UNIVERZA V LJUBLJANI
Fakulteta za strojništvo

Razvoj vmesnika za vpis naročil in generiranje vhodnih podatkov za digitalnega dvojčka proizvodnega procesa

Zaključna naloga Univerzitetnega študijskega programa I. stopnje Strojništvo – Razvojno raziskovalni program

Nikola Vukas

Ljubljana, september 2023

UNIVERZA V LJUBLJANI
Fakulteta za strojništvo

Razvoj vmesnika za vpis naročil in generiranje vhodnih podatkov za digitalnega dvojčka proizvodnega procesa

Zaključna naloga Univerzitetnega študijskega programa I. stopnje Strojništvo – Razvojno raziskovalni program

Nikola Vukas

Mentor: doc. dr. Marko Šimic, univ. dipl. inž.
Somentor: prof. dr. Niko Herakovič, univ. dipl. inž.

Ljubljana, september 2023

Univerza v Ljubljani
Fakulteta za strojništvo

Aškerčeva cesta 6
1000 Ljubljana, Slovenija
telefon: (01) 47 71 200
faks: (01) 25 18 567
dekanat@fs.uni-lj.si

PRIJAVA TEME ZAKLJUČNEGA DELA

Študent: Nikola Vukas

Študijski program: Univerzitetni študijski program prve stopnje Strojništvo - Razvojno raziskovalni program

Številka teme zaključnega dela: UN I/1888

Naslov teme zaključnega dela v slovenskem jeziku: Razvoj vmesnika za vpis naročil in generiranje vhodnih podatkov za digitalnega dvojčka proizvodnega procesa

Naslov teme zaključnega dela v angleškem jeziku: Interface development for capturing orders and generating input data for the digital twin of the manufacturing process

Dispozicija zaključnega dela:
V sodobni industriji, kjer sta produktivnost in kakovost izdelkov izjemnega pomena, raziskave na področju digitalnih dvojčkov in industrije 4.0 kažejo potrebo po ustvarjanju intuitivnega sistema za izbiro izdelkov s strani kupcev. V tej nalogi se zato osredotočite na razvoj in izdelavo digitalnega vmesnika za naročilo predvidenih izdelkov in shranjevanje ustreznih podatkov, ki bodo primerni za digitalnega dvojčka proizvodnega procesa. Vmesnik razvije z uporabo platforme ThingsBoard. V začetni fazi dajte poudarek na oblikovanju vizualne identitete vmesnika, prilagojenega za tri predvidene izdelke. Pri tem uporabite obstoječa orodja in knjižnice, ki so vključene v platformo ThingsBoard. Po potrebi razvijte nove programske kode za doseganje želene funkcionalnosti vmesnika. Uporabite znanje programiranja s HTML, CSS in JavaScript-om za dodajanje novih funkcionalnosti. Osrednji del naloge naj predstavlja izdelavo diagrama verige pravil, ki bo izboljšala funkcionalnost našega vmesnika, to je ustrezno zbiranje, spreminjanje in shranjevanje informacij in generiranih podatkov naročil. V verigi pravil naj bodo informacije o naročilu strukturirane tako, da jih bo mogoče enostavno posredovati naprej do digitalnega dvojčka proizvodnega procesa, razvitega v orodju Technomatix Plant Simulation.

Mentor: doc. dr. Marko Šimic

Somentor: prof. dr. Niko Herakovič

Datum odobrene teme: 05.09.2023

UNIVERZA V LJUBLJANI
Fakulteta za strojništvo
1
Ljubljana

Opomba: Zaključno delo je treba oddati v jezikovno in terminološko pravilnem slovenskem jeziku. Rok za oddajo zaključnega dela v informacijski sistem VIS je šest mesecev od odobrene teme.

Zahvala

Na prvem mestu se želim zahvaliti doc. dr. Marku Šimicu in prof. dr. Niku Herakoviču, za mentorstvo ter pomoč in usmeritev pri pisanju te zaključne naloge.

Nato se iskreno zahvaljujem še svoji družini, ki so stalno verjeli v moje sposobnosti in me skozi vso pot podprli tako duhovno kot finančno.

Izvleček

UDK 004.5:658.51:004.896(043.2)

Tek. štev.: UN I/1888

Razvoj vmesnika za vpis naročil in generiranje vhodnih podatkov za digitalnega dvojčka proizvodnega procesa

Nikola Vukas

Ključne besede: digitalni vmesnik
proizvodno naročilo
IoT
izdelki
thingsboard
javascript
html
css

V okviru raziskav na področju digitalnih dvojčkov smo identificirali potrebo po ustvarjanju intuitivnega in učinkovitega načina izbire izdelkov. Da bi naslovili ta izziv, smo se odločili za uporabo IoT platforme ThingsBoard. Ključna prizadevanja so bila usmerjena v oblikovanje virtualne identitete vmesnika za tri izbrane izdelke, ki združuje estetsko privlačnost in funkcionalnost. Z vključevanjem in prilagajanjem integriranih gradnikov platforme (pripomočkov), ob podpori knjižnice JQuery ter programskih jezikov HTML in CSS, smo dosegli želeni uporabniški vmesnik. Nadalje smo razvili verigo pravil, z uporabo programskega jezika JavaScript in orodja Node-RED, kot tudi dokumentacije ThingsBoard platforme. Naša veriga pravil omogoča pretvorbo podatkov v ustrezen format JSON, pripravljen za interakcijo z našim digitalnim dvojčkom. Ta razvoj poudarja pomembnost prilagodljivih in dinamičnih rešitev v sodobnem digitalnem okolju.

Abstract

UDC 004.5:658.51:004.896(043.2)

No.: UN I/1888

Razvoj vmesnika za vpis naročil in generiranje vhodnih podatkov za digitalnega dvojčka proizvodnega procesa

Nikola Vukas

Key words: graphical interface
production order
IoT
products
thingsboard
javascript
html
css

In the context of research in the field of digital twins, we identified the need to create an intuitive and effective way of selecting products. To address this challenge, we opted to use the IoT platform ThingsBoard. Key efforts were focused on designing a virtual identity interface for three selected products, combining aesthetic appeal with functionality. By incorporating and adapting integrated components of the platform (widgets), with the support of the JQuery library and the programming languages HTML and CSS, we achieved the desired user interface. Furthermore, we developed a rule chain, using the JavaScript programming language and the Node-RED tool, as well as the documentation of the ThingsBoard platform. Our rule chain enables the conversion of data into the appropriate JSON format, ready for interaction with our digital twin. This development underscores the importance of adaptable and dynamic solutions in a modern digital environment.

Kazalo

Kazalo slik

Seznam uporabljenih okrajšav

Okrajšava	Pomen
API	Application Programming Interface
CSS	Cascading Style Sheets
DOM	Document Object Model
HTML	HyperText Markup Language
IoT	Internet of Things
REST	Representational state transfer
RFID	Radio Frequency Identification
Wi-Fi	Wireless Fidelity

1 Uvod

1.1 Ozadje problema

V dobi digitalizacije in hitrega razvoja tehnologije je zagotavljanje učinkovitega in uporabniku prijaznega vmesnika za izvedbo naročil izdelkov postalo ključno, še posebej pri spletnem nakupovanju. Izziv ni samo v ustvarjanju vizualno privlačnega in intuitivnega vmesnika, ampak tudi v funkcionalnosti vmesnika kar se odraža v obdelavi podatkov in prenosu kompleksnih informacij na ustrezne sisteme v realnem času. Prav v tem kontekstu je prišlo do potrebe po razvoju specifičnega vmesnika na IoT platformi ThingsBoard. Ta vmesnik bo uporabnikom ponudil tri ključne skupine izdelkov: vzorec obarvanih kock, gravure in rezervni deli. Poleg tega, da bo uporabnikom omogočil intuitivno izbiro izdelkov in prijetno spletno nakupovalno izkušnjo, bo ta vmesnik zagotovil tudi učinkovito komunikacijo z našim digitalnim dvojčkom proizvodnega procesa, izdelanim v simulacijskem orodju Tecnomatix Plant Simulation.

1.2 Cilji

V okviru tega diplomskega dela je glavni cilj načrtovanje in izvedba funkcionalnega vmesnika za naše tri omenjene izdelke. Čeprav platforma, na kateri delujemo, ni bila primarno zasnovana za ta namen, nam ponuja orodja in vire, ki bodo ključnega pomena za to pobudo. Zaradi takšnih tehničnih okoliščin nas čakajo izzivi pri ustvarjanju uporabniškega vmesnika, ki bi imel lastnosti, podobne standardnim spletnim trgovinam.

Naša prva točka razmisleka je ustvarjanje vizualne identitete, ki bi primerno predstavljala naše izdelke, hkrati pa zagotavljala enostavno in učinkovito interakcijo za končne uporabnike. Znotraj platforme ThingsBoard so predvidene modifikacije obstoječe kode pripomočka (Ang.: *Widget*) za dosego večje vizualne privlačnosti in optimizacijo uporabniške izkušnje.

Po uspešni implementaciji vmesnika je osrednja naloga integracija in strukturiranje informacij, podatkov, ki jih je treba nadalje zagotoviti kot ustrezne vhodne podatke simulacijskemu modelu proizvodnega procesa in digitalnemu dvojčku. Ta faza je, glede na kompleksnost platforme, prinesla svoje izzive. To vključuje, vendar ni omejeno na,

združevanje in oblikovanje informacij in podatkov za nadaljnjo obdelavo, omogočanje izbora več vrst obarvanih kock, razvoj sistema za sledenje zgodovini naročil in učinkovit mehanizem za brisanje izbranih izdelkov.

Zaključno, naš namen je razviti vmesnik, ki, čeprav je ustvarjen v specifičnem tehničnem okolju, ima vse ključne značilnosti, ki bi jih pričakovali od moderne spletne trgovine.

2 Teoretične osnove in pregled literature

2.1 Uvod v IoT

Internet stvari ali IoT predstavlja revolucionarni koncept, v katerem so vsakdanji predmeti povezani z internetom, kar jim omogoča komunikacijo med seboj in z drugimi digitalnimi sistemi. Ta ideja spremeni običajne predmete, kot so žarnice, ure ali celo čevlji, v pametne naprave (Ang.: *Smart Devices*), ki so sposobne zbirati, pošiljati in prejemati podatke. Zahvaljujoč tehnologijam, kot so Wi-Fi, Bluetooth in RFID, postaja svet okoli nas vedno bolj povezan, kar nam ponuja nove možnosti in preoblikuje način, kako živimo in delamo. [1]

2.2 Digitalni vmesnik ThingsBoard

ThingsBoard je odprta platforma za IoT, ki uporabnikom omogoča zbiranje, obdelavo, analizo in vizualizacijo podatkov iz IoT naprav (Ang.: *Devices*) v realnem času. S nodularno zasnovo in naborom integriranih orodij ThingsBoard ponuja fleksibilnost pri izdelavi uporabniških vmesnikov in upravljanju naprav. Platforma prav tako ponuja možnost ustvarjanja verige pravil (Ang.: *Rule Chain*) in obvestil, kar omogoča avtomatizacijo in odzivanje na določene dogodke ali pogoje. Ta funkcija nam bo posebej koristna pri naši nalogi. Uporaba takšne platforme lahko bistveno pospeši razvoj in implementacijo IoT rešitev, zmanjšuje čas in stroške razvoja.[2]

2.2.1 Naprava

Da bi optimizirali uporabo ThingsBoarda, je ključnega pomena razumeti koncept naprave (Ang.: *Device*)znotraj platforme. V tradicionalnem kontekstu bi ta naprava predstavljala fizično enoto, ki jo želimo upravljati ali iz katere zbiramo informacije. Vendar pa v določenih implementacijah, kot je naša, fizična prisotnost naprave ni nujna. Ne glede na to, ustvarjanje virtualne naprave znotraj platforme ThingsBoard postane ključno, da lahko vzpostavimo osnovo za obdelavo in upravljanje podatkov.

Znotraj ekosistema ThingsBoard je vsaka naprava povezana z nizom atributov, ki definirajo njene lastnosti in vedenje. Atributi delujejo kot metapodatki za napravo, zagotavljajoći podrobne informacije o njenem statusu, konfiguraciji in drugih relevantnih informacijah. Prav ti atributi bodo postali osnova za shranjevanje izbranih informacij s strani uporabnikov.

2.2.2 Shranjevanje podatkov v ThingsBoard oblak

ThingsBoard uporablja sodobne oblačne tehnologije, da bi učinkovito shranjeval in upravljal z atributi naprave (Ang.: *Device Attriutes)*. Ko naprava pošlje podatke, se ti avtomatsko shranijo znotraj ThingsBoard-ove oblačne infrastrukture, uporabljajoč optimizirane podatkovne baze. V našem konkretnem primeru, ker fizična naprava ni prisotna, nismo morali obravnavati specifičnih protokolov za komunikacijo z oblakom, saj je ThingsBoard že imel integrirane standardne protokole, ki so bili več kot ustrezni za naše potrebe.

2.2.3 Pripomčki

Upravljanje informacij v ThingsBoardu poteka centralizirano preko nadzorne plošče (Ang.: *Dashboard*). Uporabniki lahko dodajo, odstranijo ali prilagodijo pripomočke (Ang.: *Widgets*) na svoji nadzorni plošči glede na svoje potrebe. Ta prilagodljivost omogoča uporabnikom, da prilagodijo svoj vmesnik natanko tako, kot si želijo, zagotavljajoč, da imajo dostop do vseh ključnih informacij na enem mestu.

Pripomočki (Ang.: *Widgets*) so majhne grafične komponente, ki lahko na različne načine prikazujejo informacije. Lahko segajo od preprostih indikatorjev stanja do zapletenih grafov ali zemljevidov. Čeprav ThingsBoard vključuje veliko pripomočkov, bomo predstavili samo tiste, ki so bili ključni v našem delu:

Posodobi več atributov (Ang.: *Udate Multiple Attributes Widget*) : Ta pripomoček omogoča uporabnikom, da hkrati posodabljajo več atributov določene naprave. To je še posebej uporabno, kadar je potrebno hitro spremeniti ali sinhronizirati več parametrov. Intuitivni vmesnik omogoča enostavno izbiro in spremembo želenih atributov, s čimer se poveča učinkovitost in natančnost upravljanja naprave.

Kartica atributov (Ang.: *Attributes Card Widget*): Ta pripomoček ponuja uporabnikom vizualni prikaz ključnih atributov naprave v obliki kartice. Vsak atribut je jasno označen s svojo vrednostjo, kar to komponento naredi idealno za hiter pregled stanja ali lastnosti naprave. Poleg tega se lahko oblikovanje kartice prilagodi tako, da je v skladu s specifičnimi potrebami uporabnika ali estetiko nadzorne plošče.

Posodobi deljene atribute (Ang.: *Update Shared Attributes Widget*): Z uporabo tega pripomočka imajo uporabniki možnost posodobiti atribute, ki so skupni več napravam.
HTML kartica (Ang.: *HTML Card Widget*): Ta pripomoček omogoča uporabnikom prikaz informacij ali podatkov na digitalnem vmesniku ThingsBoard v obliki HTML kartice. Z možnostjo prilagajanja vsebine HTML-a lahko uporabniki prilagodijo vizualne prikaze, vključno s tekstom, slikami ali drugimi multimedijskimi elementi, da najbolje ustrezajo

njihovim potrebam. Ta pripomoček je postal ključen za izboljšanje uporabniške izkušnje in za zagotavljanje fleksibilnosti pri prikazovanju podatkov na platformi ThingsBoard.
Posodobi atribut naprave (Ang.:*Update Device Attribute Widget*): Ta pripomoček je specializirano orodje, ki uporabnikom omogoča posamezno posodobitev specifičnega atributa naprave. Ta pripomoček se je izkazal za še posebej koristnega v procesu ustvarjanja naše verige pravil.

