, Angular, React in druga. Ponuja širok nabor komponent, ki omogočajo razvoj namiznih, spletnih, mobilnih in strežniških aplikacij. Za uporabo knjižnice je treba pridobiti licenco. Te se med seboj razlikujejo in je na strani razvijalca oziroma podjetja, da izbere tisto, ki mu bo prišla v poštev.

3.1 Zakaj Syncfusion?

Omogoča širok nabor komponent in grafičnih elementov: koledar, tabela s podatki, urejanje slik, urejanje besedila, zemljevid in seveda najbolj pomembna ter uporabljena komponenta, ki je znatno doprinesla pri izboru ravno te knjižnice, je komponenta diagram. Ta je v osnovi sestavljena iz dveh osnovnih komponent: »ejs-symbolpalete« in »ejs-diagram«. Komponenta »ejs-symbolpalete« predstavlja nabor vseh možnih gradnikov, ki se lahko uporabijo v diagramu: vozlišča, predstavljena s pomočjo oblik, in povezave, predstavljene kot puščice. Komponenta »ejs-diagram« definira polje oziroma območje, v katerem se gradi diagram. Možno je spreminjati videz in funkcionalnosti skoraj vsem komponentam. Syncfusion sam po sebi ponuja zelo dobro podporo strankam in na voljo ima dokaj dobro dokumentacijo, ki nam je pomagala pri razvoju. Gre za aktualno knjižnico, ki sledi najnovejšim trendom in tehnologijam, ter pogosto izide nova verzija, kar omogoča posodobitev in posledično pridobitev novih stvari.

3.2 Komponente

Posamezne komponente lahko v diagram dodamo programsko (s pomočjo kode) tako, da točno definiramo, kje je postavljeno vozlišče ali povezava. Omogočeno je tudi spreminjanje vizualnega prikaza. Druga možnost je, da vnaprej določimo, kako bodo prikazana vozlišča in povezave, ter jih nato s pomočjo poteze »povleci in spusti« preprosto dodamo v sam diagram. Dodana vozlišča in povezave lahko nato znotraj diagrama prilagajamo (spreminjamo velikost in pozicijo).

3.2.1 Vozlišče, predstavljeno s pomočjo oblike

Knjižnica nam omogoča uporabo vnaprej določenih oblik za predstavitev vozlišča. Spodaj prikazani izsek kode (Izsek kode 3.1) prikazuje definiranje vozlišča, ki je v obliki petkotnika, na katerem je napis »TEST«.

```
public getBasicSteps() {
    let basicstep: any[] = [
        {
            id: 'TEST', shape: { type: 'Basic', shape: 'Pentagon' },
            annotations: [{ content: 'TEST' }]
        },
    ];
    return basicstep;
}
```

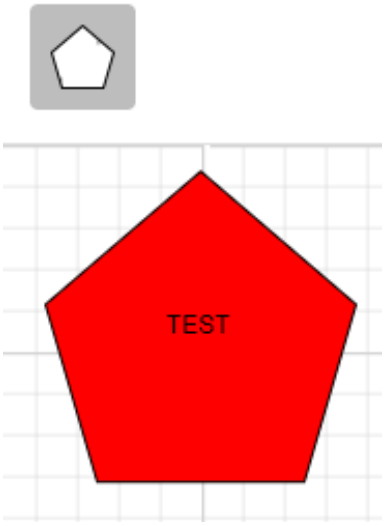
Izsek kode 3.1: Definiranje lastnosti oblike

Vozlišča lahko tudi spreminjamo po svoji želji in potrebi. V primeru, da želena oblika ne obstaja, jo lahko definiramo tudi sami. V osnovi lahko obliki določimo: tip, obliko, dodamo napis na obliko, prilagodimo velikost, spremenimo barvo, dodamo ime oblike in še veliko več možnega prilagajanja glede na potrebe in želje. Funkcija »getNodeDefaults« definira izgled oblike: velikost in barvo (Izsek kode 3.2).

```
public getNodeDefaults(node: NodeModel): NodeModel {
    node.height = 150;
    node.width = 150;
    node.style.fill = 'red';
    return node;
}
```

Izsek kode 3.2: Definiranje izgleda vozlišča

Spodaj (Slika 3.1) je prikazan rezultat ustvarjenega vozlišča, predstavljenega z obliko petkotnika.



Slika 3.1: Uporaba vozlišča v diagramu

Med drugim lahko tudi točno določimo pozicijo vozlišča v diagramu. Seveda sama knjižnica nam omogoča še veliko več prilagajanja oblik, vendar za naše potrebe je bilo navedeno dovolj. Vozlišče, ki je služilo za predstavitev koraka »pogoj IF«, smo izdelali sami, saj take oblike ni bilo predefinirane. Šlo je za skupek oblike z dvema puščicama. Na njih je pisalo »true« in false. Sam potek integracije je bil nato odvisen od izpolnjevanja pogoja, ki je bil določen znotraj koraka.

3.2.2 Povezava

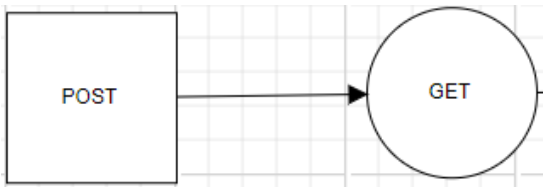
Podobno, kot velja za vozlišča (predstavljena z oblikami), velja tudi za povezave. Te lahko spreminjamo in prilagajamo glede na svoje potrebe ali uporabimo vnaprej določene ali

izdelamo čisto nove povezave (Izsek kode 3.3). V osnovi lahko povezavi določimo tip, obliko, dodamo napis, prilagodimo velikost, spremenimo barvo in dodamo ime. Povezave imajo podprto funkcionalnost, ki nam omogoča razbrati potek same povezave, torej od kod do kod gre povezava.

```
public getBasicSteps() {
    let basicstep: any[] = [
        {
            id: 'Link1', type: 'Orthogonal', sourcePoint: { x: 0, y: 0 },
            targetPoint: { x: 60, y: 60 }, targetDecorator: { shape: 'Arrow' }
        },
    ];
    return basicstep;
}
```

Izsek kode 3.3: Definiranje povezave in izgleda povezave

Želimo vedeti, kje je začetek povezave »sourcePoint« in konec povezave »targetPoint«, ki je predstavljena v paru tipa A->B. V omenjeni povezavi A predstavlja začetek oziroma izhodišče povezave in B predstavlja konec oziroma cilj povezave (Slika 3.2). V primeru, da začetek ali konec povezave ni določen, je tudi to ustrezno označeno. Torej s pomočjo te podprte funkcionalnosti lahko določimo, od katerega do katerega vozlišča gre posamezna povezava, kar nam v nadaljevanju pripomore pri določanju samega poteka. Pozicijo same povezave lahko določimo tudi programsko; tega nismo veliko uporabljali le pri koraku »pogoj IF«. Knjižnica omogoča ogromno prilagajanja povezav, vendar pri razvoju produkta ni bilo potrebe po tem.



Slika 3.2: Povezava v diagramu

3.3 Vgrajeni dogodki

Komponenta diagram nam v osnovi omogoča uporabo številnih dogodkov (angl. events), na primer: »double click« – gumb miške je bil pritisnjen dvakrat (Izsek kode 3.4), »selection change« – zgodila se je sprememba v diagramu, »connection change« – zgodila se je sprememba na gradniku povezava; te dogodke lahko dopolnimo s poljubno svojo logiko, glede na potrebe. Zraven tega je podprto tudi dodajanje svojih dogodkov. V razvoju produkta smo ponujene dogodke prilagodili in ustvarili svoje dogodke – zaradi specifičnih potreb. Uporabili smo dogodek »double click« pri kliku uporabnika na vozlišče v diagramu.

```
(doubleClick)="openPopup($event)"
```

Izsek kode 3.4: Dvoklik miške sproži funkcijo »openPopup«

Ob kliku se odpre pojavno okno (Izsek kode 3.5) za vnos podatkov. Dogodki so nam bili v veliko pomoč, saj so v osnovi beležili, na kaj smo kliknili, in tako smo lahko preprosto dostopali do ID-ja izbranega vozlišča.

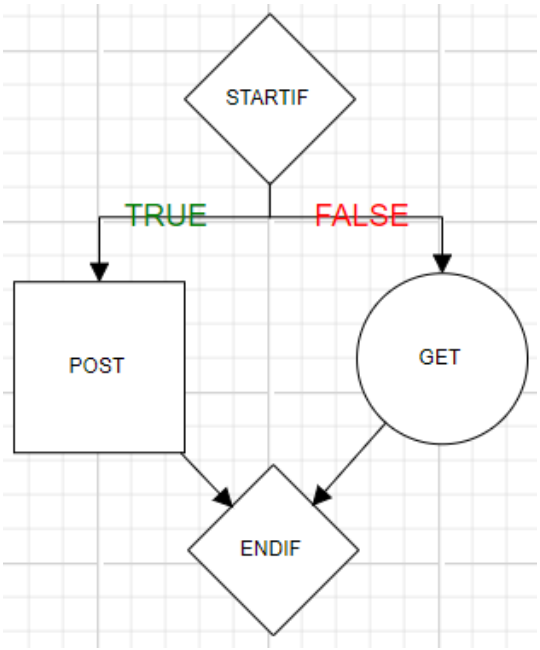
```
async openPopup(clickedStep: any) {
    let clickedStepName = undefined;
    if (!isNullOrUndefined(clickedStep?.source?.properties?.id))
        clickedStepName = clickedStep?.source?.properties?.id;
    else if (!isNullOrUndefined(clickedStep?.properties?.id))
        clickedStepName = clickedStep?.properties?.id;
    else {
        this.messageService.showMessage({ type: MessageType.info, content: 'stepNotSelected' });
        return;
    }
}
```

Izsek kode 3.5: Funkcija »openPopup«, ki odpre pojavno okno ob dvokliku miške

3.4 Izgradnja diagrama

Za izgradnjo diagrama je treba vnaprej določiti vozlišča in povezave ter dodelati dogodke do te mere, da bodo uporabni pri sami izgradnji diagrama. Gradnja diagrama poteka na čisto preprost način – tako, da s pomočjo miške povlečemo vozlišče ali povezavo, ki jo želimo v območje diagrama. Gradnik nato spustimo na določeno mesto v diagramu. Vse

gradnike, ki so v diagramu, lahko nato prosto premikamo po območju diagrama, spreminjamo velikost gradnika, ga kopiramo in izbrišemo po želji. Specifično za povezavo dodatno velja, da jo povežemo z vozliščem tako, da jo primemo na enem koncu in jo povežemo s poljubnim vozliščem, nato primemo še na drugem koncu in povežemo z drugim vozliščem (Slika 3.3).



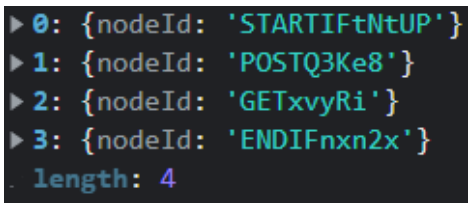
Slika 3.3: Primer diagrama

3.5 Problem diagrama

Komponenta diagram je v osnovi namenjena vizualnemu prikazu diagrama. Kljub veliki prednosti in enostavni uporabi se v ozadju izvaja logika le za vizualne dogodke. To nas je pripeljalo do problema in dodatnega dela, saj je bilo treba določene dogodke samostojno dodati. Struktura diagrama je shranjena v JSON-formatu. Največjo težavo nam je predstavljal potek oziroma pot diagrama. Iz strukture diagrama ali akcije dogodka, ki nam vrne povezave, uspemo dobiti le vsa vozlišča (Izsek kode 3.6 in Slika 3.4).

```
public exportShapes(): void {
    var nodes = this.diagram.nodes.map((nodes) => {
        return {
            nodeId: nodes.id
        };
    });
    console.log(nodes);
}
```

Izsek kode 3.6: Funkcija, ki prikaže vsa vozlišča, ki so v diagramu

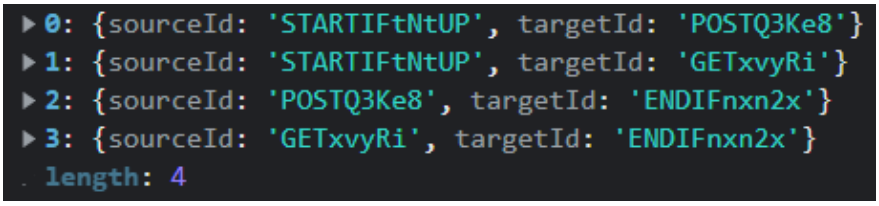


Slika 3.4: Vozlišča v diagramu

Uspe nam dobiti tudi povezave med oblikami v diagramu (Izsek kode 3.7 in Slika 3.5), ne pa tudi same poti. Enostavnih poti ni bilo težko podpreti in dobiti. Večji problem so nam predstavljale zahtevnejše poti in razni scenariji, ki se lahko pripetijo.

```
public mapNodePath(): void {
    var connections = this.diagram.connectors.map((connector) => {
        return {
            sourceId: connector.sourceID,
            targetId: connector.targetID
        };
    });
    console.log(connections);
}
```

Izsek kode 3.7: Funkcija, ki prikaže vse povezave v diagramu

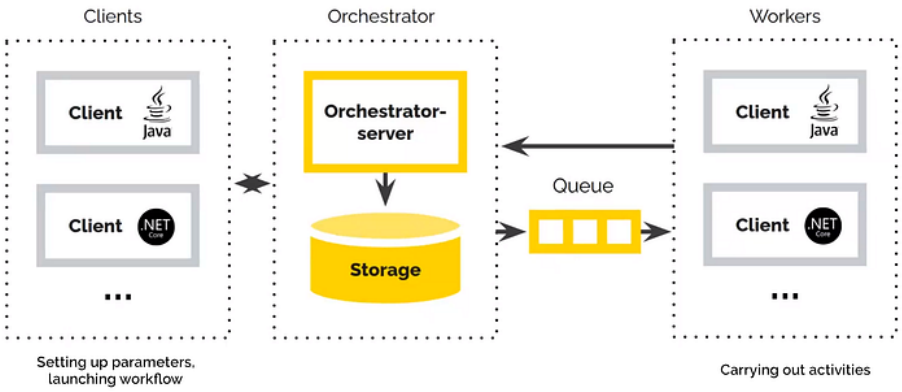


Slika 3.5: Povezave v diagramu

4 ORKESTRACIJA MIKROSTORITEV

4.1 Kaj je orkestracija mikrostoritev?

Mikrostoritve [12] so majhni, ločeni deli programske opreme. Izvajajo točno začrtana dela, ki jim jih naroči orkestrator, ki je hkrati tudi vodja in nadzornik (Slika 4.1). Komunikacija poteka prek protokolov. Prednost orkestracije mikrostoritev je predvsem prilagodljivost, odpornost na napake in olajša se nam medsebojna komunikacija pri izvajanju kompleksnih procesov. Pomaga nam predvsem pri ustvarjanju kompleksnih procesov brez našega posega, saj zato poskrbi orkestrator. Ta kot vodja in organizator ter upravitelj nalog sprejme nalogo in jo dodeli v delo določeni mikrostoritvi. Zraven upravljanja skrbi tudi za življenjski cikel, saj ustvari vrstni red izvajanja mikrostoritev. Te se nato izvajajo v točno določenem vrstnem redu, v primeru napake se nadaljuje s stopnje, kjer je prišlo do napake. Rešitev se uporablja predvsem za vrstni red izvajanja mikrostoritev in kadar želimo imeti popoln nadzor nad mikrostoritvami. Najbolj poznano orodje je Kubernetes[11].



Slika 4.1: Orkestracija mikrostoritev [13]

4.1.1 Prednosti

Orkestracija mikrostoritev [25] prinaša številne prednosti pri uporabi. Kot prvo ni vezana na jezik izvajanja. Določi se lahko potek izvajanja z vsemi omejitvami. V primeru težje

obremenitve se samodejno prilagodi ter še vedno omogoča učinkovitost in odzivnost, kar je tudi najbolj pomembno. Rezultat ukrepanja ob napakah prinese zanesljivost sistema, saj se v primeru napake lahko sistem obnovi. Neodvisnost omogoča lažje testiranje. Viri, ki so na voljo, omogočajo ločeno upravljanje tako, da lahko dodelimo količino strojne opreme, ki je potrebna za neko izvajanje, in tako, kar se da najbolje, izkoristimo sistem. Ker se vsaka mikrostoritev izvaja ločeno, imamo vpogled v proces in analitiko.

4.1.2 Slabosti in omejitve

V osnovi gre za kompleksen proces, dosti načrtovanja in analize. Pojavijo se dodatni stroški, zaradi vzpostavitve orkestracijske infrastrukture. Dobra izbira vsega potrebnega pri sami izgradnji in vzpostavitvi zna prinašati kar nekaj težav. Gre za nekakšno specifiko, za katero sta potrebna čas in veliko znanja, da se izobrazi določene ljudi. Delna omejitev je zaradi orodij, ki nam omogočajo orkestracijo mikrostoritev. Zaradi velikosti pride tudi do vprašanja varnosti in obvarovanja le-teh – v primeru napadov. Kljub olajšanju določenega dela gre na drugi strani za veliko količino mikrostoritev, ki so del orkestracije. Mikrostoritve so znotraj orkestracije odvisne ena od druge tako, da v primeru odpovedi ene oziroma napake na eni mikrostoritvi lahko pride do težav. Naslednje mikrostoritve, ki so odvisne od te, ne morejo delovati.