2.2.4 Veriga pravil

Veriga pravil (Ang.: *Rule Chain*) na platformi ThingsBoard predstavlja močno orodje za obdelavo podatkov v realnem času in njihovo distribucijo. V osnovi je veriga pravil logično povezana, kar omogoča da podatke obdelamo v večih korakih po točno določenem zaporedju operacij, da podatke dobimo v pravem zapisu in strukturi potrebni za uporabo pri končnem porabniku, napravi. Vsako pravilo v verigi lahko prevzame vhodne podatke, izvede določeno operacijo na njih (kot je filtriranje, transformacija ali agregacija) in nato pošlje obdelane podatke naslednjemu pravilu v verigi.

Ena od ključnih prednosti zaporedja pravil v ThingsBoard-u je fleksibilnost. Uporabniki lahko preprosto dodajajo, spreminjajo ali odstranjujejo pravila brez ogrožanja preostalega postopka obdelave. S tem sistemom ThingsBoard uporabnikom ponuja močno orodje za avtomatizacijo in optimizacijo procesa obdelave podatkov.

2.2.4.1 Niz vozlišč (Node RED)

Node-RED je orodje, ki temelji na vizualnem programiranju in uporabnikom omogoča oblikovanje in implementacijo podatkovnih tokov z uporabo niza vozlišč (Ang.: *Nodes*). Vsako vozlišče predstavlja določeno funkcionalnost ali nalogo. Po drugi strani pa je ThingsBoard odprtokodna IoT platforma, ki nudi različne možnosti, vključno z upravljanjem naprav, zbiranjem podatkov in vizualizacijo. Povezanost med Node RED-om in ThingsBoard-om je v možnosti uporabe Node-RED-a kot posrednika za obdelavo in integracijo podatkov pred njihovim pošiljanjem na platformo ThingsBoard. Ta kombinacija nudi fleksibilen in močan način za obdelavo in vizualizacijo podatkov v IoT okolju.

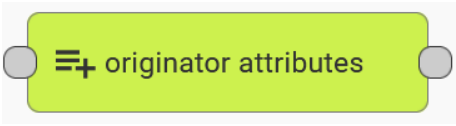
Na kratko, Node-RED je orodje, ki uporabnikom omogoča, da vizualno ustvarjajo podatkovne tokove s povezovanjem različnih vozlišč.[3]

2.2.4.2 Vozlišča

Ustvarjanje verige pravil (Ang.: *Rule Chain*) na platformi ThingsBoard temelji na nodularnem principu, kar omogoča vizualno in intuitivno gradnjo kompleksnih procesov obdelave podatkov. Razumevanje in smotrna uporaba različnih vozlišč (Ang.: *Nodes*) je bila ključna za optimizacijo našega programa. Poglejmo podrobneje glavna vozlišča, ki smo jih uporabljali:

- Vhodno vozlišče (Ang.: *Input Node*): To vozlišče je komponenta, ki prejema podatke od naprav ali drugih virov. Služi kot vstopna točka za podatke, jih pretvarja in posreduje naprej znotraj sistema za nadaljnjo obdelavo ali vizualizacijo. V osnovi je most med napravo in platformo, zagotavljajoč, da se podatki natančno in učinkovito integrirajo.

- Vozlišče za obogatitev podatkov o izvoru (Ang.: *Enrichment Originator Attributes Node*): To vozlišče se uporablja za obogatitev podatkov o izvoru. V praksi nam omogoča, da dodamo ali kombiniramo trenutne atribute naprave (Ang.: *Device Attributes)* z drugimi relevantnimi informacijami znotraj sistema. To vozlišče spada v skupino obogatitvenih vozlišč (Ang.: *Enrichement Nodes*), prikazan na sliki 2.1, ki so prepoznavni po svoji zeleni barvi.

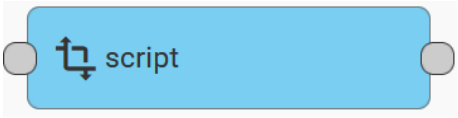


Slika 2.1 : Vozlišče za obogatitev podatkov o izvoru.

- Vozlišče za filtriranje poatkov (Ang.: *Filter Script Node*): S tem vozliščem lahko definiramo pogoje za filtriranje podatkov, ki potekajo skozi verigo pravil. Z uporabo preprostih skript lahko odločimo, kateri podatki bodo prepuščeni naprej v verigo, in kateri bodo ustavljeni. To je še posebej koristno, ko želimo obdelati samo določene podatke, ki izpolnjujejo specifične kriterije. To vozlišče spada v skupino filter vozlišč (Ang.: *Filter Nodes*), ki so prepoznavni po svoji rumeni barvi kot se vidi na sliki 2.2.

Slika 2.2 : Vozlišče za filtriranje podatkov.

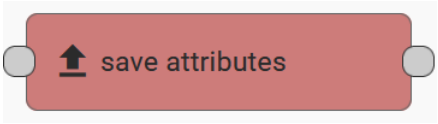
- Vozlišče transformacijo podatkov (Ang.: *Transformation Script Node*): Tukaj se podatki lahko spremenijo ali transformirajo, preden nadaljujejo k naslednjemu koraku v verigi. To vozlišče se lahko uporablja za prilagajanje formata podatkov, kombiniranje informacij, izračun novih vrednosti ali celo za ustvarjanje popolnoma novih nizov podatkov na podlagi vhodnih informacij. Ono je vključeno v skupino transformacijskih vozlišč (Ang.: *Transformation Nodes*), ki so prepoznavni po svoji modri barvi kot na sliki 2.3.



Slika 2.3 : Vozlišče za transformacijo podatkov.

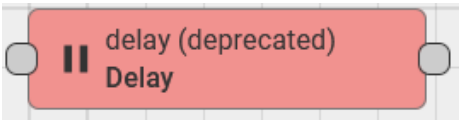
- Vozlišče za shranjevanje in posodabljanje atributov (Ang.: *Save Attributes Node*) : Kot ime že nakazuje, to vozlišče omogoča shranjevanje ali posodabljanje atributov naprave. Potem ko podatki preidejo procese obogatitve, filtriranja in transformacije, ono omogoča, da se rezultati shranijo neposredno na določeno napravo, s čimer se zagotovi, da so najnovejše informacije na voljo in posodobljene v realnem času. To vozlišče spada v

skupino akcijskih vozlišč (Ang.: *Action Nodes*), ki so prepoznavni po svoji rdeči barvi, to je vidno na sliki 2.4.



Slika 2.4 : Vozlišče za shranjevanje in posodabljanje atributov.

- Vozlišče za časovno zakasnitev (Ang.: *Delay Node*): Osnovna njegova funkcija je zagotavljanje časovne zakasnitve procesa. Konkretno, ko so bile določene naloge dokončane in podatki so bili pripravljeni za brisanje, nam je omogočal, da zagotovimo dodaten čas. To vozlišče prav tako spada v skupino akcijskih vozlišč (Ang.: *Action Nodes*), prikazan je na sliki 2.5.



Slika 2.5 : Vozlišče za časovno zakasnitev.

Ta vozlišča, čeprav predstavljajo le del obsežne palete orodij, ki jih ThingsBoard ponuja, so bili ključni za ustvarjanje prilagojene in učinkovite verige pravil, ki je zagotovil naše specifične potrebe.

2.3 JavaScript

JavaScript je dinamičen programski jezik, ki omogoča interaktivnost na spletnih straneh. Vendar je sprva bil razvit kot del Netscape Navigatorja, hitro je pridobil priljubljenost in postal nujno orodje za spletne razvijalce po vsem svetu. On omogoča ustvarjanje dinamičnih funkcij, kot so obdelava dogodkov, manipulacija z vsebino in komunikacija s spletnimi strežniki. Da bi kar najbolje izkoristili možnosti platforme ThingsBoard, je osnovno znanje JavaScript-a ključno, še posebej, ko gre za prilagajanje in spreminjanje pripomočkov (Ang.: *Widgets*).[4]

2.3.1 Tipi podatkov v JavaScript-u

JavaScript podpira različne tipe podatkov, ki jih lahko grobo razdelimo na primitivne in objektne tipe.

- Primitivne tipe podatkov običajno predstavljajo število (Ang.: *Number*), besedni niz (Ang.: *String*) in logična vrednost (Ang.: *Boolean*). Number je tip podatkov, ki predstavlja tako cele številke kot številke s plavajočo vejico (Ang.: *Float*). String se nanaša na niz znakov in se uporablja za predstavitev tekstualnih podatkov. Ono je običajno postavljeno med dvojne ali enojne narekovaje. Boolean, na drugi strani, ima

samo dve vrednosti: resnično (Ang.: *True*) in neresnično (Ang.: *False*) ter se uporablja za predstavitev logičnih vrednosti ali za testiranje pogojev.
- Po drugi strani pa objekti v JavaScriptu, kot so objekt (Ang.: *Object*), niz (Ang.: *Array*) in funkcija (Ang.: *Function*), nudijo zapletenejše strukture, ki lahko vsebujejo več vrednosti ali funkcionalnosti. Razumevanje teh osnovnih tipov podatkov je ključno za učinkovito programiranje v JavaScriptu. Poznavanje teh tipov podatkov in upravljanje z njimi je nujno.

Oglejmo si še 2 pomembna koncepta, ki nam bosta pomagala pri našem delu.

2.3.2 Funkcije v JavaScriptu

Funkcije so ključni elementi programiranja v JavaScriptu. Omogočajo nam, da strukturiramo zapletenejše programe in zmanjšamo redundanco. Medtem ko povprečna oseba, ki govori angleško, ima približno 20.000 besed v svojem besedišču, programski jeziki nimajo toliko vgrajenih ukazov. Zato funkcije v JavaScriptu služijo kot sredstvo za ustvarjanje nove, prilagojene terminologije, kar nam omogoča, da učinkovito izrazimo svoje ideje brez nepotrebnega ponavljanja[4].

Pisanje niza »Hello World« je za mnoge študente pogosto prvi korak v svet programiranja. Oglejmo si zdaj preprosto funkcijo, ki bi to predstavila:

```
function pozdrav() {
console.log(„Hello World!“);
}
pozdrav();  // Izpisuje: Hello World!
```

console.log() je metoda v JavaScriptu, ki se uporablja za izpisovanje sporočil ali vrednosti v konzoli brskalnika. Ko pokličemo console.log() z določenim argumentom, se ta argument izpiše kot sporočilo v konzoli.

S to programsko kodo smo ustvarili funkcijo, brez zahtevanih argumentov, z imenom »pozdrav«, ki izvede izpis niza »Hello World«.

Oglejmo si še funkcijo, ki zahteva argumente:

```
function sestej(a, b) {
  return a + b;
}
console.log(sestej(3, 4));  // Izpisuje: 7
```

V tej programski kodi kličemo funkcijo z imenom »seštej«, ki kot argumente vzame števila (a=3, b=4) in jih sešteva. Te funkcije predstavljajo le vpogled v njihovo vsestranskost in uporabnost v svetu programiranja.

2.3.3 Objekti

Primer programiranja za objekte:

```
let oseba = {
  ime: „Nikola“,
  priimek: „Vukas“,
  let: 23,
  pozdrav: function() {
    console.log(„Zivjo, jaz sem „ + this.ime);
  }
};
```

Objekti v JavaScriptu predstavljajo osnovni način strukturiranja in organizacije podatkov. Objekti so, preprosto povedano, zbirke ključ-vrednost. Ključ je vedno niz, medtem ko lahko vrednost predstavlja katerakoli vrednost v JavaScriptu, vključno z drugimi objekti ali funkcijami. Na primeru programiranja za objekte smo ustvarili objekt z imenom »oseba«. Ta objekt vsebuje ključe: ime, priimek, let in pozdrav, ter vrednosti: Nikola, Vukas, 23 in funkcija. Sedaj, ko smo ustvarili objekt, bi bilo smiselno dostopati do njegovih informacij ali vrednosti. To lahko storimo na preprost način:

console.log(oseba.ime); // Izpiše: Nikola

oseba.pozdrav(); // Izpiše: Živjo, jaz sem Nikola
Sintaksa »objekt.ključ« nam omogoča neposreden dostop do vrednosti tega ključa. Kar nam bo koristilo v nadaljnjem delu.

2.3.4 JavaScript knjižnica (jQuery) in DOM

V kontekstu programiranja in JavaScripta knjižnice predstavljajo zbirke vnaprej napisane kode, ki jih lahko programerji uporabljajo za poenostavitev in pospešitev razvoja svojih aplikacij. Namesto da bi programer pisal vso kodo od začetka, knjižnice omogočajo uporabo že testirane in optimizirane kode, da se doseže želena funkcionalnost.

Eden od najbolj znanih primerov JavaScript knjižnice je prav jQuery, vendar obstaja na tisoče drugih knjižnic, ki ponujajo različne funkcionalnosti, od grafičnega oblikovanja do interakcije z bazami podatkov.

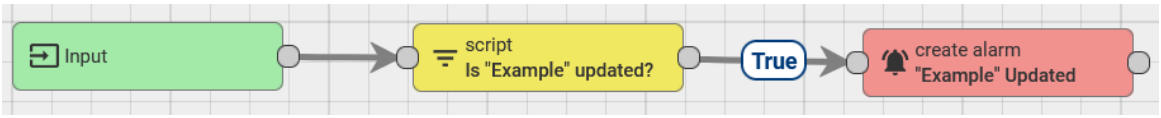
jQuery je priljubljena JavaScript knjižnica, ki je bila razvita z namenom olajšanja manipulacije z HTML dokumentom in obdelave dogodkov v spletnem okolju. Omogoča preprosto izbiranje in spreminjanje elementov v Document Object Modelu ali DOM, ter nudi orodja za učinkovito ravnanje z interaktivnimi dogodki, kot so kliki ali spremembe obrazcev.

DOM je programski vmesnik za spletne dokumente. V bistvu DOM predstavlja strukturiran prikaz dokumentov in omogoča programom manipulacijo z vsebino, strukturo in slogom spletnih strani. Preko DOM-a je dokument predstavljen kot hierarhično drevo vozlišč, kjer vsako vozlišče predstavlja del strani (npr. element, atribut ali besedilno vsebino).

Ko spletni brskalnik naloži HTML dokument, ustvari DOM tega dokumenta, kar omogoča JavaScript kodi dostop in spreminjanje vsebine in videza te strani dinamično, v realnem času.[5]

2.4 Primer zaporedja pravil – Javascript

Sedaj ko smo seznanjeni z osnovami javascripta in zaporedja pravil, pokažimo njihovo uporabo na primeru s slike 2.6.



Slika 2.6 : Primer zaporedja pravil.

Na sliki 2.6 je vizualno prikazana osnovna struktura, ki vključuje vhodno vozlišče (Ang.: *Input Node*), vozlišče za filtriranje podatkov (Ang.: *Filter Script Node*) z naslovom »Is „Example“ updated? « in alarm vozlišče (Ang.: *Alarm Node*) z naslovom »Example has been updated«. Cilj omenjene strukture je preveriti, ali je bila v neimenovani napravi (Ang.: *Device)* posodobljena vrednost atributa imenovanega kot Example«. V primeru potrditve posodobitve, vozlišče za filtriranje podatkov vrne vrednost »True«. To povzroči aktivacijo naslednjega vozlišča, ki ustvari alarm s sporočilom, da je atribut »Example« posodobljen. Podrobneje si bomo ogledali kodo znotraj filter vozlišča in njeno delovanje.

```
msg.Example !== undefined;
```

Omenjena koda je napisana v jeziku JavaScript in deluje na objektu »msg«. Ta pristop k obdelavi podatkov v JavaScript-u, zlasti z objektom »msg«, nakazuje, da je bil koda napisan za uporabo znotraj tako imenovanega »Function« vozlišča v Node-RED-u. V Node-RED-u objekt »msg« predstavlja sporočilo, ki se prenaša skozi tok, ali v našem primeru verigo pravil, iz enega vozlišča v drugo.

Razložimo sedaj kodo podrobneje:

`msg.Example`: Tu poskušamo dostopati do lastnosti »Example« objekta msg.
`!== undefined`: Preverjamo, ali je dostopana lastnost različna od undefined.

Izraz bo:
- Pravilen »True« če objekt »msg« ima lastnost z imenom »Example«, ne glede na njeno vrednost (razen če je ta vrednost nedefinirana).
- Napačen »False« če objekt »msg« nima lastnosti z imenom »Example« ali če je njena vrednost nedefinirana.

2.5 HTML

HTML (HyperText Markup Language) je standardni jezik za ustvarjanje in oblikovanje spletnih strani. Z HTML-jem avtorji določajo strukturo in elemente spletne strani, kot so naslovi, odstavki in povezave. Ta jezik uporablja oznake za predstavitev različnih delov vsebine ter za njihovo stilizacijo ali dodajanje funkcionalnosti. Za prilagajanje vmesnika ThingsBoard platforme je razumevanje osnovnih HTML struktur ključnega pomena.[6]

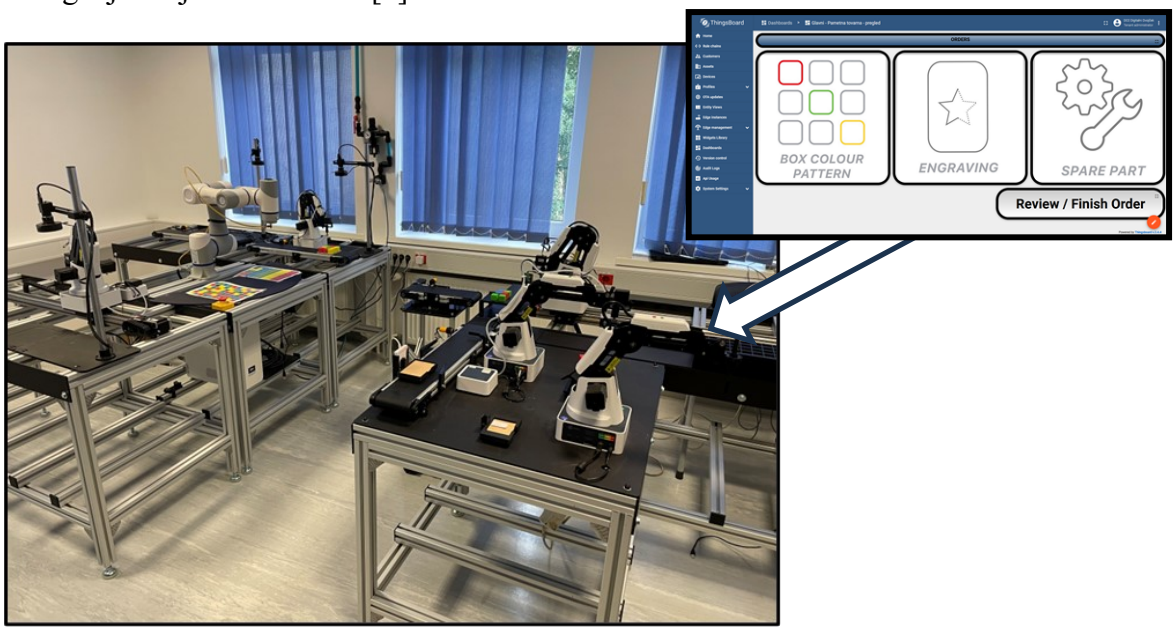
2.6 CSS

CSS (Cascading Style Sheets) je jezik, ki se uporablja za opis videza in oblikovanja dokumenta, napisanega v HTML-u. Omogoča spletnim oblikovalcem nadzor nad barvami, pisavami, razporeditvijo in na splošno estetiko spletne strani. V kombinaciji z HTML-om, CSS nudi močna orodja za oblikovanje odzivnih in vizualno privlačnih spletnih vmesnikov. Ko želimo spremeniti videz ali stil pripomočka (Ang.: *Widget*) znotraj platforme ThingsBoard, je osnovno razumevanje CSS-a nujno.[6]

3 Metodologija raziskave

V začetni fazi našega projekta smo velik del pozornosti namenili vizualni predstavitvi digitalnega vmesnika. Stremeli smo k temu, da uporabnikom ponudimo intuitiven in estetsko prijeten občutek.[7] Ko smo uspešno končali to fazo in bili zadovoljni z grafično podobo in funkcionalnostjo vmesnika, smo preusmerili svoje osredotočenje na naslednjo ključno nalogo. Posvetili smo se temu, kar se dogaja v ozadju, obdelavi in strukturiranju podatkov. Naš glavni cilj je bil, da podatke natančno pripravimo, optimiziramo in naredimo združljive z digitalnim dvojčkom Technomatix Plant Simulation. Pojasnimo razloge za to.

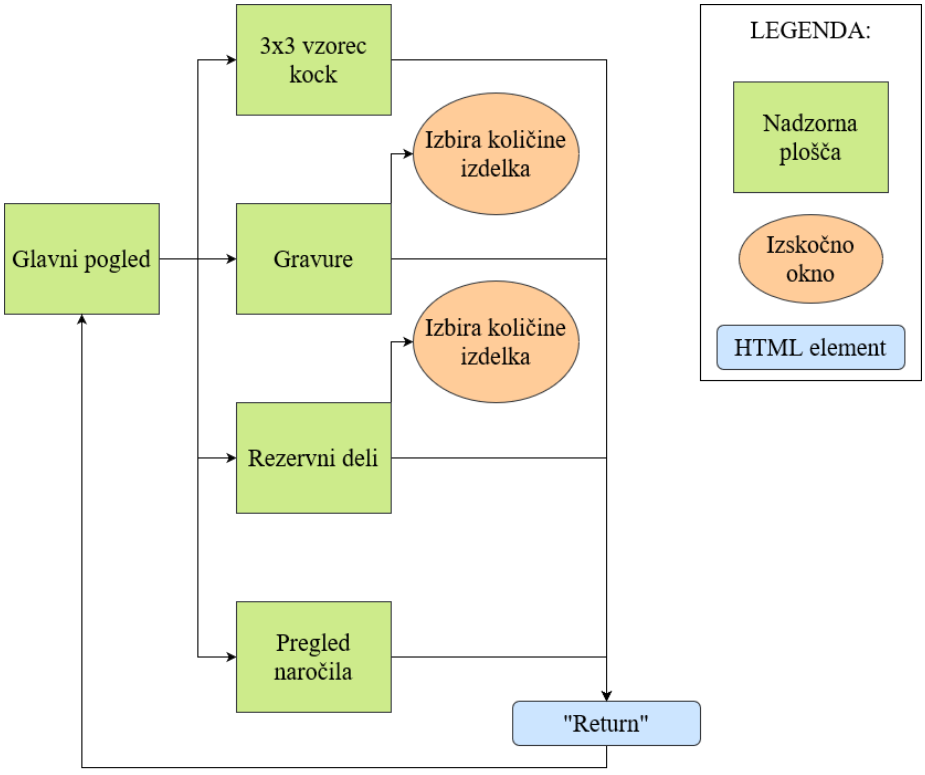
Sodobni raziskovalni trendi v izboljšanju proizvodnega procesa obsegajo nekaj področij, kot so operacijske raziskave, organizacija dobavne verige, načrtovanje proizvodnje, učinkovitost, organizacija prevoza, procesi inšpekcije in usposabljanje osebja. V logistiki se poseben poudarek daje uporabi virtualnih okolij in tehnik modeliranja in simulacije. Ta orodja nudijo ključno podporo pri reševanju izzivov v proizvodnem procesu z namenom doseganja boljših rezultatov.[8]



Slika 3.1 Demo center in digitalni vmesnik Thingsboard.

Če želimo ohraniti kvaliteto izdelka, morajo biti proizvodni sistemi prilagodljivi in trajnostni. To se doseže s »podatkovno vodenimi tovarnami«, ki dajejo prednost agilnosti, učenju in človeku usmerjeni proizvodnji. Ključna tehnologija tukaj je digitalni model, ki je strnjen prikaz obstoječega sistema, podrobno opisuje njegove funkcionalnosti in vedenje. Ta model je povezan z dejanskim sistemom in zahteva dvostransko izmenjavo podatkov. [9] Prav to je osnova in razlog za ustvarjanje našega digitalnega dvojčka in demo centra. Naš demo center je mogoče videti na sliki 3.1. Za učinkovito izvedbo naročil v predstavljenem proizvodnem sistemu je nujno zasnovati in implementirati primerni digitalni vmesnik (slika 3.1, digitalni vmesnik ThingsBoard desno zgoraj), ki omogoča naročanje naših treh specifičnih izdelkov prikazanih na sliki 3.3.

3.1 Hierarhija vmesnika



Slika 3.2 : Pregled hierarhije vmesnika.

V oblikovanju uporabniških vmesnikov ima struktura strani ključno vlogo. Dobra zasnova lahko znatno vpliva na uporabniško izkušnjo, Johnson pa navaja, da lahko dobro strukturiran vmesnik znatno izboljša produktivnost uporabnika. [7]

Ob analizi vodilnih spletnih strani, kot sta Mercedes in BMW, je jasno opaziti trend predstavitve izdelkov. Klik na določen izdelek pripelje uporabnika do podrobne strani s specifikacijami izbranega artikla. Glede na to je funkcija ustvarjanja novih stanj nadzorne plošče (Ang.: *Dashboard States*) in navigacija med njimi na platformi ThingsBoard postala ključna, saj omogoča prilagajanje uporabniških vmesnikov sodobnim standardom

oblikovanja. Integracija podobne funkcionalnosti na platformi je zahtevala osnovno znanje HTML-a, vendar je specifično zahtevala poznavanje ukaza »Actions« znotraj platforme. V tradicionalnih spletnih trgovinah se uporabniki lahko enostavno vrnejo na prejšnjo stran z uporabo standardnega gumba »nazaj« v brskalniku. V kontekstu ThingsBoard-a je takšna akcija mogoča, vendar za uporabnike vmesnika morda ni tako intuitivna. Kot rešitev tega problema je bil dodan pripomoček HTML kartica (Ang.: *HTML Card Widget*) v obliki gumba, označenega kot »RETURN« ali »Add To/Return«. Ta gumb omogoča uporabnikom enostaven prehod na prejšnjo stanje nadzorne plošče (Ang.: *Dashboard State*), kar ustvarja intuitivno in logično hierarhijo znotraj digitalnega vmesnika, kar je ključno za optimalno uporabniško izkušnjo.

Ko govorimo o spletnih trgovinah, je skoraj neizogibno imeti stran, znano kot »košarica«, kjer uporabniki lahko pregledajo vse izbrane izdelke pred dokončnim nakupom. Vendar pa v okviru platforme ThingsBoard prehod med različnimi stanji nadzorne plošče (Ang.: *Dashboard States*) ni ravno trivialen. Prav zaradi tega smo se namesto zgolj klasične izolirane strani »košarica« odločili, da izbrane izdelke vizualno predstavimo neposredno na njihovih individualnih nadzornih ploščah, kot tudi na končni, ki služi kot skupni pregled vseh izbranih izdelkov..

Na sliki 3.2 je prikazan način prehajanja med stanji nadzorne plošče (Ang.: *Dashboard States*). Prikaže se tudi uporaba izkočnega okna (Ang.: *Popover)*. Razlog za uporabo tega okna bo pojasnjen v naslednjih poglavjih.

3.2 Razdelitev v skupine

Torej, prepoznali smo pomembnost ustvarjanja vizualno privlačne in funkcionalne predstavitve naših treh izdelkov. Vzpostavitev močne povezave med uporabnikom in izdelkom se začne z ustrezno oblikovanjem in razvojem uporabniškega vmesnika. Da bi to dosegli, smo se že na začetku posvetili podrobnemu načrtovanju in strateški razdelitvi izdelkov glede na njihovo predstavitev in obdelavo atributov. S tem razmišljanjem smo naše izdelke razdelili v dve ključni skupini:

Slika 3.3 : Izdelki: a) vorec barvnih kock, b) gravure, c) rezervni del.

Prva skupina izdelkov, ki vključuje vzorec obarvanih kock (slika 3.3, a.) je predstavljala zapleteno uganko v kontekstu oblikovanja uporabniškega vmesnika. Raznolikost in številčnost atributov, ki jih je bilo treba vizualno predstaviti in organizirati, so naredili to skupino še posebej zahtevno. Zaradi te kompleksne strukture in potrebe po premišljenem razmišljanju o najboljši predstavitvi vsakega posameznega atributa, smo se odločili, da ta izdelek ločimo v svojo posebno skupino.

Po drugi strani so bili izdelki, kot so gravure in rezervni del (slika 3.3, b.,c.) čeprav so imeli svoje specifičnosti, veliko manj zapleteni za predstavitev. Za potrebo po prikazovanju in organiziranju samo dveh glavnih atributov je bil postopek oblikovanja in strukturiranja teh strani veliko bolj linearen. Čeprav govorimo o različnih izdelkih, so njihove strani, glede na samo bistvo izdelka, popolnoma enake. Zaradi teh razlogov smo jih uvrstili v drugo, enostavnejšo skupino.

3.3 Glavna stran in prehajanje med stanji nadzorne plošče

Ko smo razmišljali o oblikovanju osnovne stranice, odločili smo se za uporabo 3 pripomočka HTML kartice (Ang.: *HTML Card Widgets*), da bi jasno in preprosto predstavili naše tri možne izdelke. Card pripomočki (Ang.: *Card Widgets*) omogočajo vizualno privlačen in urejen prikaz vsakega izdelka, in z majhnimi spremembami v njihovi kodi, smo jih uspešno prilagodili da odražavajo specifičnosti naših izdelkov.

V okviru platforme ThingsBoard, za prehod med dvema stanjema nadzorne plošče (Ang.: *Dashboard States*) smo uporabili naše znanje HTML programiranja v kombinaciji z »action« možnosti znotraj same platforme. Ko izberemo »action« možnost, omogočene so nam različne aktivnosti, ki jih je mogoče izvesti s klikom na določen HTML element. Naša izbira je bila, da z akcijo omogočimo prehod na drugo stanje nadzorne plošče. Vsaka od teh akcij ima svoj edinstven identifikator, ki se uporablja znotraj HTML kode. V našem primeru smo uporabili sliko kot sprožilnik za to akcijo. Torej s klikom na sliko se uporabnik preusmeri na drugo stanje nadzorne plošče.

Ko pa gre za naš pripomoček v obliki gumba s imenom »Return«, smo uporabili podobno načelo. Tokrat, namesto slike, smo uporabili besedilni element kot sprožilnik za izvajanje akcije.

Pomembno je še omeniti da obstaja možnost, da namesto prehoda na novo stanje nadzorne odpremo izkočno okno (Ang.: *Popover)*. To dosežemo tako, da v »Dashboard state display option« izberemo »Open in popover« namesto »Open in separate dialog«.

3.4 Obravnava prve skupine izdelkov

Pri razmišljanju o najboljšem načinu prikaza izbranih atributov smo se odločili za ustvarjanje pripomočka, ki bi imel strukturo 3x3 mreže katera bi dinamično spreminjala svojo barvo v

skladu z izbranim atributom. Da bi to lahko naredili, najprej je bilo potrebno na enostaven in logičen način uporabniku predstaviti možnost izbire. Pri postopku izbire atributov smo uporabili obstoječi pripomoček za posodobitev več atributov (Ang.: *Update Multiple Attributes Widget*). Zaradi fleksibilnosti tega pripomočka, smo ga enostavno prilagodili našim potrebam, spremenili smo ga v tabelo z devetimi spustnimi meniji, vsak s tremi možnimi opcijami: Rdeča (Red), Rumena (Yellow) in Zelena (Green). V skladu s tem so bile vrednosti atributov nastavljene tako, da Rdeča ustreza »R«, Rumena »Y« in Zelena »G«. Tako smo učinkovito in intuitivno omogočili uporabnikom, da izberejo željene atribute. Vendar pa smo se soočili z izzivom - ThingsBoard ni ponujal pripomočka, ki bi prikazal atribut skozi barvo njegove vrednosti. Zato smo se lotili spremembe obstoječega pripomočka kartice atributov (Ang.: *Attributes Card Widget*). Ko smo uspešno implementirali in prilagodili sistem za izbiro in vizualizacijo atributov, je bilo treba tudi smisleno prikazati izbrane kombinacije barv in njihove količine. V ta namen smo se ponovno zanašali na naše programerske veščine, spreminjali prej omenjeni pripomoček kartice atributov, vendar z drugačnim naborom specifikacij.