4.2 Kaj so koreografske mikrostoritve?

Pri tem pristopu ne potrebujemo nobene vodje, saj so mikrostoritve možne delovati same zase – prek nekega upravitelja. Prav tako med seboj ne vplivajo ena na drugo. Vključen je posrednik, ki prenaša sporočila, da se je določena mikrostoritev izvedla v danem trenutku. Ko prejmejo to sporočilo ostale, takoj vedo, kaj storiti, brez da bi dobile kakšno dodatno obrazložitev. Proces poteka istočasno oziroma asinhrono. Omogoča lažjo manipulacijo in spreminjanje. Pristop se najbolj uporablja, kadar se želimo izogniti

eni točki napake in želimo dodati neodvisnost. Najbolj poznani orodji: Amazon SQS [3] in RabbitMQ [18].

4.2.1 Prednosti

Veliko prednost prinašajo predvsem neodvisnost in asinhronost ter večja razširljivost. Neodvisen razvoj in enostavno vzdrževanje ter manjša nagnjenost k napakam sam sistem naredijo odpornejši. Razvijalci se zaradi neodvisnosti lahko preprosto osredotočijo na specifične naloge. Odpoved sistema in napake, ki se lahko pojavijo, ne vplivajo na ostale mikrostoritve. Omogočena je uporaba različnih tehnologij in orodij. Prav tako sta samovzdrževanje in delo na sistemu lažja – zaradi neodvisnosti.

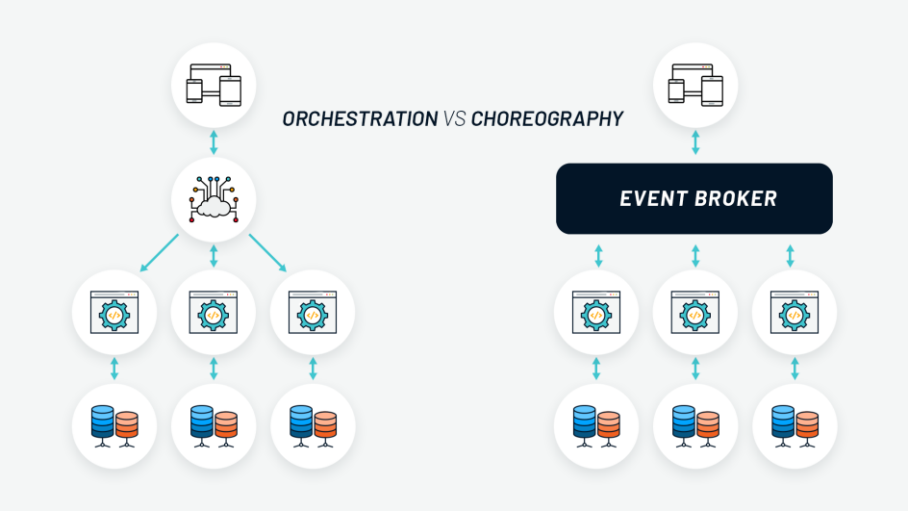
4.2.2 Slabosti in omejitve

Treba je spremeniti pogled in mišljenje o samem delovanju, da boljše razumemo samo delovanje. Zaradi svoje razpršenosti ga je težje upravljati in vzdrževati. Neodvisni odzivi posamezne mikrostoritve lahko vodijo k neuspehu. Velika kompleksnost in zapletenost sistema oslabita celoten pregled nad sistemom. Odpravljanje napak je zelo zamudno in težavno. Prednost v asinhronosti prinese tudi slabost pri komuniciranju – zaradi zagotavljanja doslednosti. Sama težavnost sistema lahko pri testiranju prinaša velik izziv. Zaradi komunikacije je treba zagotoviti neko stopnjo varnosti in zaščite. Zakasnitve med mikrostoritvami lahko privedejo do nepravilnega delovanja in napak.

4.3 Primerjava orkestracije mikrostoritev in koreografskih storitev

Koreografske mikrostoritve [15] so veliko bolj kompleksne napram orkrestraciji mikrostoritev (Slika 4.2). Za razliko orkestracije mikrostoritev, kjer potrebujemo orkestratorja kot vodjo, ta ni potreben pri koreografskih storitvah, saj so posamezne mikrostoritve neodvisne in so sposobne delovati same zase. Za izbiro koreografskih storitev bi se predvsem odločili, kadar bi želeli doseči razširljivost projekta in sprejeti

veliko kompleksnost. V primeru, da želimo stabilnejšo rešitev, ki jo bomo lažje spreminjali, je vsekakor boljše izbrati orkestracijo ali celo razmisliti o izbiri hibridnega pristopa. Orkestracija temelji predvsem na centralnem upravljanju in komunikaciji med mikrostoritvami, napram koreografskim mikrostoritvam, kjer je zgodba popolnoma drugačna, saj vse temelji na neodvisnosti. Lahko uporabimo tudi hibridni pristop, kjer gre za mešanico obeh pristopov, ki vsebuje tako sinhrone kot asinhrone komponente.



Slika 4.2: Primerjava orkestracije mikrostoritev in koreografskih storitev [14]

4.4 Uporaba v produktu

Za sam potek integracije je potreben nek postopek oziroma koraki, ki se izvajalo eden za drugim, da je integracija uspešno sestavljena in izvedena. Prav to želimo omogočiti in doseči v samem produktu TRONintegration. Zgoraj smo že omenili, da orkestracijo mikrostoritev sestavljajo mikrostoritve in orkestrator. V našem primeru bo orkestrator uporabnik orodja, ki bo s pomočjo grafičnega vmesnika izdelal mikrostoritve (s pomočjo vozlišč), ki bodo del integracije. Te se bodo izvajale ena za drugo. Zaporedje izvajanja bo določeno s pomočjo povezav (puščic). Skupek vsega bo predstavljalo orkestracijo mikrostoritev (integracijo), ki jo bo zagnal uporabnik.

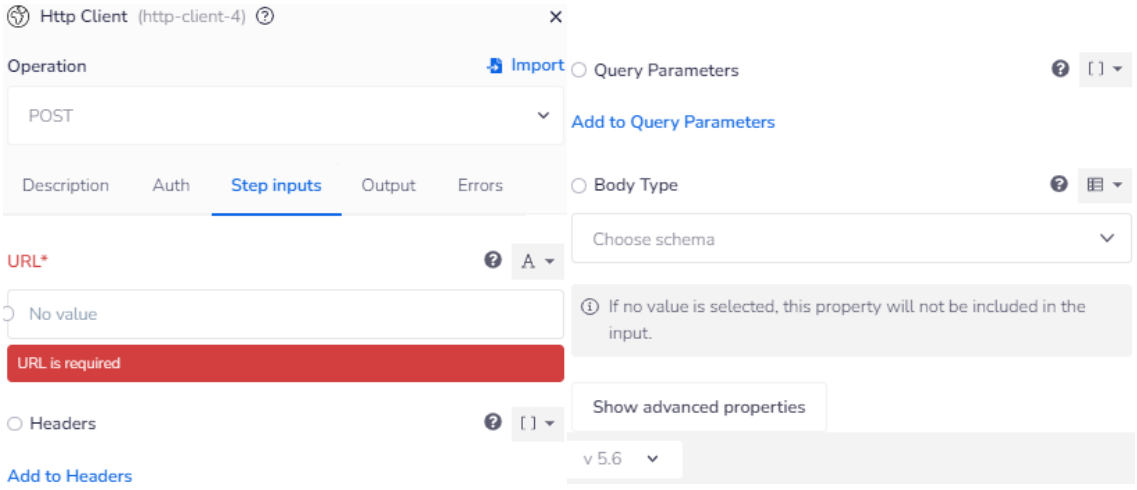
5 RAZVOJ ORODJA TRONintegration

5.1 Predpriprava in načrtovanje orodja

Pred samo raziskavo in fazami razvoja sta bila potrebna predpriprava in načrtovanje orodja. Obsežen sestanek, kjer smo se spoprijeli z nastalo težavo, zaradi katere je potrebna implementacija orodja za izgradnjo integracije. Pregled vseh potreb, ki so nam bile znane za razvoj orodja. Želje, ki smo jih morali podpreti, so prinesle težave, na katere smo še posebej morali biti pozorni v času razvoja. Od načrta samega orodja, tako logičnega kot tudi vizualnega dela, nas je pot peljala skozi razvoj samega produkta in na koncu vse do testiranja ter uporabe produkta.

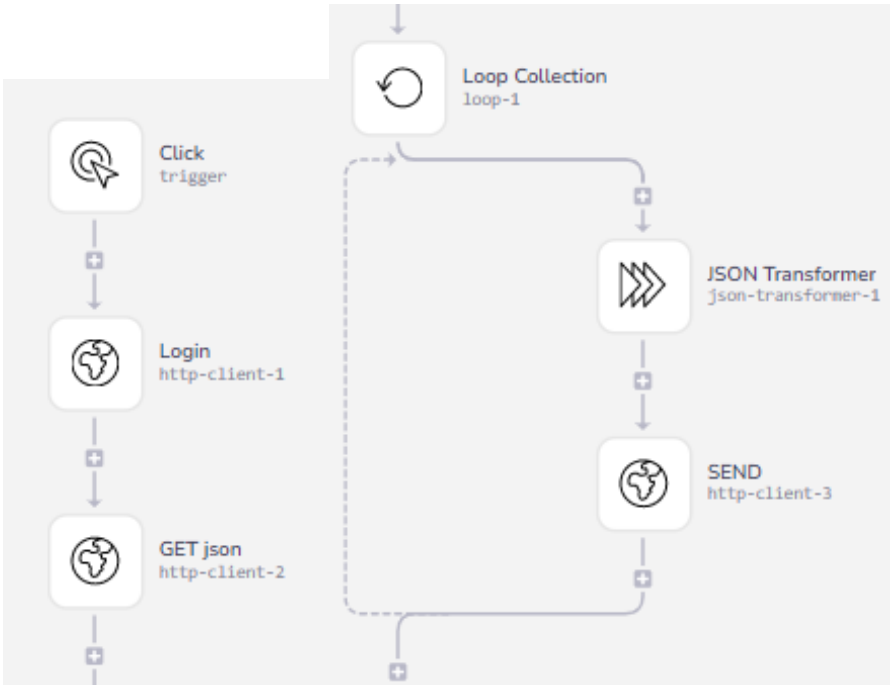
5.2 Raziskava že obstoječih rešitev in idejna zamisel

Po idejni zamisli glede na zahteve in potrebe za izdelavo produkta smo opravili raziskavo že obstoječih orodij. Z raziskavo smo želeli predvsem pridobiti idejne zamisli za izgled in uporabo, ki sta nam predstavljala največjo težavo pri razvoju. Podane ideje bi služile tudi kot idejna zamisel za samo izdelavo, ki smo jo skozi razvoj spreminjali in prilagajali glede na potrebe zahtevanega produkta in potrebe, ki so se pojavile skozi sam razvoj. Po obsežnejši raziskavi smo ugotovili, da orodje, ki bi se najbolj približalo produktu, ki ga želimo razviti, tudi obstaja in se imenuje Tray.io. Orodje med drugim omogoča integracijo s platformo Shopify [19] z že vgrajenimi gradniki. Izdelane ima tudi gradnike (Slika 5.1), ki omogočajo lažje vnašanje podatkov, kar predstavlja bistveno prednost pri izdelavi integracije.



Slika 5.1: Vnos potrebnih podatkov v gradnik Htpp Client v orodju Tray.io

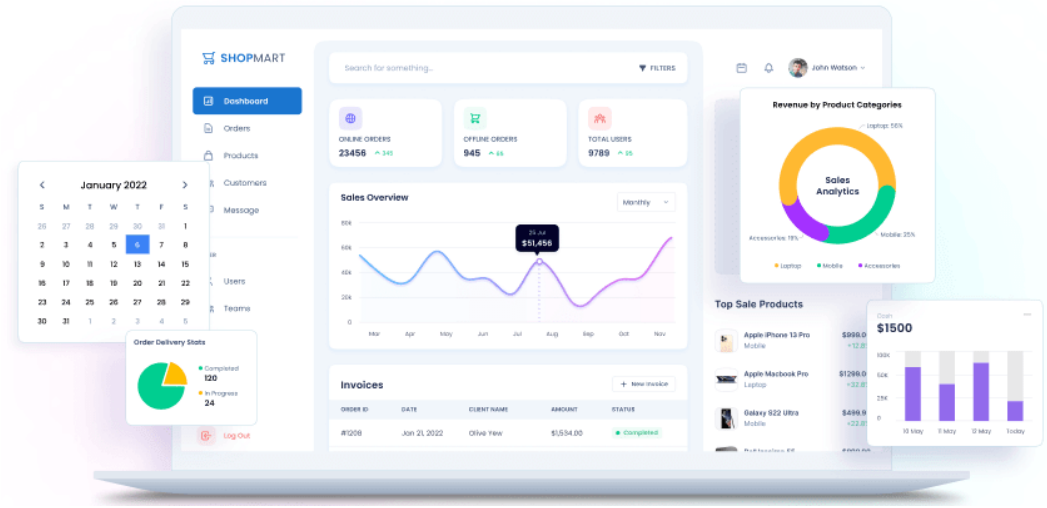
Uporabnik za izdelavo integracije (Slika 5.2) ne potrebuje veliko predznanja in integracijo izdela kar hitro. V osnovi mora vedeti, kaj med seboj povezati in katere podatke vnesti. Produkt je okrnjen, saj ne omogoča dodajanja svojih funkcionalnosti. Brez prilagoditev glede na želje in potrebe, ki so bistveni del razvoja, smo zelo omejeni. Omenjen produkt nam je tako v času načrtovanja in razvoja služil kot iztočnica za implementacijo.



Slika 5.2: Diagram integracije za dodajanje produktov na Shopify v orodju Tray.io

5.3 Izbira tehnologij

Treba je bilo izbrati tehnologije, ki smo jih uporabili za izdelavo. Ta faza je ključna za nadaljnji razvoj, saj je treba uporabiti kar se da dobre tehnologije. Uporabljene tehnologije, ki so omenjene zgoraj, so bile skrbno izbrane glede na tehnologije, ki se že uporabljajo pri drugih produktih v podjetju. Naslednji izziv nam je predstavljala raziskava knjižnice, ki bi nam omogočala estetski in pregleden vizualni prikaz gradnikov, ki služijo kot del diagrama poteka oziroma kot mikrostoritve, ki se izvajajo. Po testiranju izbranih knjižnic se je najbolje izkazala knjižnica Syncfusion, ki deluje najbolje skupaj z ogrodjem. Kljub nekaterim pomanjkljivostim se je na testiranju najbolje izkazala napram drugim. Knjižnica vsebuje velik nabor vizualnih komponent (Slika 5.3) in podpira veliko platform.



Slika 5.3: Komponente knjižnice Syncfusion [21]

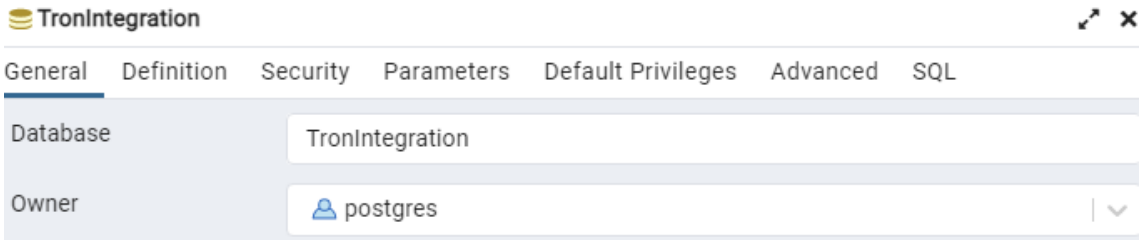
Izmed velikega nabora vizualnih komponent, ki jih knjižnica ponuja, smo se odločili uporabiti diagram, saj prav to potrebujemo za izdelavo vizualnega prikaza poteka integracije. Komponenta knjižnice nam omogoča izgradnjo diagrama z gradniki: vozlišča (ta so predstavljena kot oblike), povezave (te so predstavljene kot puščice) in vgrajene akcije oziroma funkcije, ki nam omogočajo uporabo že vnaprej definiranih funkcij. Vozlišča (v nadaljevanju koraki) v našem produktu predstavljajo posamezno mikrostoritev. Puščice (v nadaljevanju povezave) nakazujejo pot oziroma potek zaporedja mikrostoritev. Vozlišča in povezave je mogoče prilagajati glede na potrebe in

želje. Knjižnica nam v osnovi ponuja vgrajene funkcije, ki jih je mogoče uporabiti in tudi spreminjati. Prav tako je podprto dodajanje svojih lastnih funkcij, kar je bila še ena prednost pri izbiri te knjižnice. S pomočjo dodajanja in spreminjana dosežemo lažjo manipulacijo in prilagajanje diagrama po svojih željah.

Izdelali smo plan razvoja orodja in ga realizirali: vzpostavitev jedra (angl. core), vzpostavitev podatkovne baze, kreiranje potrebnih entitet s pomočjo modelov, dodajanje povezav med njimi, programiranje API-klicev in krmilnikov za CRUD-operacije, testiranje API-klicev, dodajanje knjižnice Syncfusion v projekt, dodajanje gradnikov vozlišč in povezav, dodajanje pojavnih oken za vnos podatkov, izdelava seznamov za prikaz podatkov, dodajanje potrebnih stvari za potrebe integracije, pregled celotnega produkta in testiranje robnih primerov ter popravljanje le-teh, izdaja in predstavitev.