Ena od ključnih funkcionalnosti pri spletnem nakupovanju je omogočanje uporabniku, da v primeru napake hitro in učinkovito spremeni svoje naročilo. Za to funkcionalnost smo se odločili, da uporabimo pripomoček za posodobitev deljenih atributov (Ang.: *Update Shared Attributes Widget*). Specifični razlogi za izbor tega pripomočka in kako je bil integriran v celoten sistem bodo podrobneje razloženi v segmentu, posvečenem ustvarjanju in delovanju zaporedja pravil kasneje v tem poglavju.

3.5 Obravnava druge skupine izdelkov

Da bi izboljšali hierarhijo in strukturo naše spletne strani, smo se tokrat odločili za implementacijo funkcije »Open in Popover« v platformi ThingsBoard. Ta funkcija uporabnikom omogoča izbiro količine izdelkov preko izkočnega okna, kar pomeni, da ni potrebno prehajati na drugo stanje nadzorne plošče. Tako optimizirana navigacija naredi uporabniško izkušnjo veliko boljšo in učinkovitejšo, zmanjšuje nepotrebne korake in olajša izbiro izdelkov.

Pri tej skupini je postopek ustvarjanja vizuelne predstavitve izdelkov in izbire njihove količine bil precej linearen in neposreden. Kot osnovo za prikaz izdelkov smo uporabili pripomoček HTML kartica (Ang.: *HTML Card Widget*) v katerega smo integrirali sliko izbranega izdelka. S klikom na to sliko izdelka bi se pa enostavno aktiviralo izkočno okno. V tem oknu smo še implementirali pripomoček za posodobitev deljenih atributov (Ang.: *Update Shared Attributes Widget*), ki uporabnikom omogoča preprosto in intuitivno določanje želene količine izdelka.

Ko smo razvijali stran za predstavitev prve skupine izdelkov, smo poudarili potrebo po pripomočku, ki bo prikazoval izbrane izdelke. V kontekstu druge skupine izdelkov se ta potreba ponovno pojavi. Glede na to, da pripomočki v platformi ThingsBoard, ki smo jih uporabljali, delujejo na principu izbranih atributov, ni bilo potrebe po ustvarjanju popolnoma novega pripomočka. Namesto tega je bil že obstoječi pripomoček prilagojen in modificiran, da podpira to dodatno funkcionalnost. Ta optimizacija je znatno pospešila razvojni proces in zagotovila skladnost v uporabniškem vmesniku.

Kot zadnja funkcionalna komponenta smo dodali gumb za brisanje naročila. Ta element je bil zasnovan in implementiran na enak način kot v prvi skupini izdelkov. S tem smo želeli zagotoviti skladnost uporabniške izkušnje in omogočiti kupcem intuitivno upravljanje svojih naročil skozi celoten sistem.

3.6 Košara

Na zaključni strani, ki predstavlja košarico, smo implementirali podobne pripomočke (Ang.: *Widgets*) kot smo jih uporabljali za vizualizacijo prvega in drugega tipa izdelka pri prejšnem podpoglavlju. Vendar je sedaj tukaj poudarek na prikazu celotne izbire uporabnikovih izdelkov. Poleg tega je uporabniku omogočeno, da vnese svoje ime in edinstveni ID naročila preko dodatnega pripomočka. Ko uporabnik določi vse podrobnosti, lahko odda naročilo s klikom na posebej zasnovan gumb »Place Order«. Ta gumb je zgrajen po podobnem načelu kot gumb za brisanje naročila v prejšnem podpoglavlju. Ko je naročilo uspešno oddano, se osredotočimo na obdelavo in formatiranje zbranih podatkov, da so pripravljeni za nadaljnjo obdelavo in pošiljanje na digitalni dvojček.

3.7 Razvoj novega zaporedja pravil

Digitalni dvojček proizvodnega procesa je izdelan v programskem orodju Tecnomatix Plant Simulation[10], zahteva, da se informacije iz ThingsBoarda do digitalnega dvojčka pošljejo v JSON formatu. Čeprav je bila večina podatkov na naši IoT platformi shranjena kot besedilo (Ang.: *String*) ali številka (Ang.: *Integer*), je bil izziv te podatke pretvoriti v želeni JSON format. Glavni razlog, zakaj smo uporabljali besedila in številke, je bila zmanjšanje potrebe po dodatnem programiranju, zahvaljujoč obstoječim pripomočkom na platformi.

Format, ki je bil potreben za »Tecnomatix Plant Simulation«, je bil takšen:

{„ID_Narocila”: 1234, „Model_1_Gravura”: 2}

V tem formatu, »{...}« predstavlja JSON objekt. Znotraj tega objekta, v tem primeru, imamo dva para ključ-vrednost. Prvi par, »ID_Narocila«: 1234, kaže, da obstaja atribut (ali ključ) z imenom »ID_Narocila«, katerega vrednost je 1234. Enako velja za drugi par »Model_1_Gravura«: 2, ki kaže na ključ »Model_1_Gravura« z vrednostjo 2. Vsak ključ predstavlja specifično informacijo, medtem ko vrednost zagotavlja podrobnosti te informacije.

Naš glavni cilj je bil pretvoriti nabor atributov, v strukturiran JSON format, pripravljen za nadaljnjo obdelavo. Da bi dosegli ta cilj, smo se zanašali na ustvarjanje nove verige pravil (Ang.: *Rule Chain*) in na prilagojene skripte, napisane v JavaScriptu. S tem pristopom smo uspešno premostili vrzel med uporabniško izbiro in tehničnimi potrebami simulacijskega orodja, s pomočjo moči in prilagodljivosti, ki nam jo ponujata JavaScript in ThingsBoard platforma.

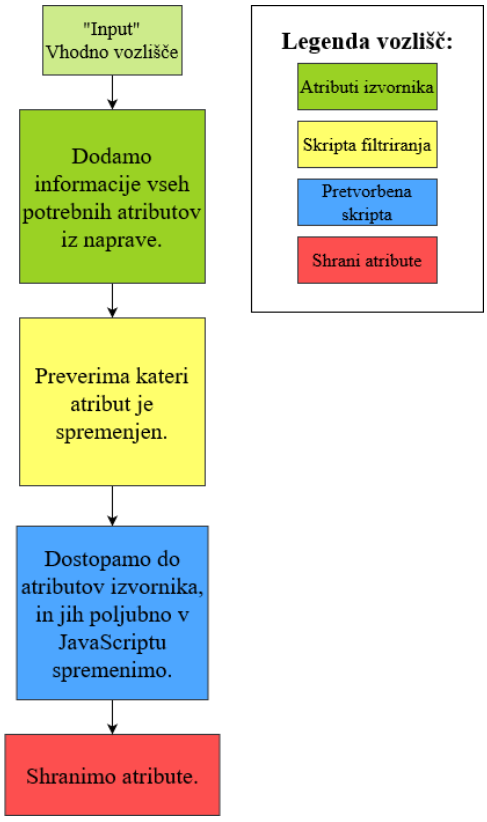
Glavne funkcije, ki jih zahtevamo od naše verige pravil in so enako pomembne, so:

- Združevanje in pravilno formatiranje vseh izbranih izdelkov in njihove količine.
- Omogočanje izbire različnih shem obarvanih kock v istem naročilu.
- Brisanje izbranih izdelkov na njihovih posameznih straneh.
- Arhiviranje naročil.

V tem poglavju smo poudarili uporabo pripomočka za posodobitev deljenih atributov (Ang.: *Update Shared Attribute Widget*) za funkcionalnost brisanja izbranih izdelkov. Ta izbira se morda na prvi pogled zdi nelogična, zato je sedaj pomembno pojasniti razloge za to odločitev.

Čeprav platforma ThingsBoard ponuja različne mehanizme za komunikacijo med verigo pravil in nadzornimi ploščami, kot so »RPC Call« ali »REST API Call«, vsak od teh mehanizmov prinaša svoje tehnične izzive in specifike. V našem primeru bi večina standardnih komunikacijskih metod zahtevala precej bolj kompleksno programiranje, kar bi povečalo možnost napak in podaljšalo čas razvoja.

Zato smo se odločili za pristop, ki temelji na posodabljanju atributov naprave (Ang.: *Device Attributes)*. Osrednja ideja tega pristopa je, da vsaka sprememba ali posodobitev atributa naprave postane sprožilec za določene operacije znotraj verige pravil. Z drugimi besedami, namesto da izrecno pošiljamo zahteve ali ukaze, uporabljamo posodobljene atribute kot signal za verigo, da izvede določene naloge. To omogoča večjo prilagodljivost pri upravljanju logike in olajša integracijo med različnimi komponentami sistema. Vodeni s tem principom pristopa in preusmerjanja podatkov pri posodabljanju atributov smo ustvarili recept, ki nam je pomagal pri razvoju ustreznega blokovnega diagrama zaporedja pravil. Na sliki 3.4 je predstavljen blokovni diagram, ki prikazuje ta proces.



Slika 3.4 : Blokovni diagram zaporedja pravil.

Zdaj, ko smo postavili osnovo za ustvarjanje naše verige pravil (Ang.: *Rule Chain*), si poglejmo, kako smo reševali naše zahtevane funkcije:

Pri ustvarjanju vmesnika za vzorec obarvanih kock smo se soočili z izzivom inherentne zasnove platforme ThingsBoard. Čeprav je uporabnik lahko enostavno definiral želeni vzorec, ga konfiguriral in izbral njegovo količino, sam ThingsBoard ni podpiral možnosti shranjevanja več vrednosti v en atribut. To je predstavljalo težavo, saj bi, če bi uporabnik želel dodati še en vzorec samo različnih barv, prejšnja konfiguracija bila avtomatsko zamenjana, namesto da bi se ji dodalo. Da bi premagali to oviro, smo uporabili naše programerske spretnosti in uporabili določena vozlišča (Ang.: *Nodes*) v verigi pravil, s katerimi smo ustvarili sistem, ki omogoča shranjevanje več različnih izbranih obrazcev skupaj z njihovimi izbranimi količinami.

Pri obdelavi in formatiranju vseh pridobljenih informacij smo pa uporabili metode podobne tistim pri prejšnjem združevanju podatkov vzorcev obarvanih kock. Osnove programiranja in podrobno poznavanje vozlišč verige pravil so nas hitro pripeljale do želenih rezultatov. Ko je bilo še treba arhivirati naročila, je pa bil postopek ustvarjanja novega atributa in shranjevanje informacij vanj opravljen na podoben način.

3.8 Sprememba Attributes Card pripomočku

Kot smo že omenili potrebo po prikazovanju izbranih izdelkov, je bilo potrebno spremeniti že obstoječi pripomoček kartice atributov (Ang.: *Attributes Card Widget*). Kodiranje in prilagajanje programske rešitve zahteva celovito razumevanje različnih programskih jezikov in knjižnic.

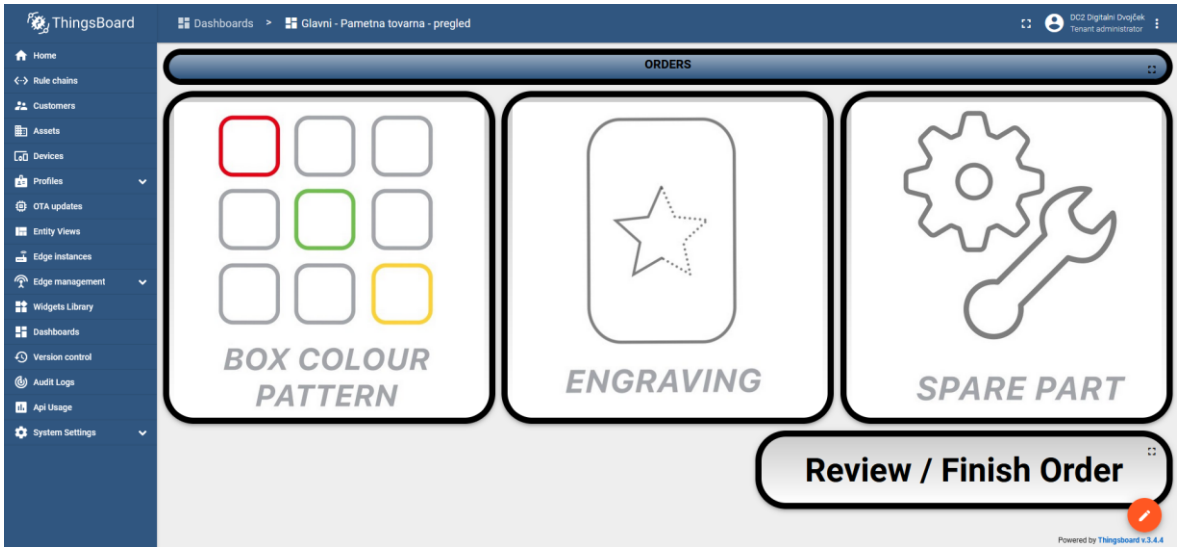
V kodi tega pripomočka (Ang.: *Widget*) se izpostavljajo štiri glavne funkcije, in sicer: onInit, onDataUpdated, onResize in onDestroy. Te funkcije igrajo ključno vlogo pri upravljanju in prikazovanju podatkov v spletnem vmesniku.

- **onInit**: Ta funkcija se pokliče ob inicializaciji spletnega vmesnika. Njena naloga je dinamično ustvariti elemente vmesnika na osnovi dostopnih virov podatkov (v našem primeru so to atributi). Torej, za vsak vir podatkov, se ustvari kontejner z imenom vira podatkov, kot tudi tabela, ki bo prikazovala ključne vrednosti iz tega vira. Poznavanje osnov jQuery-ja in manipulacija z DOM-om so nujni za spremembo kode te funkcije.
- **onDataUpdated**: Ta funkcija se pokliče vsakič, ko se podatki ažurirajo. Gre skozi vse celice, ki vsebujejo vrednosti, in jih ažurira z novimi podatki. Prav tako v funkciji obstaja logika za oblikovanje vrednosti, odvisno od tega, ali je vrednost številčna ali ne. Da bi lahko spremenili to funkcijo, bi morali razumeti koncepte upravljanja podatkov, kot so polja in objekti.
- **onResize**: Ta funkcija se pokliče vsakič, ko se spremeni velikost spletnega vmesnika. Njen cilj je dinamično prilagoditi velikost pisave naslova vira podatkov in celic z podatki v skladu z trenutno velikostjo vmesnika. Da bi spremenili to funkcijo, osnovno je razumevanje CSS-a in jQuery-ja nujno.
- **onDestroy**: Čeprav ta funkcija trenutno ne vsebuje nobene kode, je predvidena za uporabo, ko je potrebno opraviti čiščenje ali dealokacijo virov, preden se komponenta uniči.

4 Rezultati

Rezultati so predstavljeni v skladu z vrstnim redom reševanja naloge prikazanem v metodologiji. Začnemo z vmesnikom, preko prikaza posameznih stanj nadzornih plošč, do opisa njihovih funkcionalnosti.