5.4 Vzpostavitev jedra in podatkovne baze

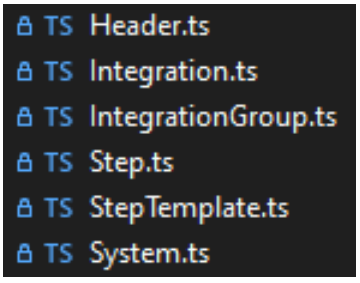
Za začetek razvoja produkta je bilo treba namestiti programsko opremo, za katero smo se skrbno odločili. S pomočjo smo vzpostavili projekt in kreirali podatkovno bazo z vsemi potrebnimi nastavitvami (Slika 5.4). Ta del je prevzel zaposlen v podjetju, saj gre za administrativne zadeve, ki jih opravi administrator. Kreiral je nov repozitorij za verzioniranje kode in vanj dodal jedro produkta. Podjetje uporablja jedro, ki je skupno vsem produktom podjetja, kot iztočnica za začetek razvoja. Znotraj tega so definirane osnovne stvari in prijavna logika. Iz pouporabljenega jedra je izbrisal dele, ki ne bi prišli v poštev in so specifični za določen produkt. To so bili predvsem deli na zalednem delu – definiranje entitet in krmilnikov, na čelnem delu praktično skoraj vse, kar je nepotrebno. Po brisanju je ostala le implementacija prijave v produkt. Sledil je kratek test ostalih funkcionalnosti. Testiranje je pokazalo, da je jedro bilo dobro vzpostavljeno in deluje, kot mora. Nato je vzpostavil podatkovno bazo in uredil vse potrebne varnostne nastavitve. Sledila sta povezava produkta in podatkovne baze ter testiranje prijave. Po končani fazi smo lahko začeli z implementacijo začrtanega produkta.



Slika 5.4: Kreirana podatkovna baza TronIntegration v PostgreSQL

5.5 Kreiranje modelov entitet in modela DTO za posamezne entitete

Že v fazi načrtovanja in planiranja smo določili, katere entitete bomo potrebovali pri implementaciji. Enako je veljalo za atribute teh entitet, ki smo jih morali naknadno dodajati in malenkostno modificirati, saj se je skozi razvoj produkta pojavila potreba po spreminjanju in dodajanju novih. Celoten produkt (koda) je bil razvit v angleščini – zaradi mednarodne uporabe. Sproti smo vse, kar se izpisuje uporabniku, tudi prevajali v ločeni datoteki. Končni produkt se lahko uporablja prav vsepovsod, saj se preprosto kreira nova datoteka in se vanjo dodajo prevodi iz angleščine v želen jezik, kar nam v prihodnje omogoča spreminjanje le določene datoteke in ni potrebe po spreminjanju celotnega produkta in iskanja, kje se pojavi kakšna beseda. V nasprotnem primeru bi morali celoten projekt prečesati in poiskati, kje vse se pojavijo besede, in jih popraviti. V osnovi smo kreirali modele entitet [1] (Slika 5.5): glava (angl. Header), integracija (angl. Integration), skupine sistemov (angl. IntegrationGroup), korak (angl. Step), predloga korakov (angl. StepTemplate) in sistem (angl. System) (Izsek kode 5.1).



Slika 5.5: Modeli entitet

```
@Entity()
export class System {

    @Column("uniqueidentifier", {
        nullable: false,
        primary: true,
        default: () => "uuid_generate_v1()",
    })
    @PrimaryColumn()
    rowGuidSystem: string;

    @Column({})
    name: string;

    @Column({})
    userCreated: string;

    @Column({ default: () => "now()" })
    createdDate: Date;

    @Column({ default: () => "now()" })
    modificationDate: Date;

    @OneToMany(() => IntegrationGroup,
        (integrationGroup: IntegrationGroup) =>
            integrationGroup.rowGuidIntegrationGroup)
    integrationGroups: IntegrationGroup[];
```

Izsek kode 5.1: Model za kreiranje entitete System

Entitete smo med seboj ustrezno povezali s povezavami. Zraven atributov, ki so namenjeni določenim entitetam, smo vsem entitetam dodali skupne: rowguid (unikatni identifikator posameznega zapisa v podatkovni bazi, createdDate (datum kreiranja zapisa v podatkovno bazo), modificationDate (datum, ko je bila na zapisu v podatkovni bazi narejena sprememba) in created user (podatki o uporabniku, ki je kreiral oziroma ustvaril zapis v podatkovno bazo).

Entiteta glava predstavlja glavo zahteve. Nekatere zahteve potrebujejo pri API-klicih tudi glavo. Odločili smo se za samostojno entiteto, ki smo ji zraven zgoraj omenjenih atributov dodali še ključ (angl. key) in vrednost (angl. value). Entiteto smo povezali z entiteto korak. Entiteto integracija smo kreirali zaradi potrebe po shranjevanju zapisa celotnega diagrama, ki smo ga shranili v JSON-format. Entiteta skupina sistemov služi kot entiteta, ki nam pomaga pri pomoči in predlogih uporabniku, saj se lahko glede na

osnovi izbire prilagajo določene stvari, ki veljajo le za določen sistem. To entiteto sestavljata dva atributa, ki sta povezana s sistemom, kjer eden predstavlja izvorni sistem – torej od kod se bodo vzeli nekateri podatki, in ciljni sistem – torej, kam se morajo prenesti in transformirati ter prilagoditi ti podatki. Entiteta korak nam predstavlja potrebne podatke za izvajanje mikrostoritve, ta se bo izvajala po zaporedju glede na povezave. Gre za najbolj pomembno entiteto, ki vsebuje: shapeId (predstavlja unikatni identifikator oblike v diagramu, ki se določi samodejno ob dodajanju vozlišča v diagram), rowGuidIntegration (gre za povezavo na integracijo in predstavlja pripadnost integraciji), slednji atributi so potrebni za pošiljanje zahtev: metoda (angl. method, gre za izbiro metode GET, POST, DELETE, PUT), ime gostitelja (angl. hostname), pot (angl. path), vrata (angl. port), uporabniško ime (angl. username), geslo (angl. password), HTTPS (ali gre za varno povezavo ali ne), telo zahteve (angl. body) in glava (angl. headers, kjer gre za povezavo na tabelo Header). Za potrebe poti integracije smo dodali naslednje atribute: vrstni red (angl. sortOrder, ki se določi vsakič, ko se zažene integracija, saj se vedno znova računa pot), starš (angl. rowGuidStepParent, vodimo si, kdo je starš trenutnega koraka – zaradi lažjega izvajanja integracije), dodatni starš (angl. rowGudiStepExtra, ki smo ga bili primorani dodati zaradi potrebe pri koraku »pogoj IF«, dodati smo morali nov atribut extra, ki nam omogoča, da točno vemo, ali je korak posledica pogoja pravilen angl. TRUE ali posledica neustrezanja pogoju (angl. FALSE). Nekatere zahteve morajo vsebovati žeton (angl. token), zato smo tudi to dodali kot atribut. Eden izmed korakov je korak podatki (angl. data), kjer se vnese poizvedba (angl. query), v formatu JSONata [9], ki se izvaja in transformira podatke iz ene v drugo strukturo. Entiteta predloga korakov (angl. stepTemplate) je praktično zelo podobna koraku, saj predstavlja korak, ki je namenjen za pouporabo, torej kreira se predloga in se lahko ponovno uporabi, kar pohitri izdelavo integracije. Od koraka se razlikuje zgolj po tem, da nima atributov, ki so namenjeni vodenju poti. Ob uporabi se tako ali tako kreira korak iz predloge koraka. Entiteta sistem (angl. system) predstavlja prednastavljen sistem, za katerega veljajo določene stvari, ki se uporabniku podajo kot predloge. Sistem ima atribut ime in atribut, ki predstavlja povezavo na skupino sistemov.

Po uspešnem kreiranju modelov za entitete smo naredili zagon zalednega dela sistem. V podatkovni bazi so se nam ustvarile tabele entitet. Vsem entitetam smo ustvarili modele DTO [24] (Izsek kode 5.2). Slednji nam omogočajo in olajšajo prenašanje podatkov, brez da bi bili izpostavljeni. Uporabljajo se predvsem zato, ker so klici draga operacija. S pomočjo modela DTO prenašamo le podatke, ki jih bomo potrebovali, in ne celotnih entitet. Znotraj modela DTO dodamo še kriterij (angl. criteria), ki predstavlja nek filter. Na primer, da želimo prejeti podatke iz podatkovne baze, pošljemo nanjo klic s kriterijem, ki določa, da se vrnejo le določeni podatki, ki ustrezajo kriteriju. Atributi, ki smo jih določili v modelu, se morajo ujemati z modeli DTO.

```
export class systemDTO {

    constructor() {
        this.rowGuidSystem = this.userCreated = this.createdDate =
        this.modifyDate = this.name = this.rowGuidIntegrationGroup = undefined;
    }

    rowGuidSystem: string;
    rowGuidIntegrationGroup: string;
    name: string;
    userCreated: string;
    createdDate: Date;
    modifyDate: Date;
}
export class systemCriteriaDTO {

    constructor() {
    }

    dateFrom: Date;
    dateTo: Date;
    filter: string;
}
```

Izsek kode 5.2: Kreiranje systemDTO in systemCriteriaDTO za entiteto System

5.6 Aplikacijski vmesnik

Logika domene oziroma poslovna logika (angl. bussines logic) kodira pravila in zahteve ter podpira uporabo CRUD-operacij. Znotraj te smo spisali celotno logiko. Dogajanje med nivojem podatkov (angl. data layer) in nivojem predstavitve oziroma prikaza (angl. presentation layer). Na primeru entitete »System« bomo predstavili funkcije, ki smo jih spisali (funkcije ostalih entitet so dokaj podobne):

- »saveSystem« (Izsek kode 5.3) shrani oziroma posodobi obstoječo entiteto glede na to, ali ima definiran »rowGuidSystem« ali ne, saj pridobi podatke in preveri, če že obstaja;
- »getSystem« pridobi en sistem iz podatkovne baze na osnovi »rowGuidSystem«;
- »getSystems« pridobi vse sisteme iz podatkovne baze in
- »deleteSystem« izbriše entiteto iz podatkovne baze na osnovi »rowGuidSystem«.

Za potrebe po dodatnih funkcijah in prilagoditvah smo posebej spisali unikate funkcije za določene entitete. Prav tako smo ustrezno obravnavali napake, do katerih bi lahko prišlo pri uporabi funkcij.

```
public static async saveSystem(req: RequestEx, systemDto: systemDTO): Promise<ResponseMsgWCode> {
    let connection = getConnection(req.databaseConnectionName);
    let systemRepository = connection.getRepository(System);

    if (isNullOrUndefined(systemDto.name) && isNullOrUndefined(systemDto.rowGuidSystem))
        return { Success: false, ErrorMessage: 'systemNameNotSet' };

    let system: System = new System();

    if (systemDto.rowGuidSystem != undefined) {
        system.rowGuidSystem = systemDto.rowGuidSystem
    }

    if (systemDto.createdDate == undefined) {
        system.createdDate = new Date();
        system.userCreated = req.user.FirstName + ' ' + req.user.LastName;
    }

    system.modificationDate = new Date();
    system.name = systemDto.name;
    if (isNullOrUndefined(systemDto.rowGuidSystem))
        system.rowGuidSystem = newGuid()
    await systemRepository.save(system);

    return { Success: true };
}
```

Izsek kode 5.3: BussinesLayer za »saveSystem«

Krmilnik (angl. controller), kot pove že samo ime, krmili in izvaja operacije, ki so definirane za posamezni API. Glede na vrsto API-zahteve (POST, GET, PUT, DELETE) je treba prilagoditi krmilnik. Spodaj prikazan primer »router.post« (Izsek kode 5.4) glede na pot »/saveSystem« opravi klic funkcije »saveSystem« z ustreznimi parametri, ki so v tem primeru zahteva (ang. request) in DTO System. Klic se izvede na

»IntegrationBussinessLayer«, znotraj katerega je definirana funkcija in spisana logika. Ta zahteva je namenjena, kadar želimo shraniti nov sistem ali posodobiti obstoječega. Omenjeno je definirano v zalednem delu produkta.

```
router.post('/saveSystem', (req: Request, res: Response) => {
    let system: systemDTO = req.body;

    IntegrationBusinessLayer.saveSystem(req, system).then(_res => {
        res.json(_res);
    }).catch(err => {
        console.log(err);
        HandleError_ResponseMsgWCode(req, res, err);
    });
});
```

Izsek kode 5.4: Controller za »saveSystem«

Na čelnem delu je treba obravnavati to zahtevo s pomočjo servisa (angl. service) in poti (angl. route). Spodaj je prikazan servis »saveSystem« (Izsek kode 5.5), ki kot vhodni parameter prejme »systemDTO«, opravi klic in vrne sporočilo tipa »ResponseMsgWCode«, ki predstavlja statusno kodo glede na uspešnost izvedbe. Ob klicu se pošlje še JSON »system«, ki predstavlja telo zahteve in »generateHeader()«, ki predstavlja generirano glavo zahteve. V primeru napake se izpišeta sporočilo »saveSystem« in težava, ki je povzročila napako.

```
saveSystem(system: systemDTO): Observable<ResponseMsgWCode> {
    return this.http.post<ResponseMsgWCode>(getAPIUrl(IntegrationAPIFunctions.saveSystem),
        JSON.stringify(system), this.userService.generateHeader())
        .pipe(catchError(this.userService.handleError('saveSystem', null)));
}
```

Izsek kode 5.5: Service za »saveSystem«

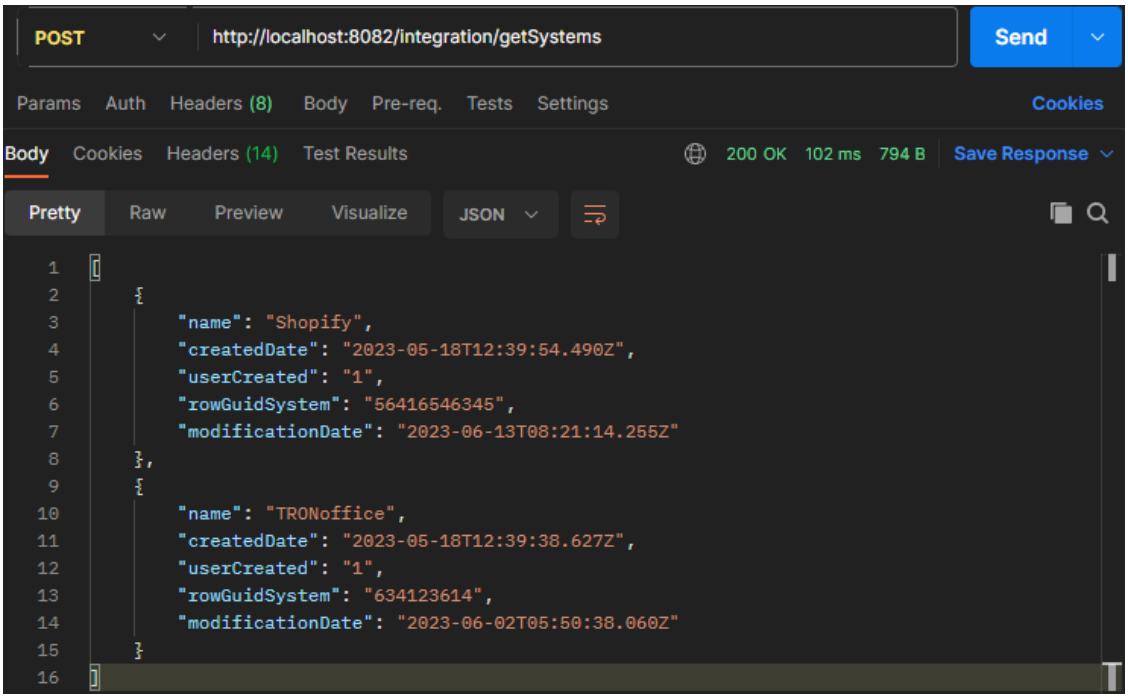
Zraven servisa je potrebna še pot (Izsek kode 5.6), da čelni del ve, kdaj prožiti servis. Omenjeno je prikazano spodaj, kjer sta ime servisa in pot, ki proži servis.

```
saveSystem: ApiURLs.integration + 'saveSystem',
```

Izsek kode 5.6: Route za »saveSystem«

5.6.1 Testiranje aplikacijskega vmesnika in dodajanje zapisov v podatkovno bazo

Po uspešno spisani logiki domene, krmilniku, servisu in poti smo vse API-zahteve testirali s pomočjo orodja Postman [17]. V primeru napake smo te popravili in odpravili. Spodaj je prikazan primer (Slika 5.6), ki vrne vse sisteme s pomočjo zahteve »getSystems«.