4.1 Vmesnik



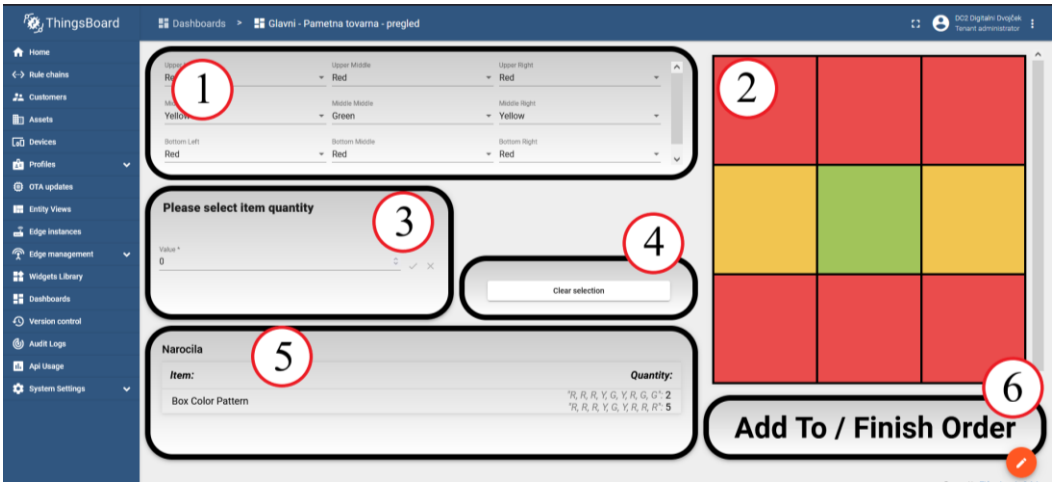
Slika 4.1 : Glavni pogled vmesnika.

Najprej bomo predstavili začetno točko uporabniške izkušnje znotraj naše platforme ThingsBoard – osnovno nadzorno ploščo (Ang.: *Dashboard*). Na sliki 4.1 prevladujejo tri poudarjene HTML kartice (Ang.: *HTML Card Widget*), ki služijo kot primarni vmesniški elementi za uporabnike.

Poleg HTML kartic je poseben poudarek dan vizualni predstavitvi izdelkov. V skladu s sodobnimi trendi v oblikovanju uporabniške izkušnje je bil izbran minimalističen prikaz izdelkov. Takšen pristop ne le omogoča uporabnikom hitro orientacijo, ampak tudi ustvarja čisto in elegantno vizualno estetiko, ki postavlja izdelke v ospredje pozornosti. Podrobnejša predstavitev grafikonov teh izdelkov je prikazana na sliki 4.2.

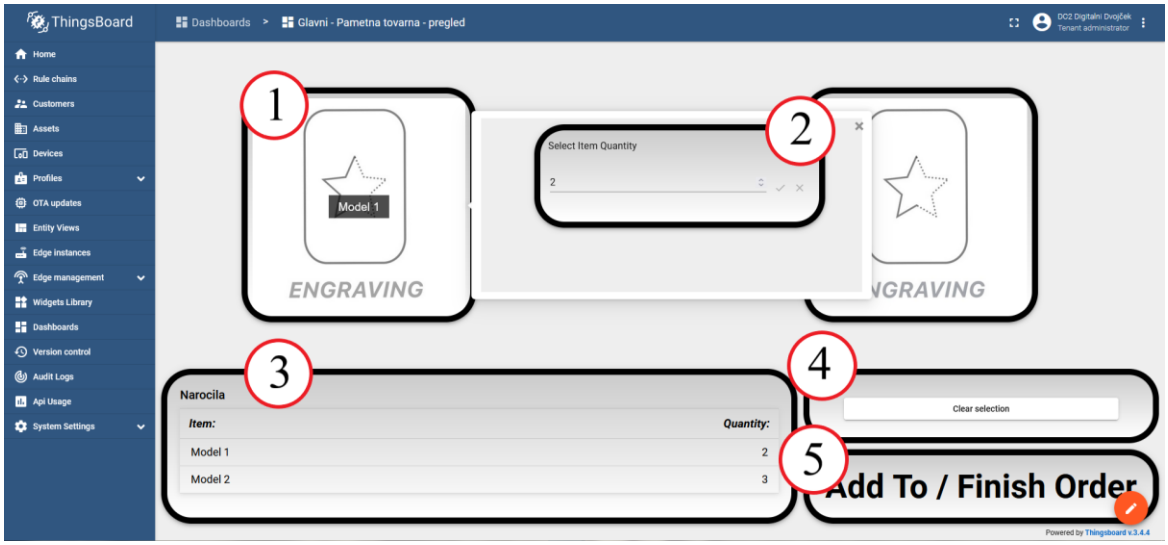
Slika 4.2 : Grafične predstavitve izdelkov.

Na sliki 4.3 lahko vidimo stanje nadzorne plošče (Ang.: *Dashboard State*), ki je namenjen izbiri vzorcev obarvanih kock. Osrednje mesto zavzema prilagojeni pripomoček za posodobitev več atributov (Ang.: *Update Multiple Attributes Widget*) (Slika 4.3 št.1), ki uporabnikom omogoča prilagoditev njihovega izbora. Poleg tega je prisoten tudi spremenjeni pripomoček kartice atributa (Ang.: *Attributes Card Widget*) (Slika 4.3 št. 2), ki vizualno predstavlja uporabnikov izbor vzorcev. V spodnjem desnem kotu je jasno viden pripomoček HTML kartica (Ang.: *HTML Card Widget*) (Slika 4.3 št. 6), ki uporabnikom omogoča vrnitev na prejšnjo stanje nadzorne plošče. Opazimo lahko tudi pripomoček za posodobitev deljenih atributov (Ang.: *Update Shared Attributes Widget*) v obliki gumba »Clear Selection« (Slika 4.3 št. 4), ki v tem primeru služi kot mehanizem za odstranjevanje že izbranih izdelkov. Dodatno, drugi prilagojeni pripomoček kartice atributov (Ang.: *Attributes Card Widget*)(Slika 4.3 št. 5) prikazuje vse izbrane izdelke skupaj z njihovimi količinami, kar uporabnikom nudi jasen pregled njihove izbire.

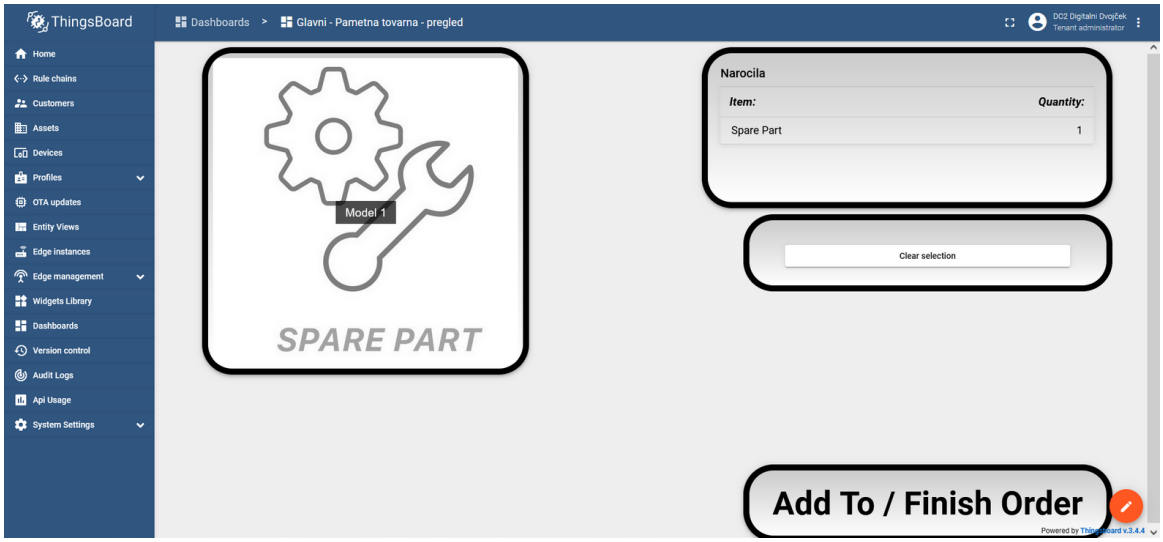


Slika 4.3 : Vmesnik za izbiro vzorca barvanih kock.

Na sliki 4.4 je prikazano stanje nadzorne plošče (Ang.: *Dashboard State*), namenjeno izboru gravur. Opazna je uporaba funkcije izkočnega okna (Ang.: *Popover*) ki uporabnikom omogoča izbiro količine gravur (Slika 4.4 št. 2). Posebej izstopata 2 HTML kartici (Ang.: *HTML Card Widget*) (Slika 4.4 št.1), ki ob prehodu miške čez njiju (Ang.: *Hover*) razkrijeta ime izbranega modela gravure. Poleg tega se na tej strani nahaja tudi prilagojeni pripomoček kartice atributov (Ang.: *Attributes Card Widget*)(Slika 4.4 št. 3), ki vizualno prikazuje uporabnikove izbire modelov gravur. Prav tako je prisoten pripomoček HTML kartica (Ang.: *HTML Card Widget*) »Add To/Finish Order« (Slika 4.4 št. 5), ki omogoča vrnitev na prejšnjo stran, ter spremenjen pripomoček posodobitve atributa naprave (Ang.: *Update Device Attribute Widget*) v obliki gumba na katerem je napisano »Clear Selection« (Slika 4.4 št.4), ki služi za odstranitev izbranih izdelkov iz uporabnikove izbire.



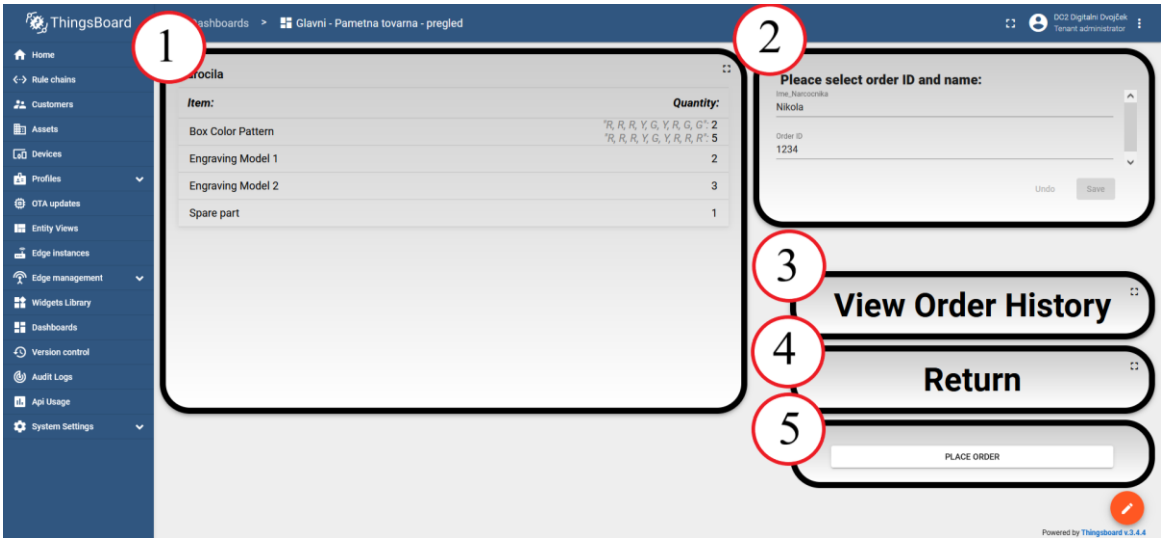
Slika 4.4 : Vmesnik za izbiro gravur.



Slika 4.5 : Vmesnik za izbiro rezervnih delih.

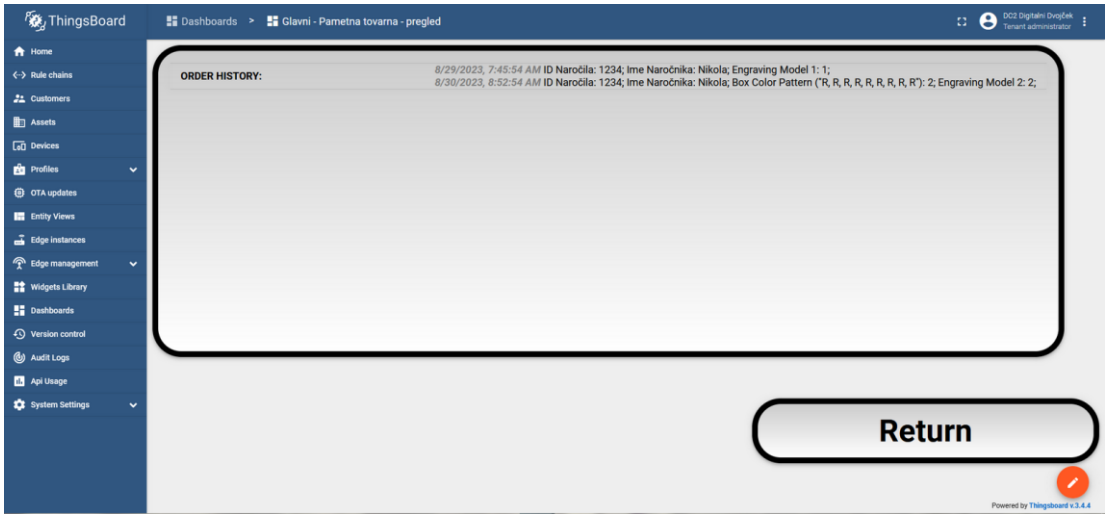
Na sliki 4.5 je prikazanao stanje nadzorne plošče (Ang.: *Dashboard State*) namenjeno izbiri rezervnih delov. Opažamo, da je oblikovana na podoben način kot stran za izbor gravur. Ta doslednost v oblikovanju omogoča uporabnikom, da se lažje orientirajo in razumejo vmesnik, ne glede na to, kateri izdelek izbirajo. Takšna doslednost v oblikovanju prav tako poudarja našo metodološko odločitev glede združevanja teh izdelkov, kar dodatno potrjuje, da je bil tak pristop logičen in usmerjen k uporabniku.

Na sliki 4.6 je prikazana zaključna stran, imenovana »Košarica«. Uporabljamo že prej omenjeni prilagojeni pripomoček kartice atributov (Ang.: *Attributes Card Widget*) (Slika 4.6 št. 1), samo so sedaj zbrani vsi predhodno izbrani izdelki. Dodatno je vključen tudi pripomoček za posodobitev več atributov (Ang.: *Update Multiple Attributes Widget*), prek katerega uporabnik vpiše identifikacijsko številko naročila in ime naročnika (Slika 4.6 št. 2). V spodnjem desnem kotu strani je gumb za potrditev naročila (Slika 4.6 št. 5), ki predstavlja pripomoček za posodobitev atributa naprave (Ang.: *Update Device Attribute Widget*). S pritiskom na ta gumb uporabnik posreduje vse informacije povezane z naročilom v nadaljnjo obdelavo. Ob tem gumbu je tudi pripomoček HTML kartica (Ang.: *HTML Card Widget*), ki deluje kot povratni gumb na začetno stran (Slika 4.6 št. 4) ter gumb za prehod na stanje nadzorne plošče, kjer je prikaz zgodovine naročil (Slika 4.6 št. 3), zagotavljajoč uporabnikom intuitivno in enostavno navigacijo skozi postopek nakupa.



Slika 4.6 : Vmesnik za prikaz naročila in finalizacija.