Slika 5.6: Testiranje API-klica

Po uspešnem testiranju smo ročno dodali zapise v podatkovno bazo. To nam je omogočilo začetni nabor podatkov, ki je bil predvsem namenjen testiranju in kasnejšim fazam razvoja. (Slika 5.7) prikazuje entiteto »Header« z atributi: rowGuidHeader, key, value, userCreated, modificationDate, rowGuidStep (povezava z entiteto »Step«) in rowGuidTemplate (povezava z entiteto »StepTemplate«).

rowGuidHeader [PK] character varying	key character varying	value character varying	userCreated character varying	createdDate timestamp without time zone
06c061fa-b481-4bea-a...	Content-Type	application/json	Luka Comtron	2023-05-22 13:59:21.975

modificationDate timestamp without time zone	rowGuidStep character varying	rowGuidStepTemplate character varying
2023-05-22 13:59:21.975	33b13407-76e0-47d8-bd4c-f8159c6b2...	[null]

Slika 5.7: Zapis entitete »Header« v podatkovni bazi

Zapis iz podatkovne baze (Slika 5.8) predstavlja primer entitete »Integration« z lastnostmi rowGuidIntegration, name, diagramStructure, userCreated, createdDate, modificationDate in povezavo na entiteto »IntegrationGroup«, predstavljeno kot rowGuidIntegrationGroup.

rowGuidIntegration [PK] character varying	name character varying	diagramStructure json	userCreated character varying	createdDate timestamp without time zone
03ffc4dd-7e80-4340-8a90-ccdb61dd0f3e	TEST	"\"{\\\"enableRtl\\\"...	Luka Comtron	2023-05-29 08:05:29.441

modificationDate timestamp without time zone	rowGuidIntegrationGroup character varying
2023-05-29 13:17:47.516	715d4622-904f-4974-962d-eb5ac5edec...

Slika 5.8: Zapis entitete »Integration« v podatkovni bazi

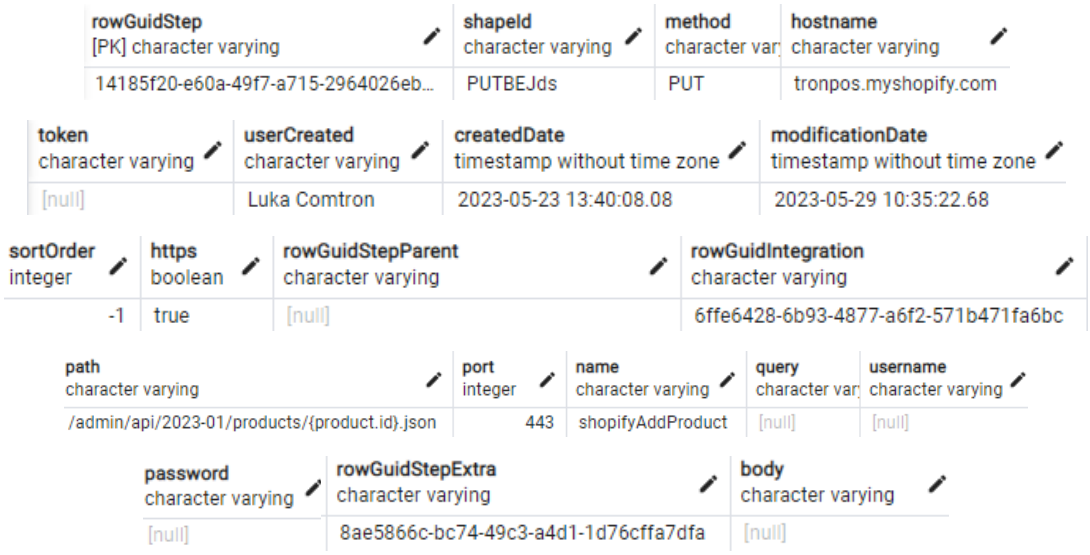
Spodaj prikazan zapis (Slika 5.9) predstavlja entiteto »IntegrationGroup« z vsemi pripadajočimi atributi.

rowGuidIntegrationGroup [PK] character varying	name character varying	userCreated character varying	createdDate timestamp without time zone
3217598e-77ff-414c-b129-e458dea7c0b5	1	Luka Comtron	2023-05-23 10:14:38.563

modificationDate timestamp without time zone	rowGuidSourceSystem character varying	rowGuidTargetSystem character varying
2023-05-23 10:14:45.537	56416546345	634123614

Slika 5.9: Zapis entitete »IntegrationGroup« v podatkovni bazi

(Slika 5.10) prikazuje entiteto »Step« z atributi: rowGuidStep, shapeId, method, hostname, token, userCreated, createdDate, modificationDate, sortOrder, https, rowGuidStepParent (povezava z entiteto »Step«), rowGuidIntegration (povezava z entiteto »Integration«), path, port, name, query, username, password, rowGuidStepExtra (povezava z entiteto »Step«) in body.

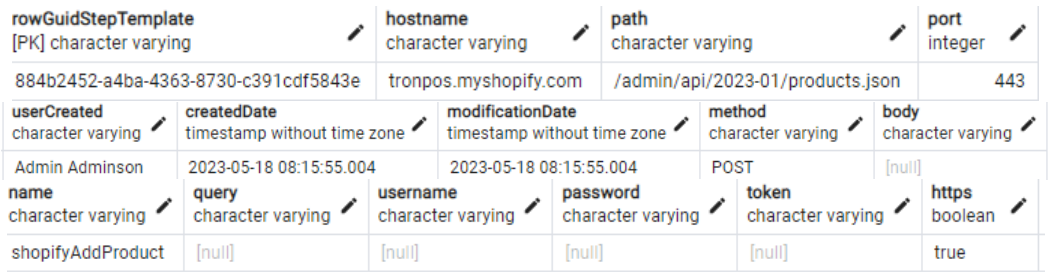


Slika 5.10: Zapis entitete »Step« v podatkovni bazi

Zapis iz podatkovne baze (Slika 5.11) predstavlja primer entitete »System« z vsemi lastnostmi rowGuidSystem, name, userCreated, createdDate in modificationDate.

Slika 5.11: Zapis entitete »System« v podatkovni bazi

Spodaj prikazan zapis (Slika 5.12) predstavlja entiteto »StepTemplate« z vsemi pripadajočimi atributi.



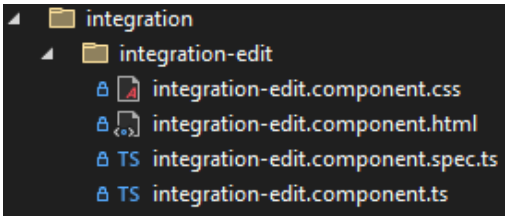
Slika 5.12: Zapis entitete »StepTemplate« v podatkovni bazi

5.7 Syncfusion

Kot prvo je bilo treba kreirati novo komponento (Slika 5.13), ki bo namenjena diagramu (Slika 5.14).

```
ng g c integration edit
```

Slika 5.13: Z ukazom »ng g c integration edit« kreiramo komponente



Slika 5.14: Kreirana komponenta vsebuje datoteke: .css, .html, .spec.ts, .ts

Nato smo v HTML-dokument (Izsek kode 5.7) dodali »ejs-symbolpalette«, ki predstavlja območje, kjer so prikazani gradniki: koraki in povezave.

```
<div>
    <ejs-symbolpalette [palettes]="symbolPalette" [symbolHeight]=50 [symbolWidth]=50 [symbolPreview]="symbolPreviewSettings">
    </ejs-symbolpalette>
</div>
<div>
    <ejs-diagram #diagram id="diagram" height="620px" (doubleClick)="openPopup($event)" (drop)="dropStep($event)">
    </ejs-diagram>
</div>
```

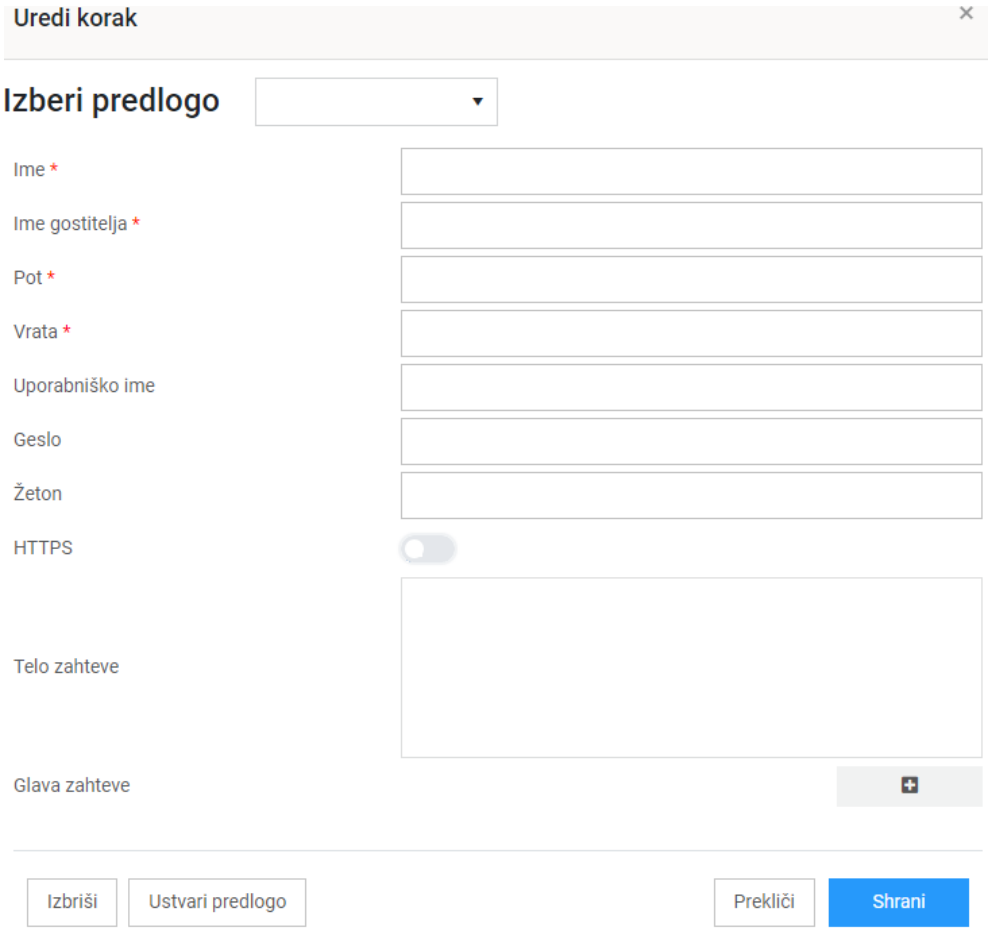
Izsek kode 5.7: HTML-dokument, kjer definiramo območje z gradniki in prikažemo območje diagrama

Zgoraj je bilo že prikazano, kako se definirajo in spreminjajo posamezni gradniki. Območju palete gradnikov lahko nastavljaš velikost, razne nastavitve in zbirko gradnikov. V HTML-dokument smo dodali tudi »ejs-diagram«, ki vizualno predstavlja diagram, ki se nato pretvori in shrani v JSON-format. Diagramu lahko nastaviš velikost in funkcije, ki se naj prožijo na določene dogodke. Posamezne korake dodajaš v diagram s pomočjo miške tako, da izbereš gradnik, ga povlečeš in spustiš v območje diagrama. Gradnik v območju diagrama nato prosto premikaš in spreminjaš velikost po želji. Lahko ga tudi kopiraš in izbrišeš. Za uspešen potek integracije je treba dodati povezave, ki

nakazujejo pot oziroma zaporedje poteka diagrama (integracije). Tako kot vozlišče dodaš tudi povezavo. Želeno povezavo povlečeš in spustiš v območje diagrama. Povezavo je nato treba povezati tako, da ji določiš, od kod do kod gre. Določiš ji torej začetek in konec povezave. Ob dodajanju korakov ali kliku na korak se odpre pojavno okno (opisano v poglavju 5.7.1), v katero je treba vnesti določene podatke – odvisno od izbranega koraka.

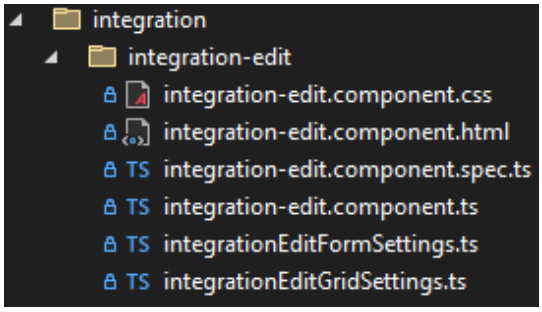
5.7.1 Pojavna okna

Vsa pojavna okna so izdelana dokaj podobno – z razliko, katere podatke je treba vnesti, in malenkostnimi prilagoditvami glede na potrebe. Za prikaz pojavnih oken smo uporabili komponento »kendo-dialog«, ki jo vsebuje knjižnica Kendo UI. Znotraj te komponente je možno narediti in prilagoditi vse potrebno po svojih željah (Slika 5.15).



Slika 5.15: Pojavno okno za vnos podatkov o koraku

Podjetje ima za obrazce izdelano svojo komponento po imenu »app-form-with-label«. To komponento smo preprosto uporabili in jo prilagodili glede na potrebe, ki so bile odvisne od posameznega koraka. V ločeni datoteki smo nastavili nastavitve obrazca (angl. formSettings) (Slika 5.16) tako, da smo definirali polja (Izsek kode 5.8), ki se prikazujejo v pojavnem oknu, jim določili tip in ostale potrebne stvari.



Slika 5.16: Dodana formSettings.ts in gridSettings.ts v komponento integration-edit

```
export const stepEditForm: FormComponentView = {
    ComponentName: 'stepEditForm',
    ComponentType: 1,
    Tabs: [
        {
            TabName: '',
            Groups: [
                {
                    GroupName: '',
                    CSSStyle: 'width: inherit; grid-template-columns: 1fr;',
                    Fields: [
                        { FieldName: 'name', FieldType: 'input', FieldRequired: true },
                        { FieldName: 'hostname', FieldType: 'input', FieldRequired: true },
                        { FieldName: 'path', FieldType: 'input', FieldRequired: true },
                        { FieldName: 'port', FieldType: 'input', FieldRequired: true },
                        { FieldName: 'username', FieldType: 'input' },
                        { FieldName: 'password', FieldType: 'input' },
                        { FieldName: 'token', FieldType: 'input' },
                        { FieldName: 'https', FieldType: 'input' },
                        { FieldName: 'body', FieldType: 'multilineInput', MultilineInputRows: 5 },
                        {
                            FieldName: 'headers', FieldVisible: false,
                            Controls: [{
                                Type: 'button', CssClass: 'headersButton fas fa-plus-square',
                                OperationEmitEnum: 'showHeaders', Title: 'headers'
                            }]
                        },
                    ]
                }
            ]
        }
    ],
    Add: false,
    Edit: true,
    Delete: true,
    CanEditComponentView: false,
    DisplayType: 0,
    TabMoveType: 1
};
```

Izsek kode 5.8: Definiranje polj s pomočjo nastavitev za pojavno okno

S pomočjo določenih nastavitev je možno dodati svoje gumbe v obrazec. Znotraj HTML-datoteke [2] smo uporabili ta obrazec znotraj pojavnega okna. Prikaz pojavnega okna je bil odvisen od vrednosti spremenljivke in pogoja »*ngIf«, ki je nadzoroval, kdaj se naj pojavno okno prikaže in kdaj naj se zapre.

Ob kliku na korak se je med drugim prožila funkcija »getDataFromDatabase« (Izsek kode 5.9), ki je na osnovi poslanih parametrov pridobila podatke o koraku, če je seveda že obstajal v podatkovni bazi, in polja obrazca so se napolnila s pridobljenimi podatki.

```
async getStepDataFromDatabase(shapeId: any, rowGuidIntegration: any) {
    this.loadingService.busyYes();
    var data = await this.integrationService.getStep(shapeId, rowGuidIntegration).toPromise();
    this.loadingService.busyNo();
    return data;
}
```

Izsek kode 5.9: Pridobitev podatkov o koraku

V nasprotnem primeru se je odprlo pojavno okno brez podatkov. Zaradi interakcije z uporabnikom smo dodali »loadingService«, ki ga ima podjetje v svojem jedru. V osnovi gre za pojavno okno, ki prikaže nalaganje tako, da se uporabniku zdi, da se nekaj dogaja. Ko se podatki oziroma akcija konča, se pojavno okno z nalaganjem tudi zapre. Pri svoji implementaciji smo uporabljali asinhrone funkcije z uporabo ukaza »async«.

Znotraj .ts datoteke smo definirali »formOperationHandler« (Izsek kode 5.10) za določen obrazec, ki je bil definiran v »formSettings«.

```
public async integrationFormOperationHandler({ operation, dataItem }): Promise<void> {
    switch (operation) {
        case 'save': {
            this.loadingService.busyYes();
            Object.keys(dataItem).forEach(key => this.currentIntegration[key] = dataItem[key]);
            this.currentIntegration.diagramStructure = JSON.stringify(this.diagram.saveDiagram());
            this.currentIntegration.rowGuidIntegration = this.rowGuidIntegration
            this.currentIntegration.integrationNameSet = this.integrationNameSet
            this.currentIntegration.rowGuidIntegrationGroup = this.rowGuidIntegrationGroup

            let saveResponse = await this.integrationService.saveIntegration(this.currentIntegration).toPromise();
            if (saveResponse.Success == true) {
                this.showAddIntegrationDialog = false;
                this.messageService.showMessage({ type: MessageType.success, content: 'saveSucessfull' });
                this.integrationName = this.currentIntegration?.name != undefined ? this.currentIntegration?.name : undefined;
            } else {
                this.messageService.showResponseMsgWCode(saveResponse);
            }
            this.loadingService.busyNo();
            break;
        }
        case 'cancel': {
            this.showAddIntegrationDialog = false;
            this.router.navigate(['/integrations']);
            break;
        }
    }
}
```

Izsek kode 5.10: Upravljalec operacij

Ta funkcija določa, kaj se bo zgodilo ob določeni akciji. Ob kliku na gumb shrani (angl. save) se pokaže okno z nalaganjem, nastavijo se določene vrednosti spremenljivkam, izvede klic funkcije na servis, pridobi se odziv in na osnovi tega se prikaže sporočilo ter na koncu se zapre pojavno okno. V primeru, da korak že obstaja, se vrednosti posodobijo. V nasprotnem primeru se ustvari nov zapis. Ob pritisku gumba prekliči (angl. cancel) se zapre pojavno okno in preusmeri uporabnika na stran »/integrations«.

Trenutni korak priredimo »formSetting.dataItem« (Izsek kode 5.11) in nato glede na pogoj nastavimo »formSetting.formGroup.patchValue«. S tem posodobimo oziroma pobrišemo podatke znotraj posameznega obrazca in mu nastavimo nove vrednosti.

```
else if (clickedStepName.startsWith('VARIABLE')) {
    this.showEditVariableDialog = true;
    this.saveVariableFormSettings.dataItem = this.currentStep
    if (this.saveVariableFormSettings.formGroup != undefined)
        this.saveVariableFormSettings.formGroup.patchValue(this.currentStep);
}
```

Izsek kode 5.11: Popravek shranjenih nastavitev

Dodatno je bilo treba podpreti akcijo na pojavnem oknu, ki se sproži ob pritisku ikone »X« (Izsek kode 5.12), ki zapre okno. Torej v osnovi se zapre okno in pobrišejo podatki iz vnosnih polj. Ob zapiranju pojavnega okna se kliče funkcija »clearInputFields«, ki izbriše korak iz diagrama v primeru, da še ni shranjen v podatkovni bazi. V primeru, da že obstaja v podatkovni bazi, »formSettings-formGroup.patchValue« ustrezno obravnava vrednosti in zapre pojavno okno.

```
public clearInputFieldsLoop() {
    if (isNullOrUndefined(this.currentStep) || isNullOrUndefined(this.currentStep?.rowGuidStep)) {
        this.removeStep()
    }
    this.currentStep = new stepDTO();
    if (this.saveLoopFormSettings.formGroup != undefined)
        this.saveLoopFormSettings.formGroup.patchValue(this.currentStep);
    this.showHideDialog("loop");
}
```

Izsek kode 5.12: Zapiranje pojavnega okna

Pojavno okno ima na vrhu spustni seznam, ki omogoča izbiro predlog, ki smo jih kreirali, kar nam prinese možnost ponovne uporabe določenih predlog. Na dnu je dodatni gumb, ki se pokaže, kadar je korak že shranjen v bazi, saj se le-tako lahko dodajajo glave zahteve in uredi povezava, saj mora imeti korak določen »rowGuidStep«, ki se avtomatsko generira ob dodajanju novega v podatkovno bazo. »RowGuidStep« je podatkovnega tipa GUID.

Gumb ustvari predlogo kreira novo predlogo, ki je vidna in jo je možno spreminjati v razdelku predloge korakov. Tipka izbriši izbriše korak iz podatkovne baze, prav tako iz diagrama. Določene akcije tudi sproti shranjujejo strukturo diagrama. Možno je tudi

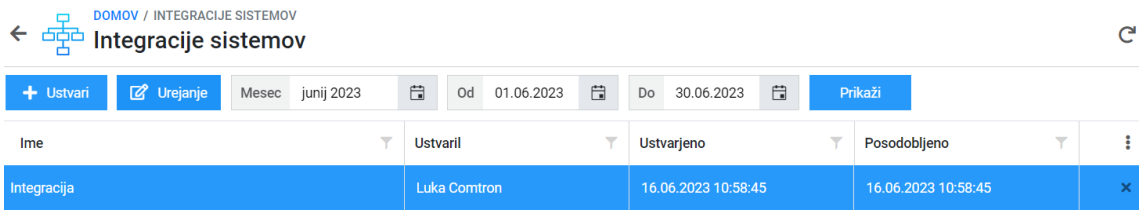
shraniti diagram s pritiskom tipke shrani nad diagramom. Določeni koraki imajo tudi tipko glave zahteve, ki se pojavi, kadar je korak že shranjen. Odpre se nam pojavno okno z možnostjo dodajanja, urejanja in brisanja glave zahteve.

5.7.2 Seznami

Za estetski prikaz, na primer vseh sistemov, smo uporabili komponento, ki je že v jedru. Gre za »app-grid«, kjer podane vrstice predstavljajo podatke v podatkovni bazi, ki se prikažejo kot seznam (Slika 5.17). Ta je definiran s pomočjo nastavitev »gridSetting« (Izsek kode 5.13) in deluje podobno kot »formSetting«.

```
export const integrationGroupComponentView: ComponentViewDTO = {
    ComponentType: 1,
    ComponentName: "IntegrationGroup",
    Tabs: [
        {
            TabName: '',
            Groups: [
                {
                    GroupName: '',
                    CSSStyle: '',
                    Fields: [
                        { FieldName: 'name', FieldType: 'input', FieldWidth: 130 },
                        { FieldName: 'userCreated', FieldType: 'input', FieldWidth: 130, FieldLabel: 'createdUser' },
                        { FieldName: 'createdDate', FieldType: 'dateTime', FieldWidth: 130 },
                        { FieldName: 'modificationDate', FieldType: 'dateTime', FieldWidth: 130 }
                    ]
                }
            ]
        }
    ]
    ,
    Add: false,
    Edit: false,
    Delete: true,
    EnableExport: true,
    isSelectable: true
};
```

Izsek kode 5.13: Definiranje mreže s pomočjo nastavitev za pojavni seznam



Slika 5.17: Seznam integracij sistemov

Nad seznamom smo dodali gumb ustvari, kjer se ob pritisku nanj na novo ustvari določen zapis ali odpre pojavno okno glede na komponento, v kateri trenutno si. Ob kliku na

izbrano vrstico se omogoči klik gumba »urejanje«, ki služi za urejanje izbrane vrstice. Zraven so tri vnosna polja, ki služijo kot filter. Ta se ob prihodu na stran avtomatsko nastavi na trenutni mesec. To nastavljanje opravi funkcija. Prav tako se nastavi datum začetka in konca meseca. Filter je možno prilagajati glede na potrebe. Na določenih komponentah smo dodali še iskalno polje, ki omogoča filtriranje po iskanem nizu.

5.8 Implementacija funkcionalnosti

Produkt se bo uporabljal mednarodno, zato je bilo treba dodati prevode in jih ustrezno prevesti. V dokumentu HTML smo s pomočjo ukaza {{ »ime« | »translate« }} (Izsek kode 5.14) ustvarili prevod, ki smo ga nato v ločeni datoteki (sl.json) ustrezno prevedli (Izsek kode 5.15).

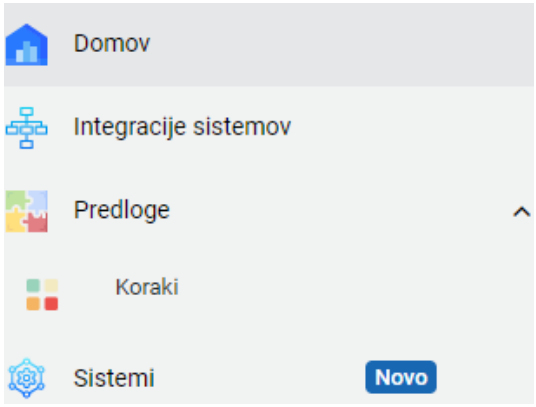
```
<kendo-dialog title="{{'editLoop' |translate }}">
```

Izsek kode 5.14: Definiranje imena in dodajanje prevoda

```
"editLoop": "Uredi zanko",
```

Izsek kode 5.15: Prevedeno v slovenščino

Ustvarili smo navigacijski seznam, ki služi kot meni (Slika 5.18). V podatkovni bazi smo definirali (Slika 5.19): pot, ime, ikono, ki naj se prikaže, in določili vrstni red.



Slika 5.18: Navigacijski seznam, ki služi kot meni

	id [PK] integer	createdAt timestamp without time zone	updatedAt timestamp without time zone	userCreated integer	userUpdated integer	Content character varying (255)
1	1	2023-05-10 09:55:04.708502	2023-05-10 09:55:04.708502	[null]	[null]	Home
2	3	2023-05-10 09:55:04.708502	2023-05-10 09:55:04.708502	[null]	[null]	IntegrationGroups
3	5	2023-05-15 11:06:56.043166	2023-05-15 11:06:56.043166	[null]	[null]	Templates
4	6	2023-05-15 11:06:56.043166	2023-05-15 11:06:56.043166	[null]	[null]	Steps
5	8	2023-05-18 14:26:27.289777	2023-05-18 14:26:27.289777	[null]	[null]	Systems

Url character varying (255)	UrlParameter character varying (255)	Icon character varying (255)
/home	[null]	assets\images\icons\home.png
/integrationGroups	[null]	assets\images\icons\diagram.svg
/templates	[null]	assets\images\icons\iconCommonData.svg
/templates/steps	[null]	assets\images\icons\main-menu.svg
/systems	[null]	assets\images\icons\process.svg

ParentID integer	SortID integer	RoleId character varying (255)	MenuID integer
[null]	0	[null]	1
[null]	10	[null]	1
[null]	20	[null]	1
5	0	[null]	1
[null]	30	[null]	1

Slika 5.19: Definiranje menija v podatkovni bazi

Dodelili smo tudi vrstni red prikaza elementov in določili vgnezdenost elementov seznama. Sledilo je definiranje poti (Izsek kode 5.16) ob kliku na posamezen element znotraj seznama. Določili smo pot in komponento, ki se naj odpre ob kliku na element seznama.

```
{ path: 'home', component: HomeComponent, canActivate: [AuthGuard] },
{ path: 'integrationGroups', component: IntegrationGroupsComponent, canActivate: [AuthGuard] },
{ path: 'integrations', component: IntegrationsComponent, canActivate: [AuthGuard] },
{ path: 'integrations/editintegration', component: IntegrationEditComponent, canActivate: [AuthGuard] },
{ path: 'templates/steps', component: TemplateStepComponent, canActivate: [AuthGuard] },
{ path: 'systems', component: SystemsComponent, canActivate: [AuthGuard] },
```

Izsek kode 5.16: Definiranje poti ob kliku na elemente menija

Za pridobitev strukture in prikaz diagrama smo spisali funkcijo »getData« (Izsek kode 5.17). Ta opravi klic in pridobi podatke o diagramu. S pomočjo »this.diagram.loadDiagram« naložimo strukturo diagrama, ki jo pridobimo iz »diagramStructure«, v kateri je shranjena celotna struktura diagrama.

```
async getData() {
    this.loadingService.busyYes();
    try {
        this.currentIntegration = await this.integrationService.getIntegration(this.rowGuidIntegration).toPromise();
        this.integrationName = this.currentIntegration?.name != undefined ? this.currentIntegration?.name : undefined;
        if (this.currentIntegration.diagramStructure != null)
            this.diagram.loadDiagram(JSON.parse(this.currentIntegration.diagramStructure));
    } catch (err) {
        console.log(err);
        this.messageService.showMessage({ type: MessageType.error, content: err });
    }
    this.loadingService.busyNo();
}
```

Izsek kode 5.17: Pridobitev podatkov o diagramu

Zaradi pomoči uporabniku in možnosti odprave napak smo se odločili izdelati funkcijo »nextStepData« (Izsek kode 5.18), ki uporabniku predlaga ime naslednjega koraka. Najprej pridobi podatke o imenih obstoječih korakov v diagramu. Nato iz imen pridobi zadnji indeks in ponudi predlogo uporabniku. Predloge so nastavljene, da se predlagajo v stilu ime metode oziroma uporabljenega koraka + zaporedni indeks.

```
public static async nextStepName(req: RequestEx, method: string, rowGuidIntegration: string): Promise<string> {
    let connection = getConnection(req.databaseConnectionName);
    let stepRepository = connection.getRepository(Step);

    let steps = await stepRepository.createQueryBuilder('step')
        .where('step.method = :method', { method: method })
        .innerJoinAndSelect('step.rowGuidIntegration', 'rowGuidIntegration')
        .andWhere('rowGuidIntegration.rowGuidIntegration = :rgd', { rgd: rowGuidIntegration })
        .getMany();


    let maxNumber = -1;
    let currentNumber = undefined;
    for (let s of steps) {
        if (s.name.slice(0, s.method.length) == s.method) {
            currentNumber = s.name.slice(s.method.length)
            if (currentNumber > maxNumber)
                maxNumber = currentNumber;
        }
    }
    maxNumber++

    return method + maxNumber.toString();
}
```

Izsek kode 5.18: Predlog imena koraka s pomočjo funkcije nexrStepName

Spodnji izsek kode prikazuje funkcijo, ki pridobi vse sisteme in jih shrani v polje tipa »LookupItem« (Izsek kode 5.19). Potreba po uporabi funkcije se je pokazala, saj je bilo treba znotraj obrazca uporabiti spustni seznam.

```
case 'Systems': {
    let res: LookupItem[] = [];
    if (!isNullOrUndefined(crit.param) && crit.param == 'true')
        res.push({ value: null, label: req.t('allSystems') });
    let systems = await IntegrationBusinessLayer.getSystems(req);
    for (let s of systems) {
        res.push({ value: s.rowGuidSystem, label: s.name });
    }
    return res;
}
```

Izsek kode 5.19: Lookup item

Za transformacijo podatkov v koraku »DATA« smo uporabili odprtokodni izrazni jezik, imenovan JSONata. Uporablja se za preoblikovanje JSON-strukture. To nam je omogočilo, da smo podatke, pridobljene v eni strukturi, pretvorili v drugo strukturo.

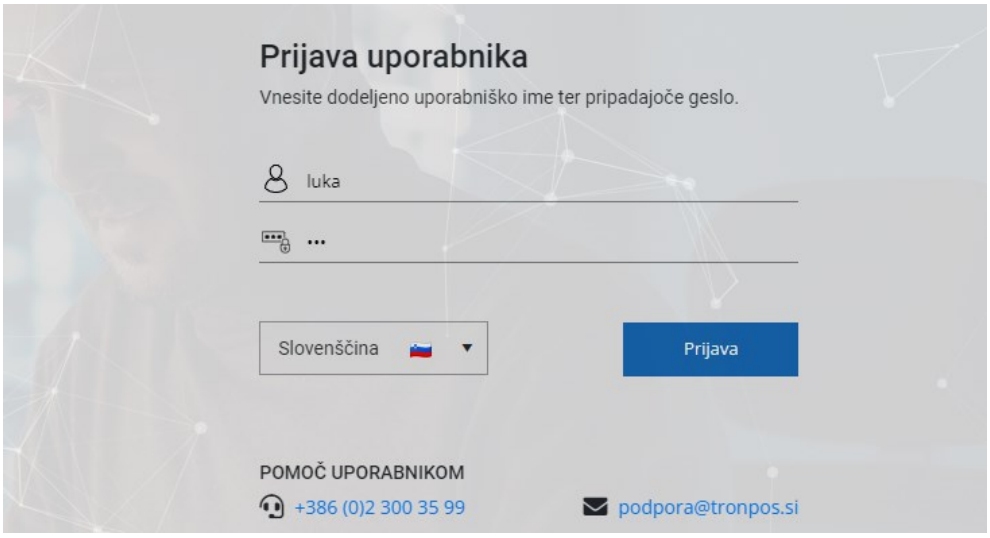
Težava se nam je pokazala pri pridobivanju poti diagrama. Knjižnica sama po sebi ne omogoča pridobitve poti iz kreiranega diagrama, temveč je možno le pridobiti podatke o povezavah, ki so urejene v takem vrstnem redu, kot so bile dodane v sam diagram. Kolega je ta problem rešil s funkcijo, ki vrne pot kreiranega diagrama in hkrati opravi validacijo diagrama.

Problem, ki se nam je še pojavil na platformi Shopify, je bil za nas presenečenje. Shopify podpira več načinov, kako uporabiti API-klic. Ker smo želeli narediti, kar se da univerzalno orodje, smo uporabili Shopify API-klic. Vendar, ta ima omejitev dve poizvedbi na sekundo, kar predstavlja prvo težavo. Naslednja težava je, da ne omogoča ustvarjanja produkta in posodabljanja produkta z uporabo univerzalnega API-klica. Tako smo bili primorani orodje temu prilagoditi, popraviti in optimizirati do te mere, da je po opravljenem popravku delovalo, kot mora.

6 UPORABA APLIKACIJE

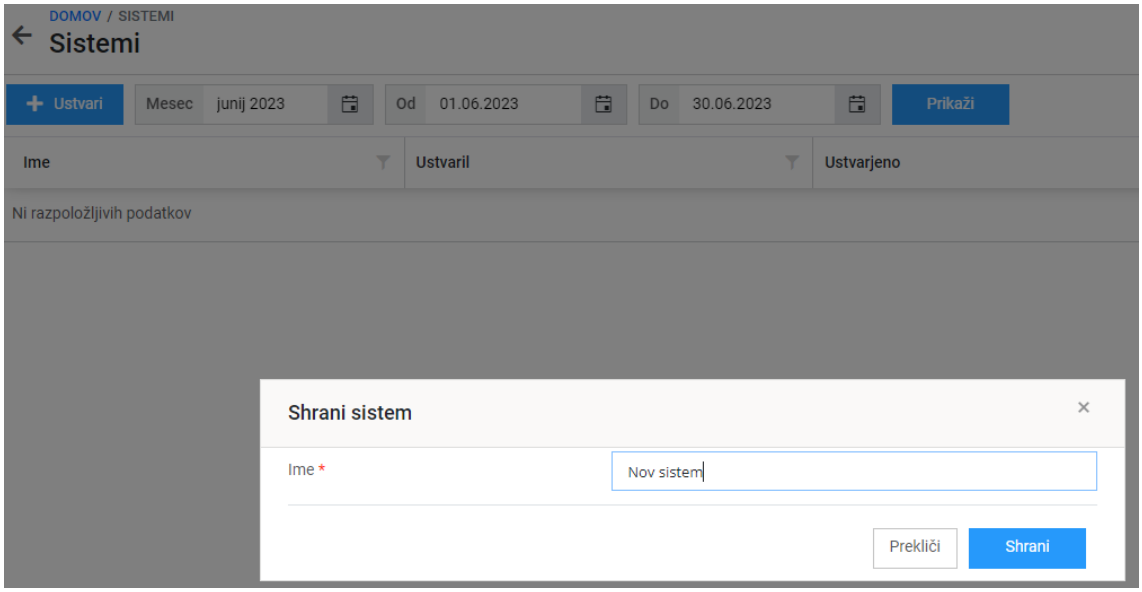
V nadaljevanju bomo po korakih zapisali postopek izdelave integracije za prenos produktov iz produkta TRONoffice v Shopify spletno trgovino in uporabo orodja razvitega orodja.