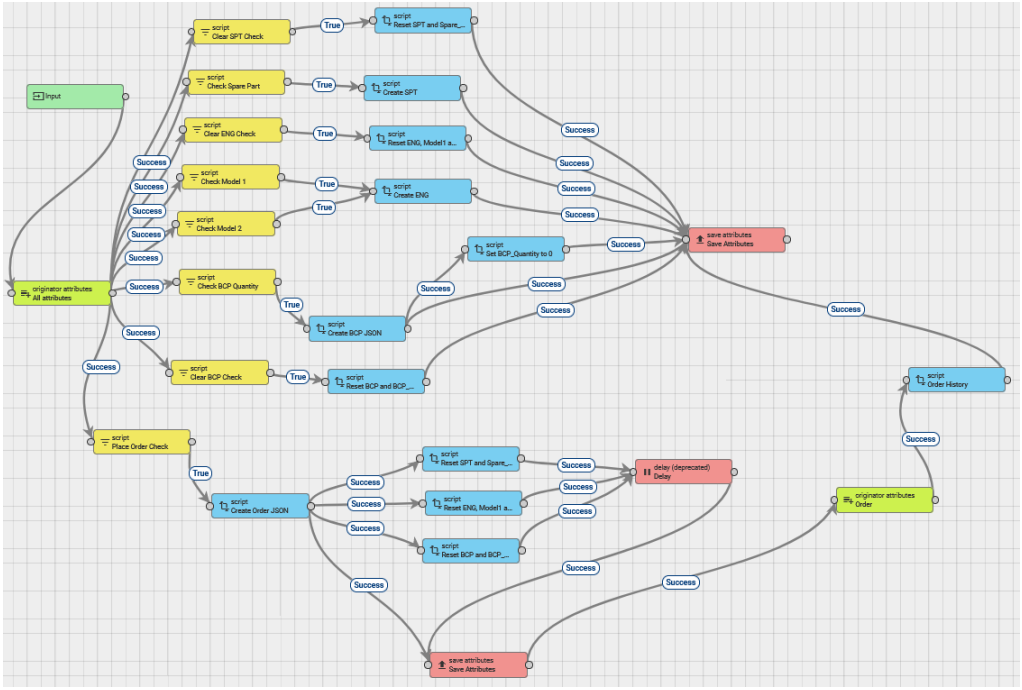
Na koncu na sliki 4.7 prikažemo stanje nadzorne plošče (Ang.: *Dashboard State*) za prikaz arhive naročil. Tukaj lahko v samem centru nadzorne plošče opazimo naš spremenjeni pripomoček kartice atributov (Ang.: *Attributes Card Widget*) in v spodnjem desnem kotu pripomoček HTML kartice (Ang.: *HTML Card Widget*) za vrnitev na prejšno stran.



Slika 4.7 : Prikaz arhive naročil.

4.2 Veriga pravil

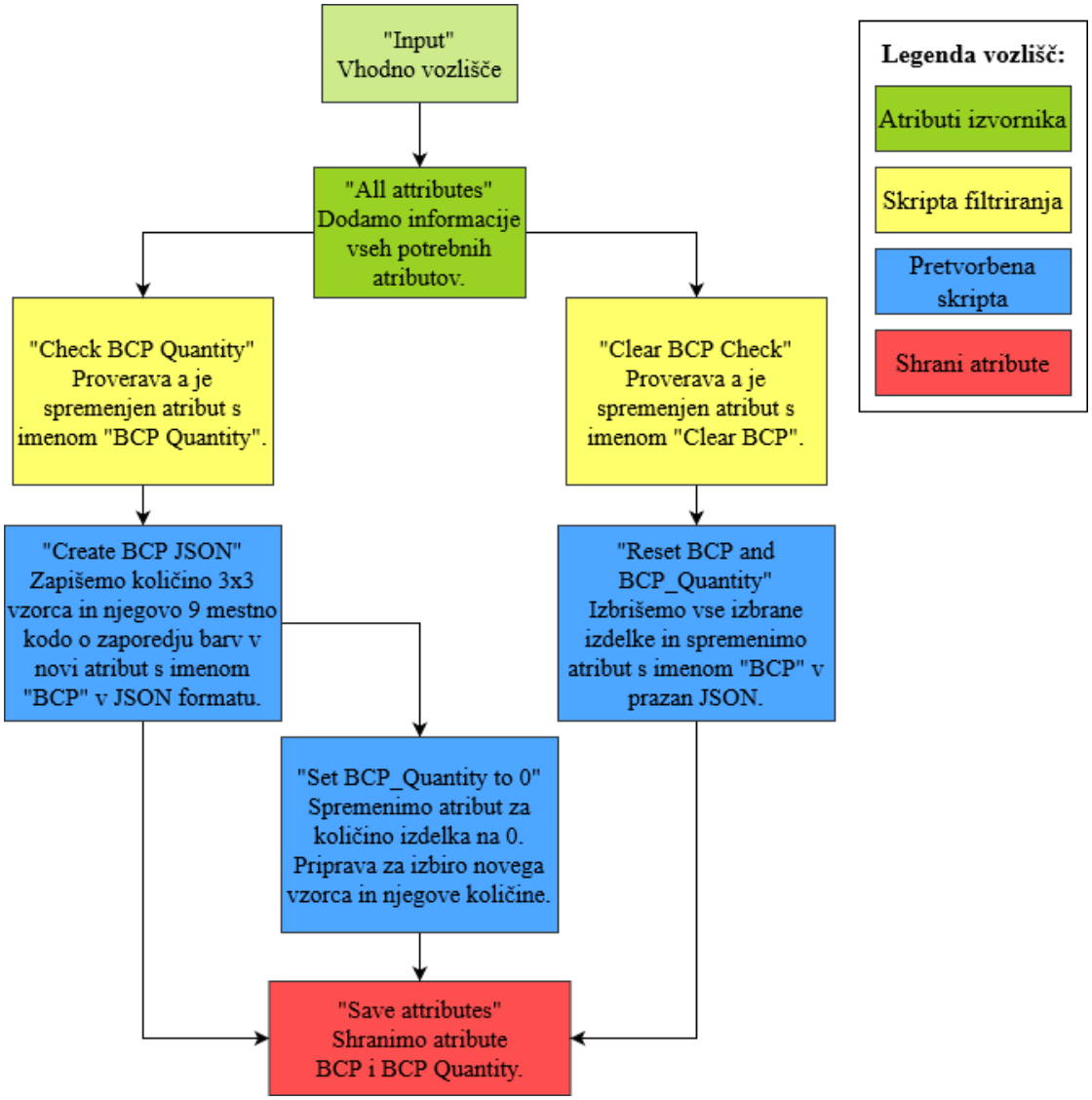
Na sliki 4.8 lahko vidimo grafično predstavitev našega zaporedja pravil (Ang.: *Rule Chain*) v ThingsBoard-u. Bistvo prikaza te slike ni v detajlih posameznih vozlišč (Ang.: *Node*) temveč prikaz različnih vrst vozlišč glede na njihove barve in njihovo povezanost, vrstni red pravil. Na prvi pogled ta veriga pravil morda deluje kompleksno, vendar lahko njegovo funkcijo strnemo in predstavimo s pomočjo blokovnih diagramov, kar olajša boljše razumevanje celotnega procesa.



Slika 4.8 : Prikaz končnega zaporedja pravil v ThingsBoardu.

Naša veriga pravil ima očitno razvejano strukturo, kjer vsaka veja predstavlja določeno funkcionalnost ali tok podatkov. V nadaljevanju bomo podrobneje obravnavali vsako pomembno vejo tega zaporedja pravil in pojasnili njihove specifične vloge in funkcionalnosti.

V naši verigi pravil vsaka veja sledi jasnemu in doslednemu toku obdelave. Najprej se uporablja vozlišče za filtriranje (Ang.: *Filter Script Node*) , da se ugotovi, ali je bil določen atribut posodobljen. Če se potrdi, da je bilo posodobljeno, se informacije prenesejo naslednjemu vozlišču, kjer se izvede transformacija podatkov. Po transformaciji se vsi atributi shranijo. Glede na to, da smo predstavili tri izdelke, je logično, da imamo v verigi pravil tri ustrezne veje, vsaka namenjena za obdelavo informacij, povezanih z vsakim posameznim izdelkom. Kot tudi eno vejo za njihovo združevanje in nadaljnjo obdelavo in formatiranje.

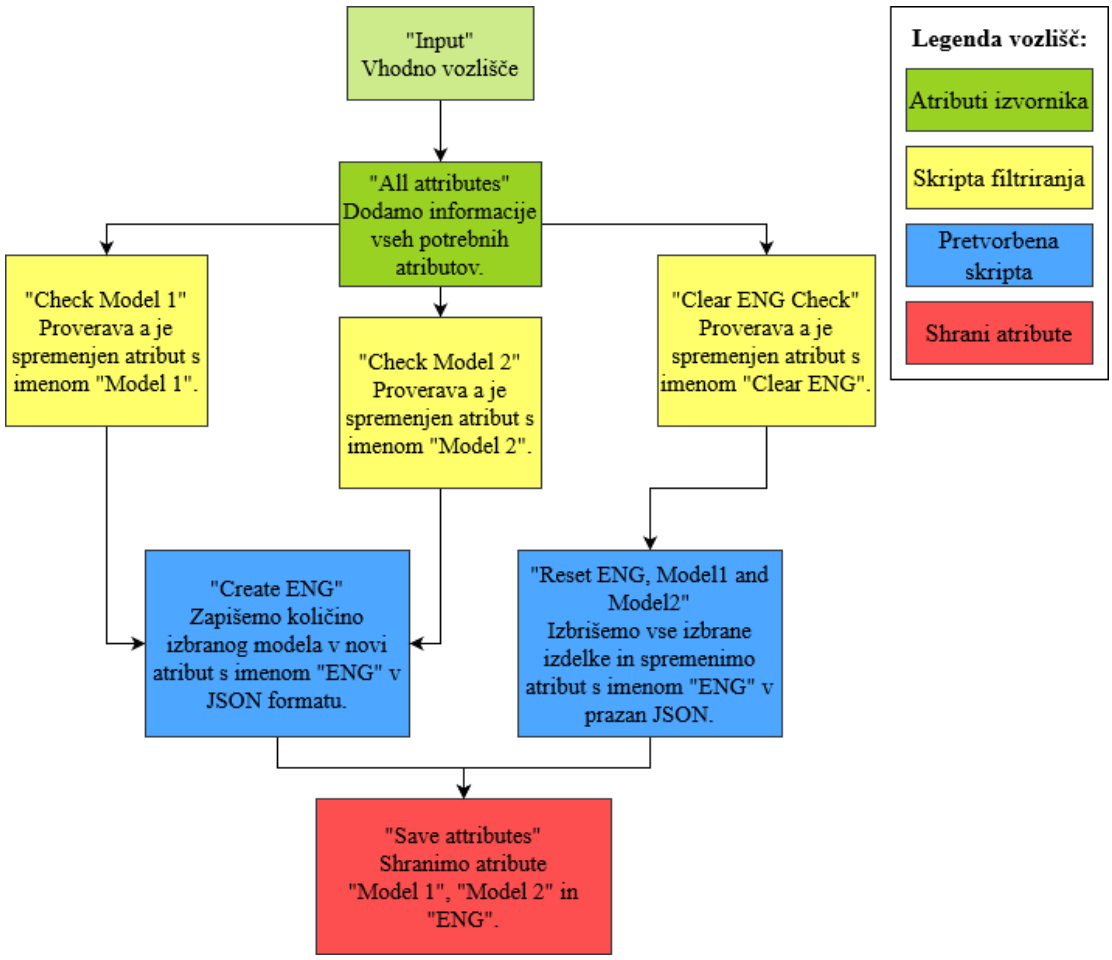


Slika 4.9 : Potek procesiranja informacij vzorca barvanih kock v verigi pravil.

Na sliki 4.9 je grafično predstavljen blokovni diagram postopka kreiranja in shranjevanja vzorca obarvanih kock. V okviru tega procesa smo združili deset atributov v enoten in strukturiran format. Za vsak izdelek smo uporabili pristop shranjevanja atributov v JSON formatu, saj je bila takšna metodologija postavljena kot zahteva naloge

Med implementacijo tiste verige pravil smo se soočili z izzivom pri izbiri večih vzorcev obarvanih kock. Ob tem koraku je bilo esencialno prilagoditi atribut za izbor količine na vrednost 0. V primeru, da ta prilagoditev ni bila izvedena, bi uporabniku onemogočili izbor vzorca z drugačno konfiguracijo ob enaki količini kot pri prejšnjem izboru.

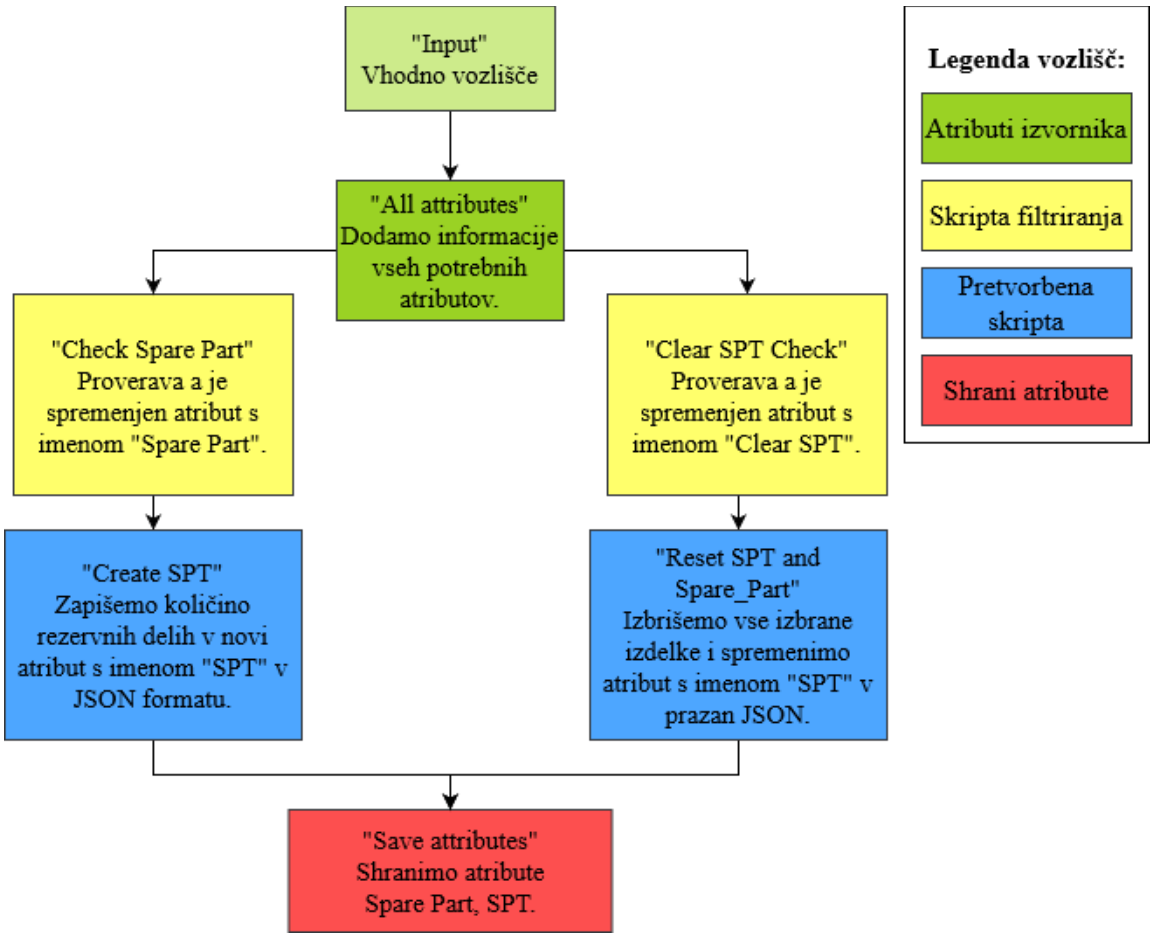
Na sliki 4.10 je prikazana veja za shranjevanje podatkov izbranih gravur. Med oblikovanjem tega zaporedja pravil nismo naleteli na večje izzive. S stališča kompleksnosti in strukture programiranja je bilo to morda celo preprostejše v primerjavi s primerom na sliki 4.9. Trenutni sistem je zasnovan tako, da sprejme dva modela, vendar je na osnovi, ki smo jo postavili, enostavno dodati še več modelov. V procesu oblikovanja zaporedja pravil za gravure, ki je predstavljen na sliki 4.11, je bilo programiranje in struktura še bolj enostavna.