Ob vstopu je treba izpolniti prijavni obrazec z vnosom uporabniškega imena in gesla (Slika 6.1).



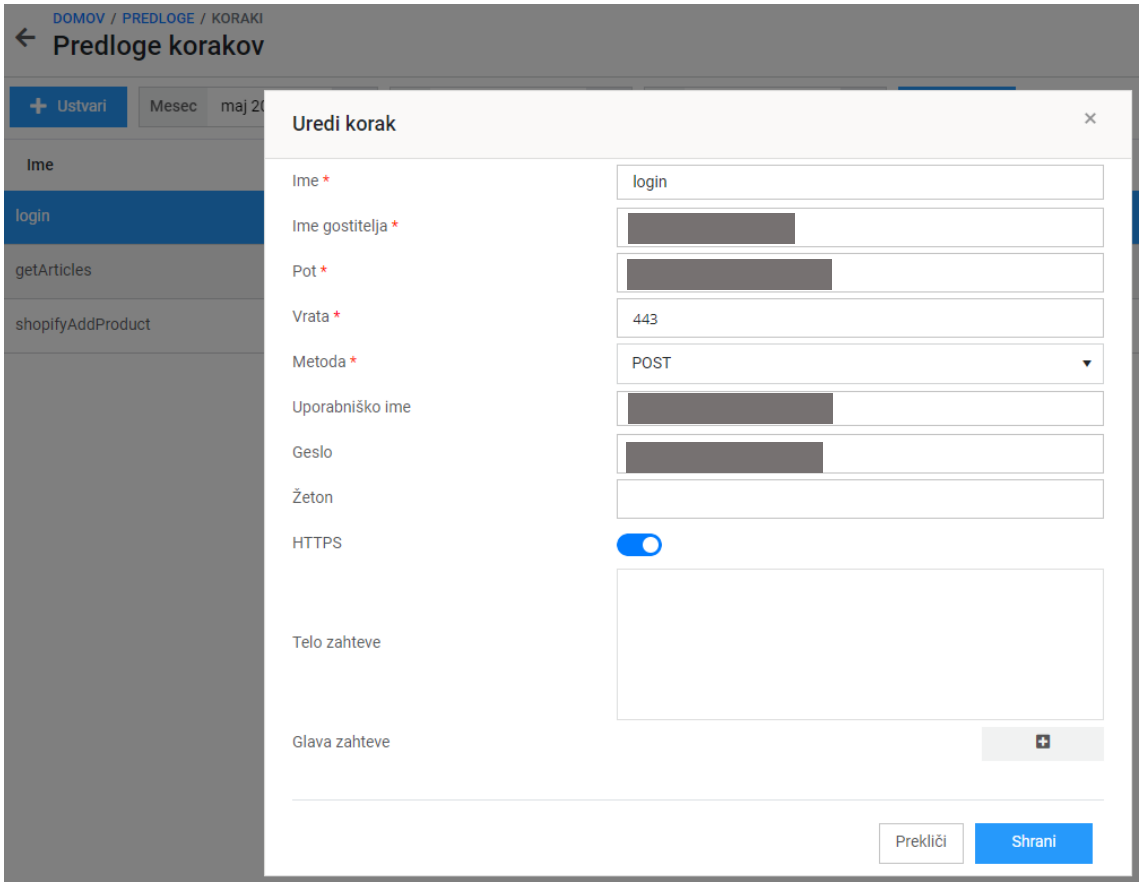
Slika 6.1: Prijava uporabnika

V meniju navigiramo na stran sistemi s klikom elementa v meniju. Ob kliku gumba »ustvari« se nam pojavi pojavno okno, kjer določimo sistem (Slika 6.2), iz katerega bi radi opravili integracijo oziroma v katerega bi radi opravili integracijo. Ob dvokliku na sistem se nam odpre pojavno okno na urejanje. Zgoraj imamo filter za filtriranje po datumu.



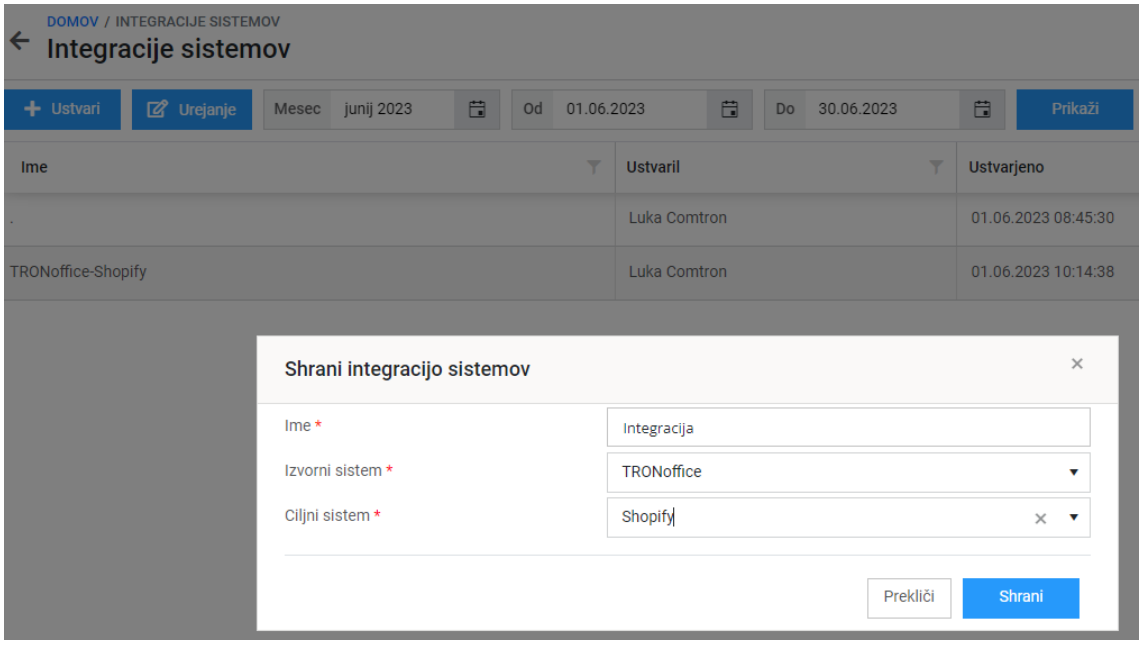
Slika 6.2: Dodajanje novega sistema

Navigacija predloge->koraki nas pripelje na stran, kjer lahko dodajamo nove predloge korakov (Slika 6.3) s klikom na gumb »ustvari«. Ob dvokliku na predlogo koraka (vrstico v tabeli) se nam odpre pojavno okno, ki je namenjeno urejanju predloge koraka. Zgoraj imamo filter, ki omogoča filtriranje po datumu.



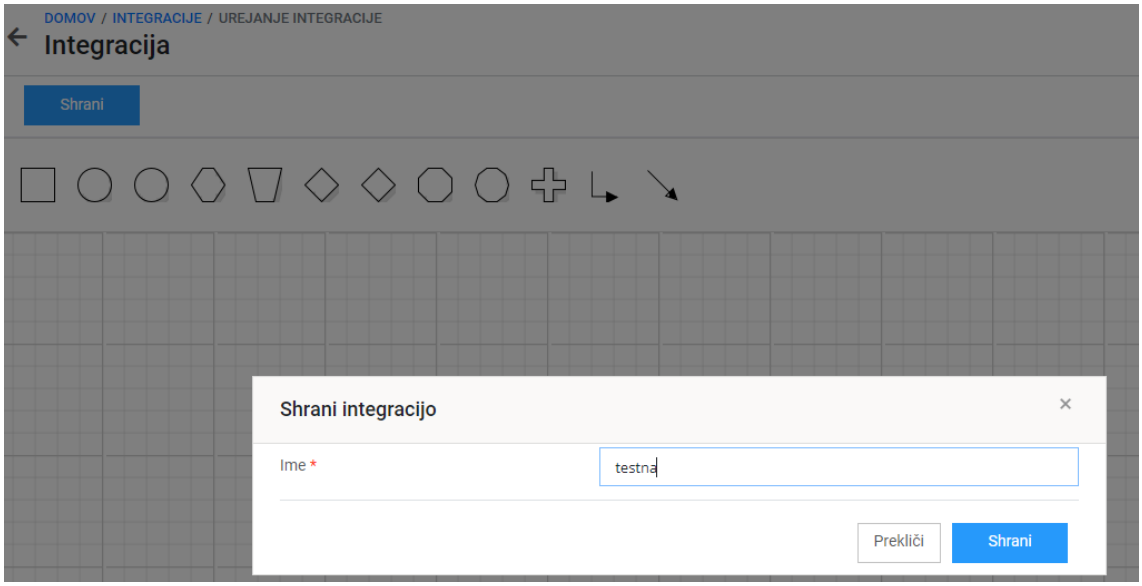
Slika 6.3: Dodajanje nove predloge koraka

Za lažjo organizacijo in pomoč uporabnikom sedaj navigiramo na stran integracije sistemov, kjer dodamo novo integracijo sistemov (Slika 6.4), ki predstavlja, iz katerega sistema bomo integrirali podatke (izvorni sistem) in v kateri sistem bomo integrirali podatke (ciljni sistem). Ob izbiri vrstice in kliku urejanje se nam odpre pojavno okno, ki omogoča urejanje. Zraven so podani filtri za filtriranje po datumu.



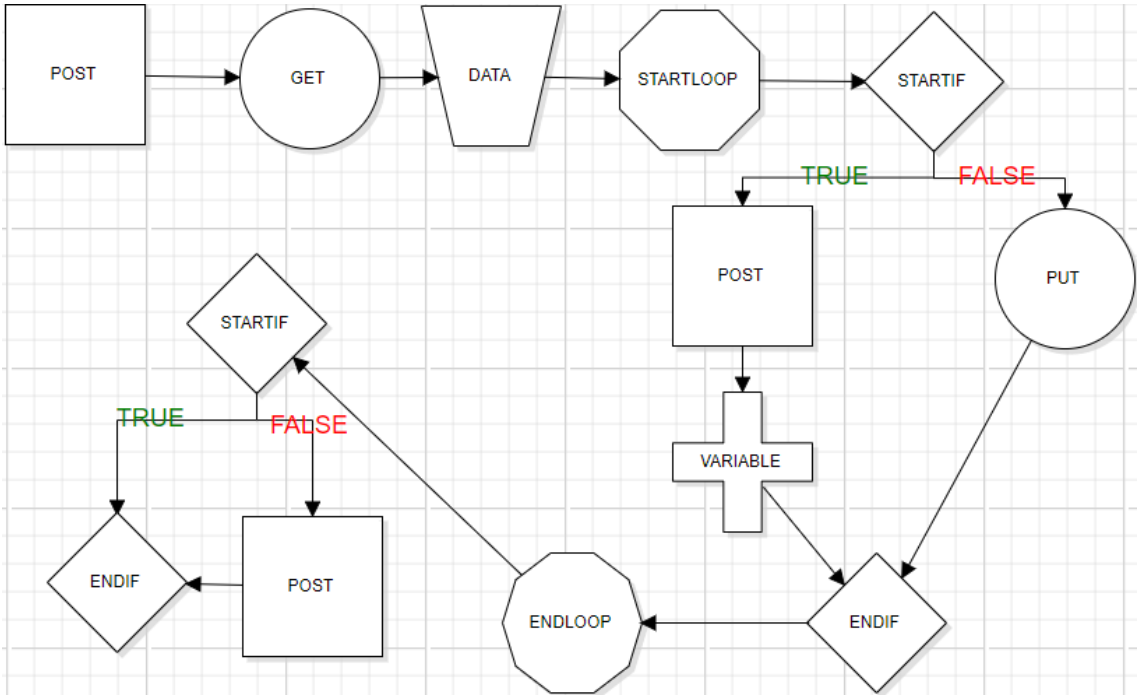
Slika 6.4: Dodajanje integracije sistemov

Ob izbiri sistema znotraj integracije sistemov in kliku ustvari smo preusmerjeni na stran, kjer potekata dejanska vizualna predstavitev in gradnja integracije (Slika 6.5).



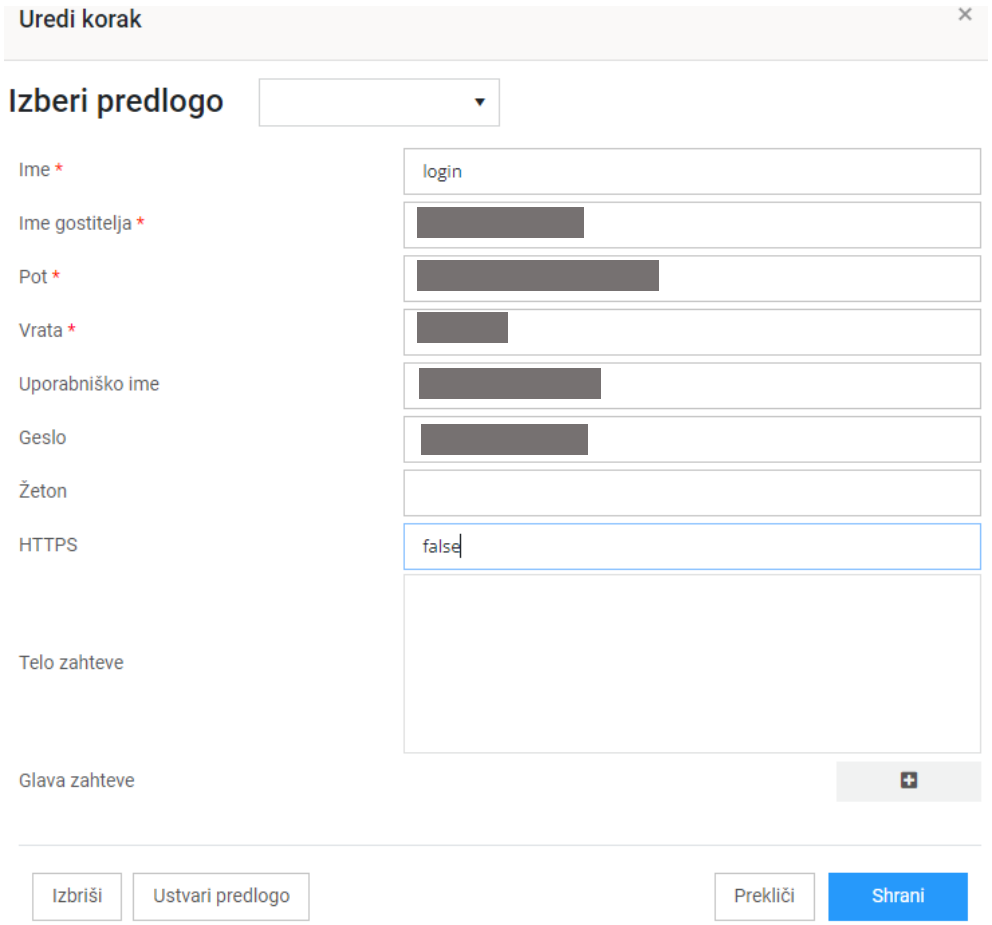
Slika 6.5: Dodajanje nove integracije

S pomočjo miške nato dodajamo posamezna vozlišča (korake) in puščice (povezave) v diagram (integracijo) (Slika 6.6). Z izpolnjevanjem obrazca tako dodamo funkcionalnost posameznim komponentam.



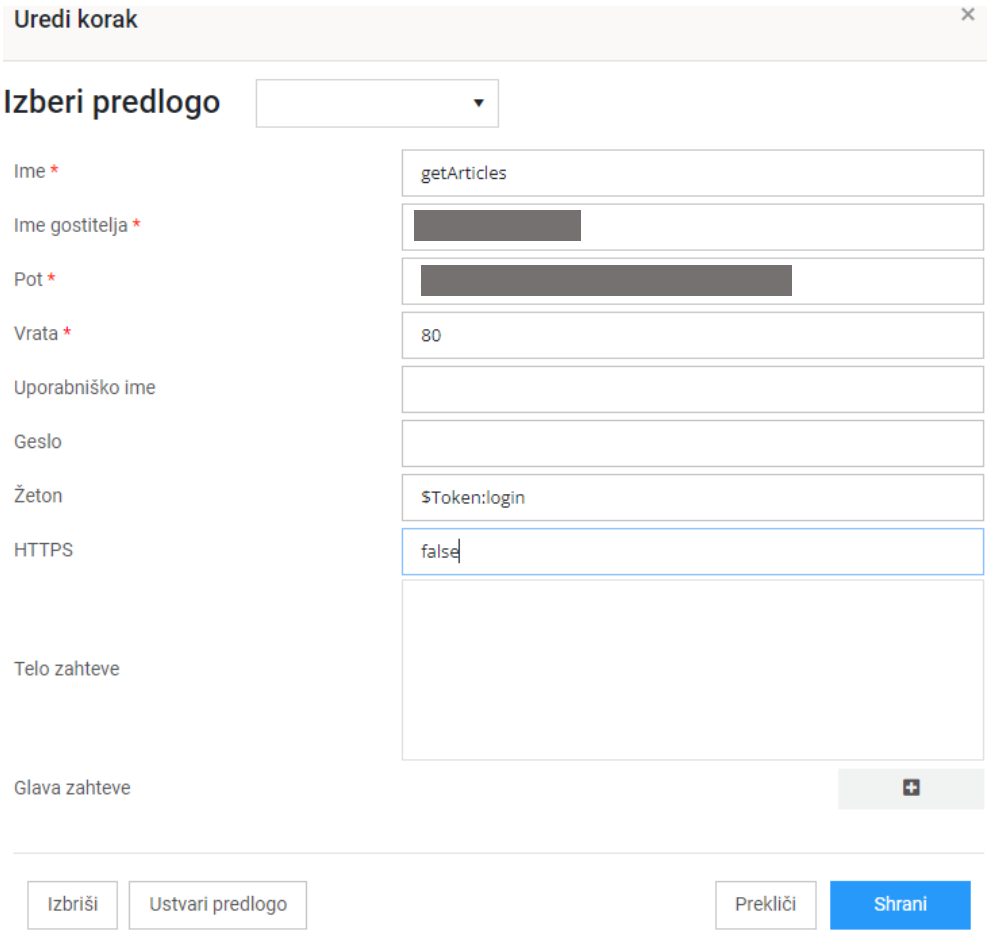
Slika 6.6: Izdelana integracija

Spodaj so prikazana pojavna okna za vsak korak (vozlišče) posebej oziroma mikrostoritev. Korak »login« (Slika 6.7) prikazuje potrebne podatke za prijavo v TRONoffice. Določeni podatki so skriti, saj gre za izdelavo integracije nad realnimi podatki. Treba je bilo vnesti ime, ime gostitelja, pot, vrata, uporabniško ime, geslo in HTTPS nastaviti na »false«, saj gre za HTTP-povezavo in ne HTTPS.



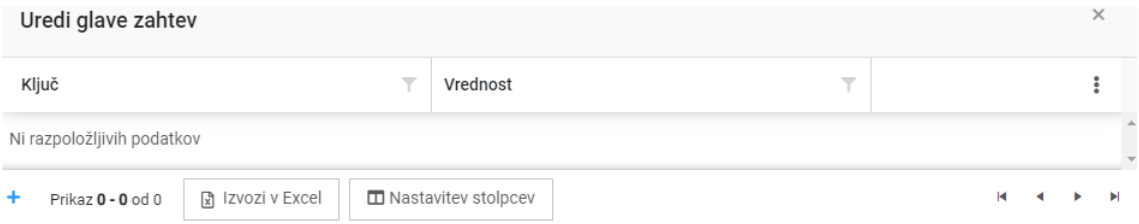
Slika 6.7: Korak POST, ki opravi prijavo na TRONoffice

Korak »getArticles« (Slika 6.8) prikazuje potrebne podatke za prejem artiklov iz produkta TRONoffice. Treba je bilo vnesti ime, ime gostitelja, pot, vrata, HTTPS nastaviti na »false« in nastaviti žeton, ki je potreben za klic. Ta se pridobi iz odgovora koraka »login«.



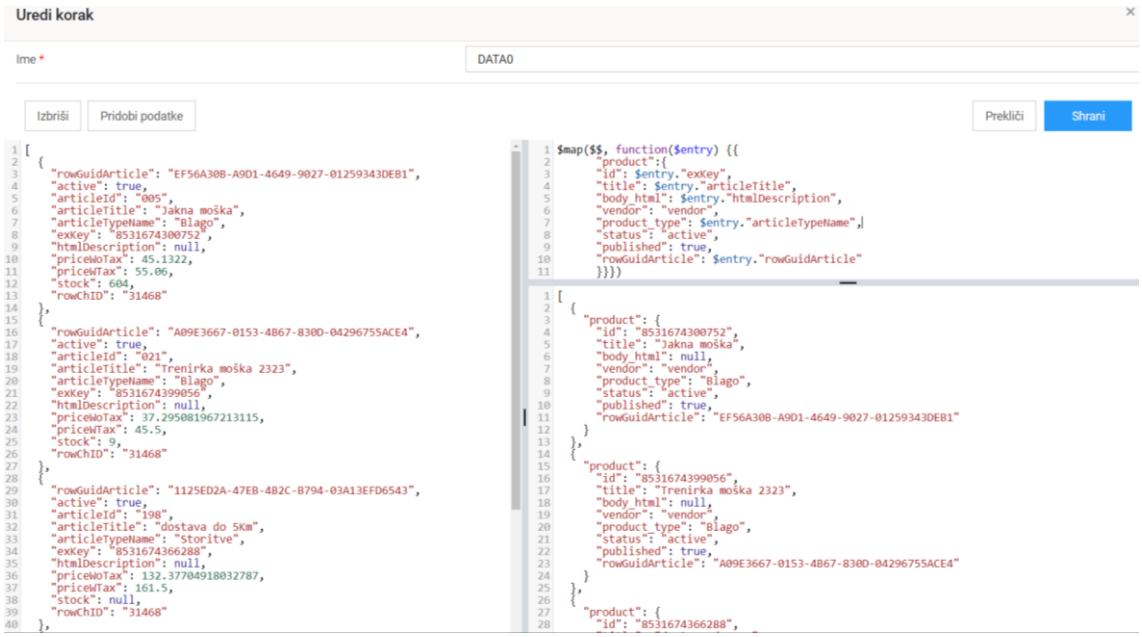
Slika 6.8: Korak GET, ki pridobi artikle iz produkta TRONoffice

Dodajanje, urejanje in odstranjevanje glav zahtev (Slika 6.9). Preko gumba »+« dodamo novo glavo zahteve.



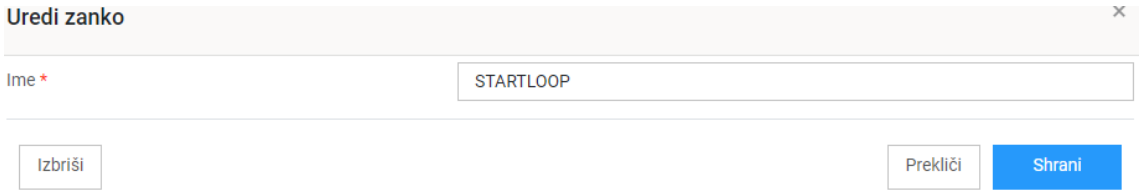
Slika 6.9: Dodajanje glav zahtev

Korak »DATA0« (Slika 6.10) prikazuje pridobitev podatkov o artiklih (s pomočjo formata JSONata), ki jih zgoraj desno preoblikujemo (transformiramo) iz strukture, ki se uporablja v orodju TRONoffice, v strukturo, ki se uporablja na platformi Shopify.



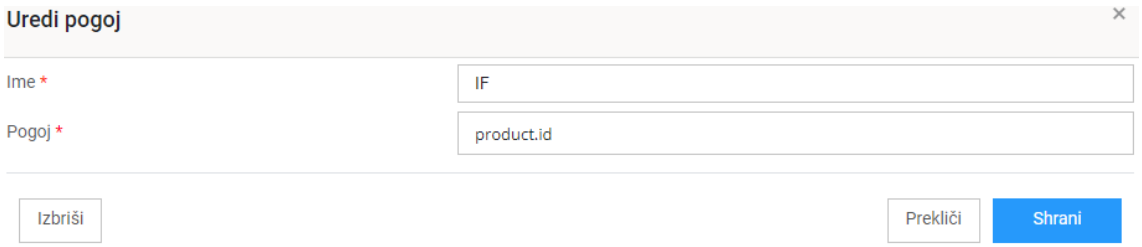
Slika 6.10: Korak DATA, ki prikaže pridobljene podatke in jih s pomočjo JSONata pretvori v strukturo, primerno za Shopify

Korak »STARTLOOP« (Slika 6.11) definira začetek zanke.



Slika 6.11: Korak STARTLOOP, ki določi začetek izvajanja zanke

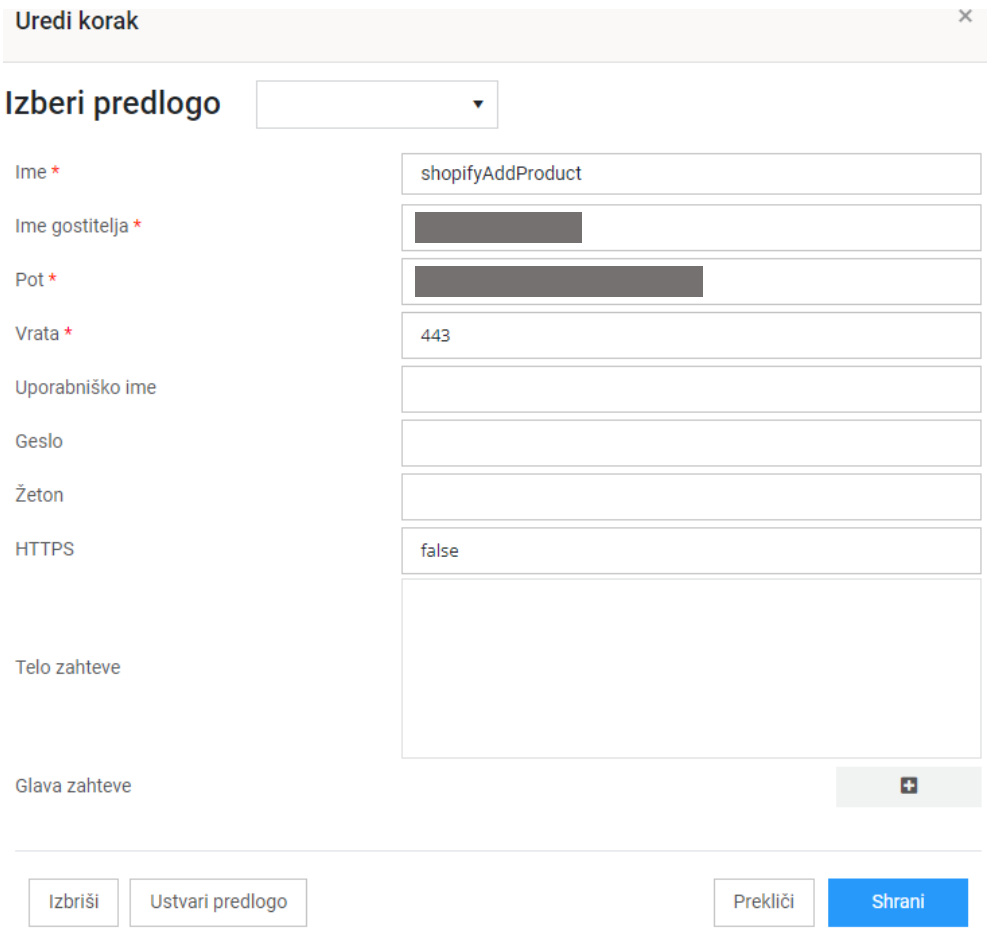
Korak »IF« (Slika 6.12) prikazuje določitev pogoja. Pot poteka je odvisna od ustreznosti pogoja.



Slika 6.12: Korak IF, ki določi pogoj

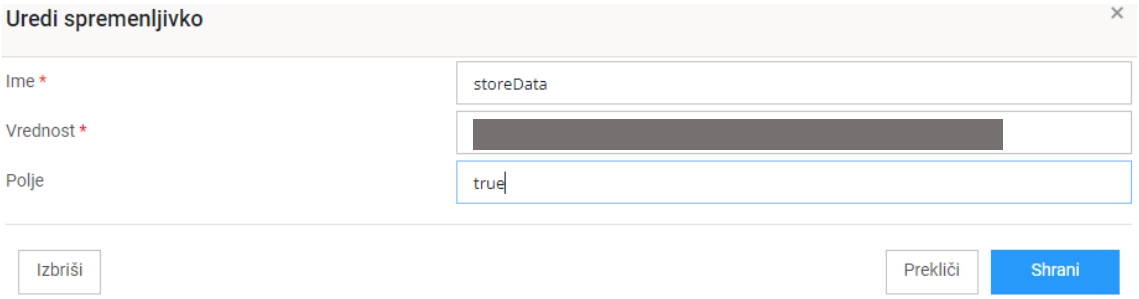
Korak »shopifyAddProduct« (Slika 6.13) prikazuje potrebne podatke za dodajanje artikla v Shopify. Treba je bilo vnesti ime, ime gostitelja, pot, vrata, HTTPS nastaviti na »false« in znotraj glave vnesti žeton. Korak se izvede v primeru, kadar je pogoj ustrezen.

Artikli se dodajajo s pomočjo zanke.



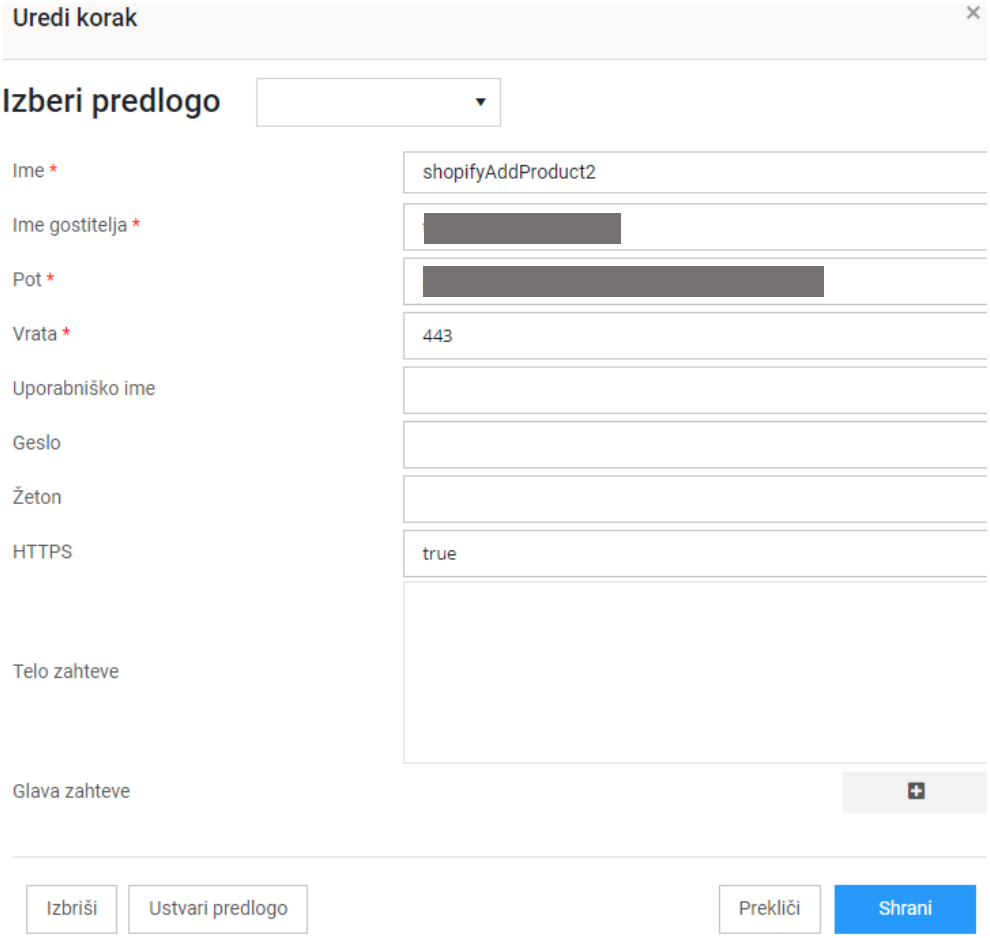
Slika 6.13: Korak POST, ki doda produkt v Shopify spletno trgovino

Korak »storeData« (Slika 6.14) prikazuje potrebne podatke za dodajanje oziroma posodobitev »exKey« v TRONoffice podatkovni bazi, ki predstavlja unikatni identifikator artikla v Shopify podatkovni bazi. Treba je bilo vnesti ime, polje in vrednost, ki predstavlja, kateri prejet podatek je treba shraniti kot podatek v TRONoffice podatkovni bazi.



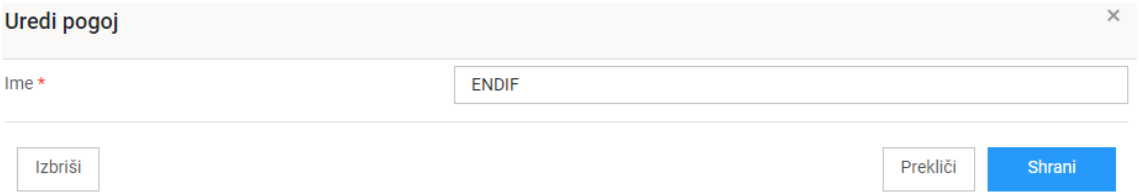
Slika 6.14: Korak VARIABLE, ki v seznam doda unikatni identifikator artikla, pridobljen iz platforme Shopify

Korak »shopifyAddProduct2« (Slika 6.15) prikazuje dodajanje produkta v Shopify, kar se zgodi znotraj zanke in v primeru, kadar je pogoj neustrezen. Treba je bilo vnesti ime, ime gostitelja, pot, vrata, HTTPS nastaviti na »true« in nastaviti žeton znotraj glave.