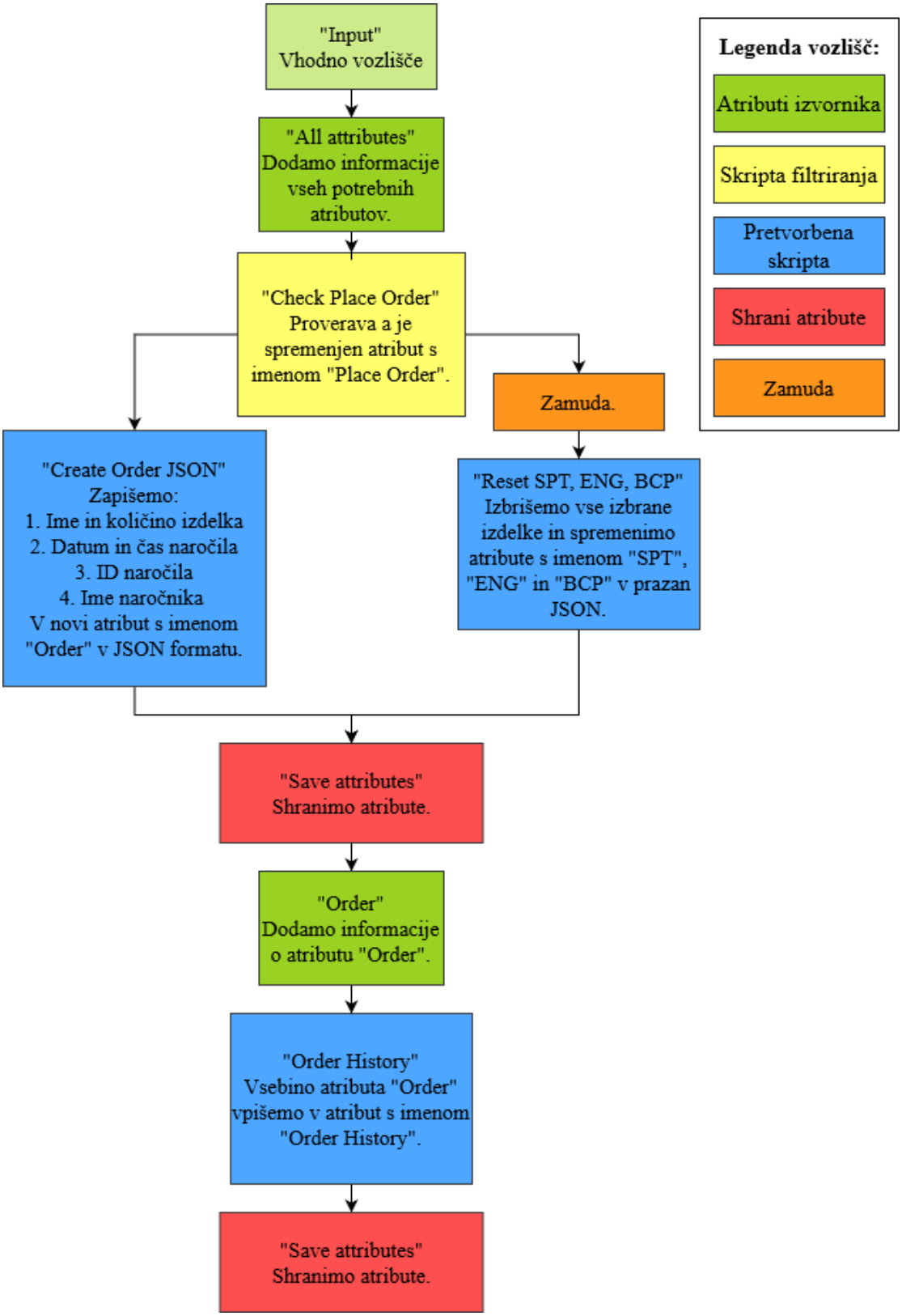
Slika 4.10 : Potek procesiranja informacij gravur v verigi pravil.

Ob koncu smo prispeli do postopka v verigi pravil, kjer je bilo potrebno združiti vse te posamezne JSON-e v en sam JSON, pripravljen za nadaljnje pošiljanje na digitalni dvojček. V veji prikazani na sliki 4.12 smo se soočali s številnimi izzivi in problemi. Po tipičnem principu, kot smo to storili v prejšnjih vejah, smo programirali preoblikovalni skript, da bi združili vse posamezne izdelke v en sam JSON. Dodali smo tudi informacije, kot so ID naročila, čas in datum ter ime naročnika. Pri shranjevanju tega JSON-a v njegov atribut je bilo smiselno vse spremenjene vrednosti ponastaviti nazaj na 0, s čimer smo omogočili naslednjemu uporabniku enako uporabo kot prejšnjim.

Med testiranjem smo ugotovili, da so se shranjevanje in resetiranje informacij pogosto nasprotovala, zaradi njihovega sočasnega pošiljanja in shranjevanja. V nekaterih primerih je prišlo do resetiranja in nato združevanja, kar je povzročilo prazen JSON. To težavo smo rešili z uporabo vozlišča za ostvaritev zamude (Ang.: *Delay Node*). Po rešitvi tega problema je bilo potrebno ustvariti še sistem za arhiviranje naročil. Zdaj smo te združene informacije morali dodati v nov JSON, ki se bo vsakič le nadgradil. To je bilo precej preprosto, vendar ob upoštevanju, kako je strukturirana koda pripomočka, ki ga uporabljamo za prikaz teh JSON-ov, ki jo bomo obravnavali v naslednjem podpoglavju, je bilo potrebno razviti logiko o spreminjanju sintakse arhiva naročil. S tem smo uspešno postavili temelj za nadaljnje delo in opravili glavno nalogo - združevanje podatkov in pripravo za pošiljanje na digitalni dvojček.



Slika 4.11 : Potek procesiranja informacij rezervnih delih v verigi pravil.

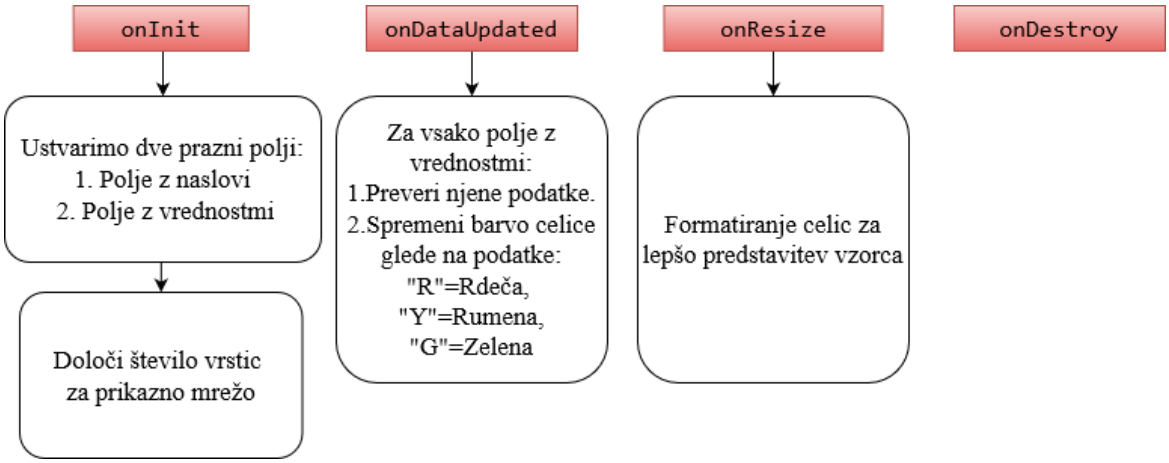


Slika 4.12 : Potek zbiranja, formatiranja in arhiviranja naročil.

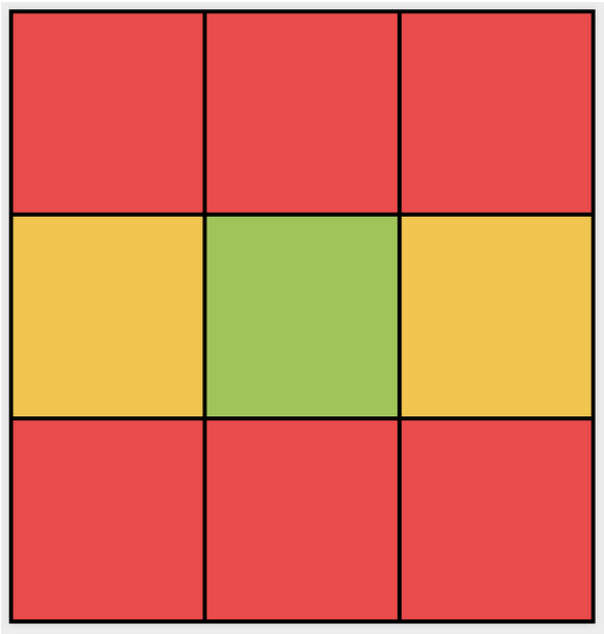
4.3 Sprememba Attributes Card pripomočka

V prejšnjem poglavju smo predstavili, kaj vsaka izmed 4 glavnih funkcij predstavlja pri programiranju naših pripomočkov. Tukaj bomo po blokih predstavili funkcionalnosti, ki smo jih dodali k že obstoječim funkcijam. Največja težava pri tem nam je bila razumevanje osnovne kode kako bi je lahko modifikovali. JQuery, JavaScript in CSS so nam bili od največjega pomena. Klicali smo se na fukncije uporabljanjem JQuery-a, programirali željene funkcije v JavaScript-u in spreminjali izgled celica s uporabo CSS-a.

Na sliki 4.13 je prikazan dodatek k funkciji za prikaz obrazca obarvanih kock. Znotraj same kode je bilo potrebno najprej narediti 3x3 mrežo. Potem ko je mreža narejena je bilo potrebno glede na informacije spremeniti barve v mreži. Na koncu smo spremenili CSS kako večjali preglednost našega pripomočka kot je vidno na sliki 4.14.



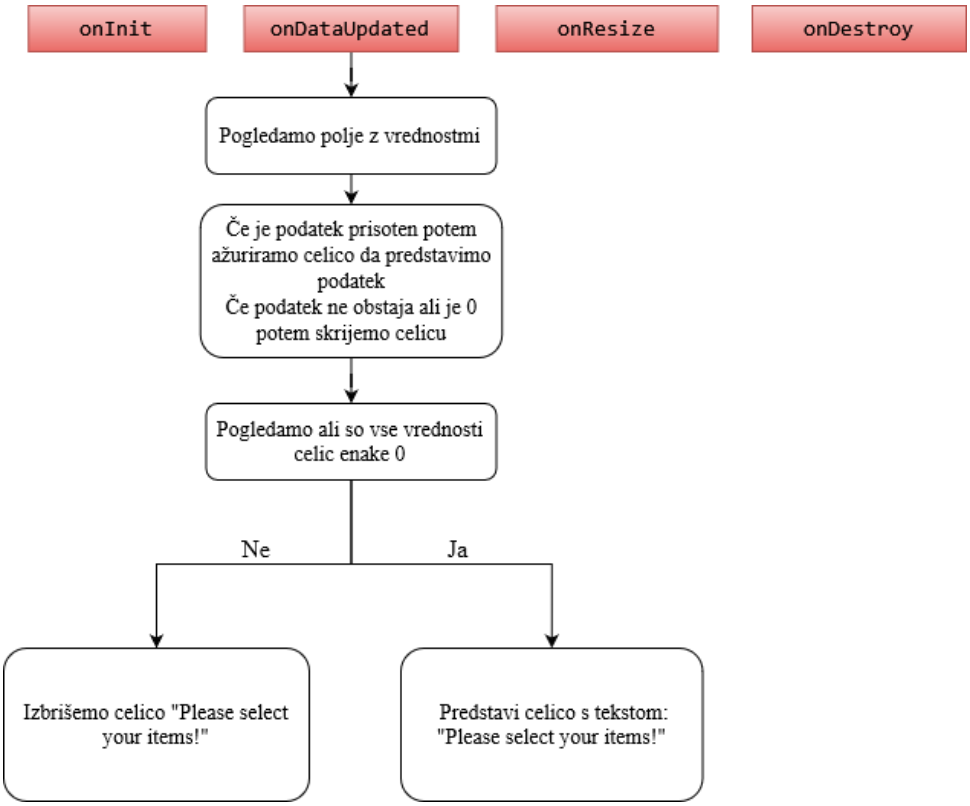
Slika 4.13 : Dodatek kodi pri prikazu vzorca obarvanih kock.



Slika 4.14 : Spremenjen pripomoček za prikaz vzorca.

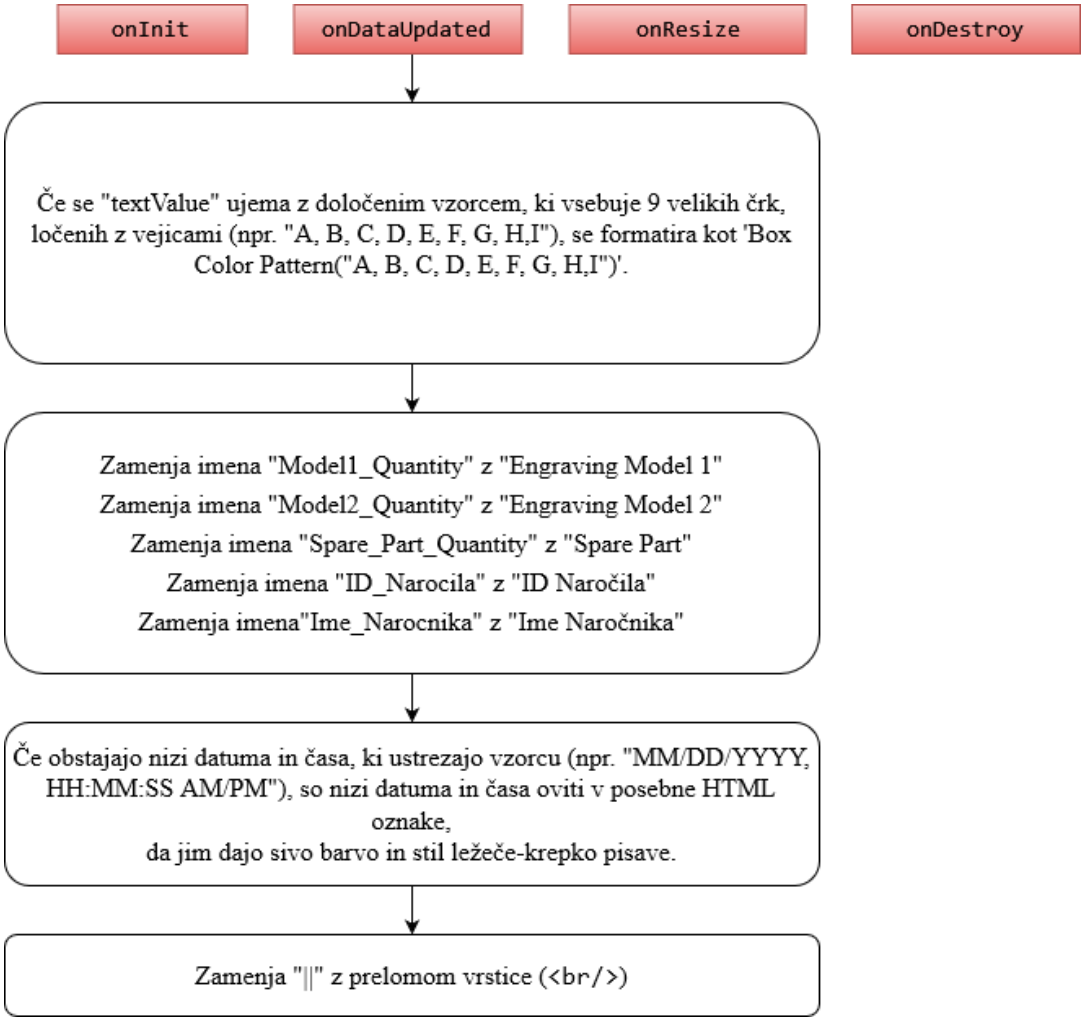
Po izdelavi pripomočka namenjenega predstavljanju našega vzorca, smo prepoznali potrebo za kreacijo dodatnega pripomočka, ki bi služil predstavljanju naših izbranih izdelkov. Pri njegovi izdelavi smo uporabili isto logiko kot v prejšnem primeru. Posebno pozornost smo posvetili estetiki, in sicer kako bi pripomoček izgledal privlačno tudi v primeru, ko uporabnik ni izbral nobenega izdelka. Čeprav je bil prvotni pripomoček zasnovan za prikaz vzorca obarvanih kock, smo želeli, da je uporaben tudi za druge izdelke, zato smo ga ustrezno prilagodili.