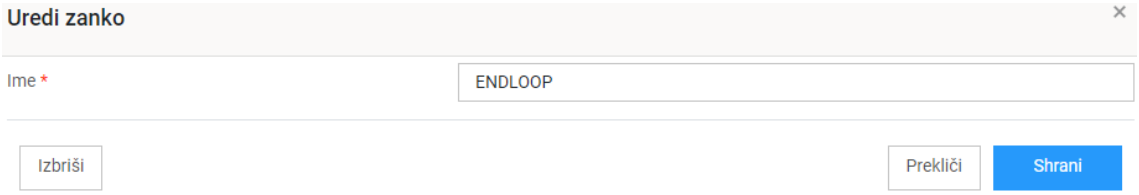
Slika 6.15: Korak PUT za posodobitev artiklov na platformi Shopify

Korak »ENDIF« (Slika 6.16) prikazuje konec območja, kjer velja pogoj.



Slika 6.16: Korak ENDIF, ki določi konec pogoja

Korak »ENDLOOP« (Slika 6.17) definira konec zanke.

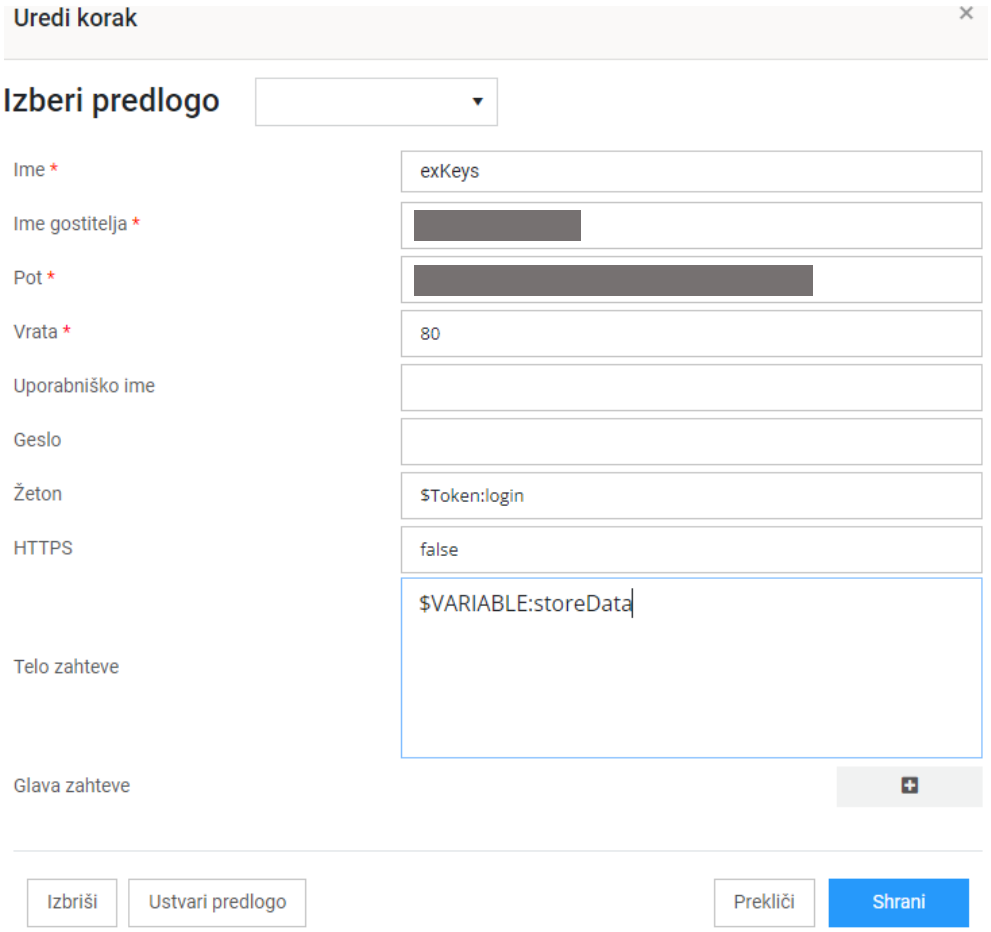


Slika 6.17: Korak ENDLOOP določi konec izvajanja zanke

Korak »STARTIF« (Slika 6.18) prikazuje določitev pogoja. Pot poteka je odvisna od ustreznosti pogoja. V tem primeru gre za pogoj, ali se dodajo oziroma posodobijo »exKey«, in sicer neodvisno od tega, ali je potrebno ali ne.

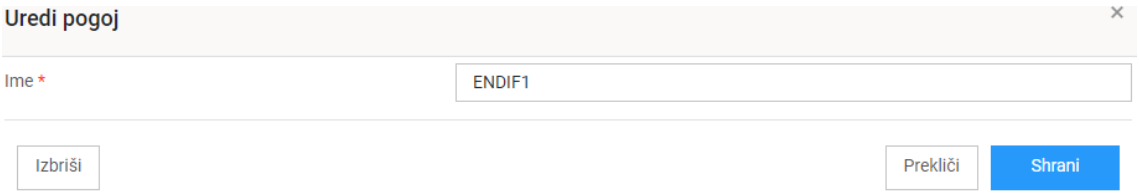
Slika 6.18: Korak STARTIF, ki določi pogoj

Korak »exKey« (Slika 6.19) prikazuje dodajanje oziroma posodobitev »exKey« v TRONoffice podatkovni bazi. Treba je bilo vnesti ime, ime gostitelja, pot, vrata, HTTPS nastaviti na »true«, nastaviti žeton, ki se pridobi iz koraka »login«, in telo zahteve, ki je v tem primeru spremenljivka, v kateri so shranjeni podatki (exKeys).



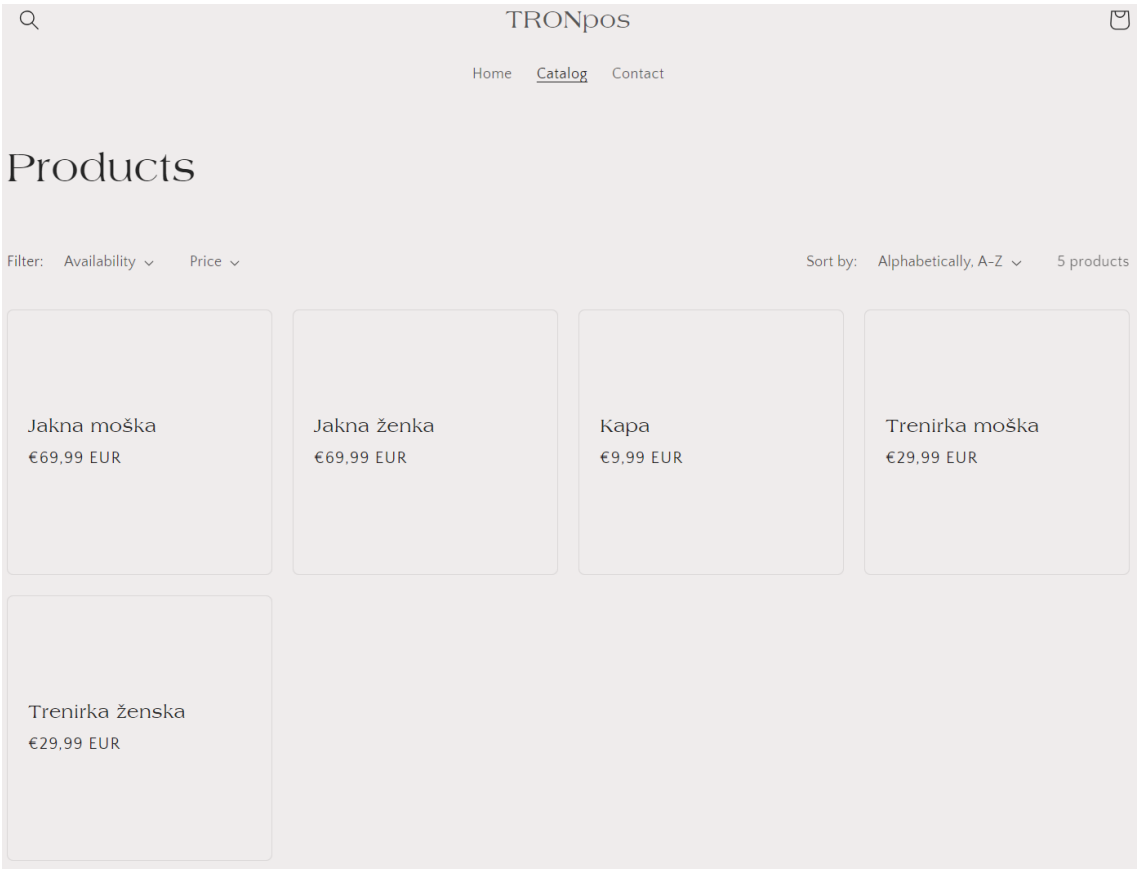
Slika 6.19: Korak POST pošlje seznam unikatnih identifikatorjev, shranjenih v spremenljivki »VARIABLE«

Korak »ENDIF« (Slika 6.20) prikazuje konec območja, kjer velja pogoj.



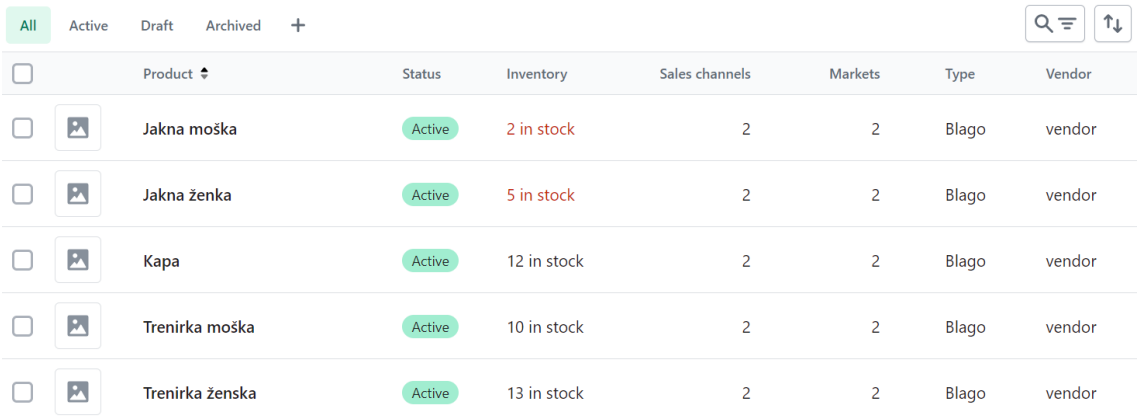
Slika 6.20: Korak ENDIF, ki določi konec pogoja

Pregled artiklov (Slika 6.21) prikazuje dodane artikle v Shopify, ki so vidni na spletni strani.



Slika 6.21: Artikli, prikazani na spletni strani Shopify

Nadzorna plošča artiklov (Slika 6.22) prikazuje artikle, ki jih je možno spreminjati.



Slika 6.22: Pregled artiklov v nadzorni plošči platforme Shopify

7 SKLEP

Sam produkt smo uspeli končati do te faze, kot smo si zastavili in planirali na začetku, ko smo zbirali ideje za izdelavo produkta. Orodje je podprto za sisteme, predloge korakov, skupine integracij in integracije. Omogočeno je izdelati integracijo po želji, ki na primer ob kliku »izvedi« doda, posodobi ali briše produkte. Trenutno smo podprli in testirali integracijo za sistema TRONoffice in Shopify. Med samim razvojem so se nam odprle nove poti in pojavile nove ideje, ki so nas pripeljale do zanimivih stvari in potreb, kaj bi bilo treba še dodelati in kako to storiti. Obsežno testiranje produkta je pokazalo potrebo po določenih popravkih in dodelavah, ki smo jih uspešno popravili in dodelali do te mere, da produkt deluje, kot mora. Produkt je še v fazi razvoja. Ta se bo nadaljeval v prihodnje nemoteno, dokler produkt ne bo pripravljen za izdajo. Zamisli za nadaljevanje razvoja seveda že imamo. Gre predvsem za dodelavo potrebnih stvari, ki smo jih opredelili kot manjkajoče, možne izboljšave in optimizacijo. Produkt bo treba v nekaterih pogledih težavnostno prilagoditi, da bo namenjen prav vsakomur. Sodelovanje z ostalo ekipo je bilo odlično in v sožitju – tako, kot mora biti. Spremljanje napredka je privedlo do razprav o izboljšavah in nadaljevanju dela. Pri svojem delu smo uporabili znanje, pridobljeno na fakulteti. Znanje ogrodja Angular in ogrodja Express smo skozi razvoj produkta nadgradili. Sedaj vemo, kje se kaj nahaja v samem produktu in kako stvari potekajo ter komunicirajo med seboj. Kendo UI, PostgreSQL, TypeORM in Syncfusion, kar so bile za nas nove stvari, smo dodobra spoznali, predvsem z uporabo dokumentacije, ki nam je bila v pomoč v času raziskovanja in razvoja samega produkta. Tudi te tehnologije smo sedaj že tako spoznali in uporabljali, da se znamo lotiti pristopa. Kot prvo smo naredili pregled in se seznanili s tehnologijami, ki smo jih nato v nadaljevanju uporabljali. Pregled dokumentacije in planiranje implementacije sta zelo pripomogla k dobremu pristopu in razvoju. Raziskava možnih orodij je pripeljala do knjižnice Syncfusion, ki je bila in je še vedno osrednji del produkta. Testiranje možnosti in vsega ostalega, kar nam knjižnica ponuja, nam je olajšalo kasnejše delo in implementacijo. Raziskava orkestracije mikrostoritev in primerjava s koreografskimi mikrostoritvami sta razjasnili razliko in podali lažjo odločitev glede izbire med njima. Razvoj po fazah se je izkazal kot dober

plan, saj nam je omogočil sprotno testiranje razvitega in nadgrajevanje delujočega. Nepopisen uspeh ob prvi integraciji je bil potrditev, da smo na pravi poti, kar nam je dodalo samo nove moči in željo po razširitvi in nadaljnjem razvoju. Skozi razvoj produkta in diplomsko delo smo se naučili predvsem dela v ekipi z ostalimi programerji. Izpolnjevanje točno določenih nalog po začrtanem planu se je izkazalo kot dobra praksa, ki nas je pripeljala do delujočega produkta, ki je še v fazi razvoja, in na poti do končnega produkta, ki se bo razvil do konca v prihodnjih mesecih.

8 VIRI IN LITERATURA

[1] Lockhart, T. (ur.). PostgreSQL Programmer's Guide. Dostopno na: https://cis.temple.edu/~vasilis/Courses/CS33/Documentation/programmer.pdf [26.05.2023].

[2] Schwarzmüller, M. Angular – The Complete Guide (2023 Edition). Dostopno na: https://www.udemy.com/course/the-complete-guide-to-angular-2/ [26.05.2023].

[3] Amazon Simple Queue Service. Dostopno na: https://aws.amazon.com/sqs/ [12.05.2023].

[4] Angular Documentation. Dostopno na: https://angular.io/docs [29.04.2023].

[5] API2CART Dostopno na: https://api2cart.com/ [25.04.2023].

[6] CDATA Dostopno na: https://www.cdata.com/ [25.04.2023].

[7] Express Documentation. Dostopno na: https://expressjs.com/ [30.04.2023].

[8] Ionic framework. Dostopno na: https://ionicframework.com/ [29.04.2023].

[9] JSON query and transformation language. Dostopno na: https://jsonata.org/ [26.04.2023].

[10] Kendo UI for Angular Documentation. Dostopno na: https://www.telerik.com/kendo-angular-ui/components/ [29.04.2023].

[11] Kubernetes. Dostopno na: https://kubernetes.io/ [10.05.2023]

[12] Microservice Orchestration. Dostopno na: https://medium.com/trueengineering/a-review-of-microservice-orchestration-frameworks-d22797b34ea5 [10.05.2023].

[13] Microservice Orchestration. Dostopno na: https://miro.medium.com/v2/resize:fit:720/format:webp/0*vxdHQ-skF9tiWCcQ.png [10.05.2023].

[14] Orchestration vs Choreography. Dostopno na: https://blog.sparkfabrik.com/hubfs/Blog/orchestration-vs-choreography-pro-contro.png [11.06.2023].

[15] Orchestration vs Choreography, which one should you use? The pros and cons

Dostopno na: https://blog.sparkfabrik.com/en/orchestration-vs-choreography [11.05.2023].

[16] PostgreSQL Documentation. Dostopno na: https://www.postgresql.org/docs/ [29.04.2023].

[17] Postman. Dostopno na: https://www.postman.com/ [30.04.2023].

[18] RabbitMQ. Dostopno na: https://www.rabbitmq.com/ [12.05.2023].

[19] Shopify API Documentation. Dostopno na: https://shopify.dev/docs/api [24.05.2023].

[20] Syncfusion Angular UI Documentation. Dostopno na: https://ej2.syncfusion.com/angular/documentation/introduction [5.05.2023].

[21] Syncfusion components Dostopna na: https://cdn.syncfusion.com/content/images/home-v1/home/home-banner-mobile-v4.png [24.06.2023].

[22] TRAY.IO Dostopno na: https://tray.io/ [25.04.2023]

[23] TypeORM Documentation. Dostopno na: https://typeorm.io/ [30.04.2023].

[24] What is a DTO? Dostopno na: https://www.okta.com/identity-101/dto/ [26.05.2023].

[25] Why is Microservice Orchestration Important Now? Dostopno na: https://orkes.io/blog/why-is-microservice-orchestration-important-now/ [11.05.2023]