Dodali smo tudi funkcionalnost, ki v primeru neizbranih izdelkov prikaže opomnik »Please select items«. Zanimivo je, da s tem pripomočkom ne dostopamo neposredno do atributov naprave (Ang.: *Device Attributes)*, ampak samo do izbranih atributov v pripomočku. To smo implementirali s preverjanjem, ali so vse vrednosti atributov enake 0. Dodatki kodi so prikazani na sliki 4.15, ter predstavidev pripomočka je prikazana na sliki 4.16



Slika 4.15 : Dodatek kodi pri prikazovanju izbranih naročil.

Slika 4.16 : Spremenjen pripomoček za prikaz izbranih izdelkov.

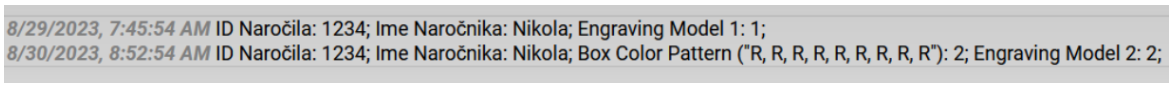


Slika 4.17: Dodatek kodi pri arhiviranju naročil.

Na, koncu smo se lotili še pripomočka za prikaz arhiviranja naročil kot je prikazano na sliki 4.17. V tem segmentu smo v verigi pravil potrebovali nekoliko bolj specifičnih znanj o tem, kako želimo formatirati naše naročilo za njegovo lažjo obdelavo. Razvoj tega pripomočka je potekal na podoben način kot v primeru prikaza izdelkov.

Kljub temu da je bila zamenjava imen atributov precej preprosta, smo ob določenih preimenovanjih naleteli na težave (pri poimenovanju barv vzorca), kar je zahtevalo uporabo regularnih izrazov ali »regex«.

Regularni izrazi (v JavaScriptu označeni kot »regex«) so močno orodje, ki omogočajo iskanje, prepoznavanje in manipulacijo specifičnih nizov znotraj besedila. Regex se pogosto uporablja za ujemanje vzorcev znotraj nizov. Primer predstavitve arhive naročil je viden na sliki 4.18.



Slika 4.18 : Arhiva naročil primer.

5 Zaključki

Rezultati naše naloge so obsegali izgradnjo digitalnega vmesnika in integrirane verige pravil. Digitalni vmesnik je zasnovan za elegantno in jasno predstavitev naših treh ključnih izdelkov, medtem ko veriga pravil predstavlja napredno orodje za obdelavo in analizo podatkov, pripravljeno za integracijo z našim digitalnim dvojčkom. Na podlagi našega dela lahko delo povzamemo v naslednje korake:

1) Identificirali smo izdelke, za katere je bila potrebna izdelava digitalnega vmesnika. Na podlagi značilnosti izdelka in načina shranjevanja ter formatiranja njihovih informacij smo izdelke razdelili v dve skupini.
2) Oblikovali smo sedaj digitalni vmesnik na platformi ThingsBoard IoT za naše dve skupini izdelkov.
3) Z digitalnim vmesnikom smo predstavili posamezne izdelke ter omogočili njihovo selekcijo preko pripomočkov na omenjeni platformi. Za vsak izdelek smo oblikovali specifično stran in smo naredili košarico za njihov izbor.
4) Programsko kodo znotraj pripomočkov kartice atributov smo prilagodili specifičnim potrebam našega projekta z namenom izboljšanja uporabniške izkušnje.
5) Nadaljnje delo je obsegalo razvoj novega zaporedja pravil za zbiranje in ustrezno preoblikovanje podatkov izdelkov za prenos na digitalni dvojček. V verigi pravil smo integrirali tudi logične funkcije za optimizacijo predstavitve in selekcije izdelkov znotraj vmesnika.
6) V verigi pravil smo na koncu naredili sistem arhiviranja vseh narocil. Ta funkcija je bila pomembna pri pošiljanju podatkov na digitalni dvojček. Vsako naročilo je bilo narejeno v JSON formatu in je vsebovalo: ID naročila, ime naročnika, datum in čas naročenja, imena in količine izdelkov.

Predlogi za nadaljnje delo

Zaradi narave platforme ThingsBoard lahko enostavno gradimo na že ustvarjenem digitalnem vmesniku in verigi pravil. S tem smo ustvarili le osnovo za nadaljnje dodajanje izdelkov in razširitev proizvodnega procesa.

Literatura

[1] Greengard, Samuel: *The Internet of Things*. The MIT Press, London 2021. str. 14.

[2] *ThingsBoard*. Dostopno na: https://thingsboard.io/docs/, ogled: 30.8.2023.

[3] *Node-Red*. Dostopno na: https://nodered.org/docs/, ogled: 30.8.2023.

[4] Haverbeke, Marijn*: Eloquent JavaScript, 3rd Edition: A Modern Introduction to Programming*. No Starch Press, San Francisco, 2018.

[5] *JQuery*. Dostopno na: https://jquery.com/, ogled: 30.8.2023.

[6] Duckett, Jon: *HTML and CSS: Design and Build Websites*. John Wiley & Sons, Nashville, 2011.

[7] Johnson, Jeff: *Designing with the Mind in Mind Simple Guide to Understanding User Interface Design Guidelines*. Morgan Kaufmann, Cambridge, 2021.

[8] Debevec, M.; Simic M.; & Herakovic N.; *Virtual factory as a useful tool for improving production processes*, 2021. str. 1-2.

[9] Resman, M.; Jernej P.; Simic M.; Jovanovic V.; & Herakovic N.; *A Five-Step Approach to Planning Data-Driven Digital Twins for Discrete Manufacturing Systems*, 2020.

[10] Vuzem, Filip Jure: Razvoj Agilnega Digitalnega Dvojcka Modularno Grajenega Proizvodnega Sistema: *magistrsko delo*. Ljubljana, 2023.

Priloga

Programska koda za filter skripte, rumena vozlišča:

Vozlišče »Check BCP Quantity«, koda:

```
return msg.BCP_Quantity !== undefined;
```

Vozlišče »Check Clear BCP«, koda:

```
return msg.Clear_BCP !== undefined;
```

Vozlišče »Check Model 1«, koda:

```
return msg.Model1_Quantity !== undefined;
```

Vozlišče »Check Model 2«, koda:

```
return msg.Model2_Quantity !== undefined;
```

Vozlišče »Clear ENG Check«, koda:

```
return msg.Clear_ENG !== undefined;
```

Vozlišče »Check Spare Part«, koda:

```
return msg.Spare_Part_Quantity !== undefined;
```

Vozlišče »Clear SPT Check«, koda:

```
return msg.Clear_SPT !== undefined;
```

Vozlišče »Place Order Check«, koda:

```
return msg.Place_Order !== undefined;
```

Programska koda za skripte transformacij, modra vozlišča:

Vozlišče »Create ENG JSON«, koda:

```
// Fetch the required attributes from metadata with
metadata.shared_x
// The parseFloat() function is used to accept a string and convert
it into a floating-point number
let model1Quantity = parseFloat(metadata.shared_Model1_Quantity);
// Parse „Model1_Quantity“ as a float (number)
let model2Quantity = parseFloat(metadata.shared_Model2_Quantity);
// Parse „Model2_Quantity“ as a float (number)
// Construct the JSON for the ENG attribute
let eng = {
    „Model1_Quantity“: model1Quantity,
    „Model2_Quantity“: model2Quantity
};
// Send the constructed ENG as a new shared attribute directly as a
JSON object
msg.ENG = eng;
return { msg: msg, msgType: „POST_ATTRIBUTES_REQUEST“ };
```

Vozlišče »Reset ENG, Model1 and Model 2«, koda:

```
// Set ENG to an empty JSON object
msg.ENG = {};
// Set Model1 and Model2 to 0
msg.Model1_Quantity = 0;
msg.Model2_Quantity = 0;

return { msg: msg, msgType: „POST_ATTRIBUTES_REQUEST“ };
```

Vozlišče »Create SPT«, koda:

```
// Fetch the required attributes from metadata
let sparePart= parseFloat(metadata.shared_Spare_Part_Quantity);
// Construct the JSON for the ENG attribute
let spt = {
    "Spare_Part_Quantity": sparePart
};
msg.SPT = spt;
return { msg: msg, msgType: "POST_ATTRIBUTES_REQUEST" };
```

Vozlišče »Reset SPT and Spare_Part_Quantity«, koda:

```
msg.SPT = {};
msg.Spare_Part_Quantity = 0;
return { msg: msg, msgType: "POST_ATTRIBUTES_REQUEST" };
```

Vozlišče »Create BCP JSON«, koda:

```
// Fetch the required attributes from metadata
let u_l = metadata.shared_U_L;
let u_m = metadata.shared_U_M;
let u_r = metadata.shared_U_R;
let m_l = metadata.shared_M_L;
let m_m = metadata.shared_M_M;
let m_r = metadata.shared_M_R;
let b_l = metadata.shared_B_L;
let b_m = metadata.shared_B_M;
let b_r = metadata.shared_B_R;
let checkValue = parseFloat(metadata.shared_BCP_Quantity); // Parse
„Check“ as a float (number)

// Construct the key for the new JSON
let key = `${u_l}, ${u_m}, ${u_r}, ${m_l}, ${m_m}, ${m_r}, ${b_l},
${b_m}, ${b_r}`;

// If BCP already exists and is a string, parse it. Otherwise, use
the provided value or initialize as an empty object.
let bcp = (typeof metadata.shared_BCP === 'string') ?
JSON.parse(metadata.shared_BCP) : (metadata.shared_BCP || {});

// Update/Add the key-value pair
if (bcp[key]) {
    // If the key exists, add the checkValue to the existing value
    bcp[key] += checkValue;
} else {
    // If the key doesn't exist, add the new key-value pair
    bcp[key] = checkValue;
}
// Send the updated BCP as a new shared attribute directly as a JSON
object
msg.BCP = bcp;
return { msg: msg, msgType: „POST_ATTRIBUTES_REQUEST“ };
msg.BCP_Quantity = 0;

return { msg: msg, msgType: „POST_ATTRIBUTES_REQUEST“ };
```

Vozlišče »Reset BCP and BCP_Quantity«, koda:

```
// Assuming the Finish attribute has been detected as updated already
```

```
// Set BCP to an empty JSON object
msg.BCP = {};

// Reset the other attributes to „R“
msg.U_L = „R“;
msg.U_M = „R“;
msg.U_R = „R“;
msg.M_L = „R“;
msg.M_M = „R“;
msg.M_R = „R“;
msg.B_L = „R“;
msg.B_M = „R“;
msg.B_R = „R“;
msg.BCP_Quantity = 0;

return { msg: msg, msgType: „POST_ATTRIBUTES_REQUEST“ };
```

Vozlišče »Create Order JSON«, koda:

```
// Fetch the ENG, BCP, SPT, ID_Narocila, and Ime_Narocila attributes
from metadata
let eng = (typeof metadata.shared_ENG === 'string') ?
JSON.parse(metadata.shared_ENG) : (metadata.shared_ENG || {});
let bcp = (typeof metadata.shared_BCP === 'string') ?
JSON.parse(metadata.shared_BCP) : (metadata.shared_BCP || {});
let spt = (typeof metadata.shared_SPT === 'string') ?
JSON.parse(metadata.shared_SPT) : (metadata.shared_SPT || {});
let idNarocila = metadata.shared_ID_Narocila;
let imeNarocnika = metadata.shared_Ime_Narocnika;

// Check if ID_Narocila and Ime_Narocila are not empty strings, „0“,
or 0
if (!idNarocila || idNarocila === „„ || idNarocila === „0“ ||
idNarocila === 0 || !imeNarocnika || imeNarocnika === „„ ||
imeNarocnika === 0) {
    // If the check fails, return false
    return false;
}
if (!Object.keys(bcp).length && !Object.keys(eng).length &&
!Object.keys(spt).length) {
    msg.Order = {};
    return { msg: msg, msgType: „POST_ATTRIBUTES_REQUEST“ };
}
// Get the current date and time and format it
let now = new Date();
let day = String(now.getDate()).padStart(2, '0');
```

```
let month = String(now.getMonth() + 1).padStart(2, '0'); // Months
are 0-based
let year = now.getFullYear();
let hours = String(now.getHours()).padStart(2, '0');
let minutes = String(now.getMinutes()).padStart(2, '0');
let formattedDate = `${day}.${month}.${year} ${hours}:${minutes}`;

// Construct the „Order“ object in the specified order
let order = {};
order[„Date&Time“] = formattedDate;
order.ID_Narocila = idNarocila;
order.Ime_Narocnika = imeNarocnika;

// Merge BCP, ENG, and SPT into the order object
Object.assign(order, bcp, eng, spt);

// Remove Model1, Model2, and Spare_Part if their values are 0
if (order.Model1_Quantity === 0) {
    delete order.Model1_Quantity;
}
if (order.Model2_Quantity === 0) {
    delete order.Model2_Quantity;
}
if (order.Spare_Part_Quantity === 0) {
    delete order.Spare_Part_Quantity;
}

// Assign the order to the „Order“ attribute in the msg
msg.Order = order;

return { msg: msg, msgType: „POST_ATTRIBUTES_REQUEST“ };
```

Vozlišče »Order History«, koda:

```
/// Fetch the current Order value from metadata
let currentOrder = metadata.shared_Order;

// If the currentOrder is a string representation, parse it to an
actual object.
if (typeof currentOrder === 'string') {
    try {
        currentOrder = JSON.parse(currentOrder);
    } catch (e) {
        //  If  there's  an  error  parsing,  don't  append  to
Order_History.
        return { msg: msg, msgType: „POST_ATTRIBUTES_REQUEST“ };
    }
}
```

```
// Remove the Date&Time attribute from currentOrder
if (currentOrder[„Date&Time“]) {
    delete currentOrder[„Date&Time“];
}

// Fetch the existing Order_History attribute and parse it or initialize a new object if it doesn't exist.
let orderHistory = (typeof metadata.shared_Order_History === 'string') ? JSON.parse(metadata.shared_Order_History) : (metadata.shared_Order_History || {});

// Fetch the existing Order_History_Display string or initialize it if it doesn't exist.
let orderHistoryDisplay = metadata.shared_Order_History_Display || '';

// Check if the currentOrder object is empty.
let isCurrentOrderEmpty = (typeof currentOrder === 'object') && !Object.keys(currentOrder).length;

if (!isCurrentOrderEmpty) {
    // Construct the current date and time in the desired format „M/dd/yyyy, hh:mm:ss a“
    let now = new Date();
    let formattedDate = now.toLocaleString();

    orderHistory[formattedDate] = currentOrder;

    // Prepare the current order's legible representation.
    let currentOrderDisplay = formattedDate;
    for (let order in currentOrder) {
        currentOrderDisplay += ' ' + order + ': ' + currentOrder[order] + ';';
    }

    // Append to the Order_History_Display string using „ || „ as separator between orders.
    if (orderHistoryDisplay) {
        orderHistoryDisplay += ' || ';
    }
    orderHistoryDisplay += currentOrderDisplay;
}

// Update msg object with both Order_History and Order_History_Display attributes.
msg.Order_History = orderHistory;
msg.Order_History_Display = orderHistoryDisplay.trim();
```

```
return { msg: msg, msgType: „POST_ATTRIBUTES_REQUEST“ };
```

Spremenjena koda pripomočka Attributes card:

Pripomoček »Order«, koda:

```
self.onInit = function() {
    self.ctx.datasourceTitleCells = [];
    self.ctx.valueCells = [];
    self.ctx.labelCells = [];

    for (var i = 0; i < self.ctx.datasources.length; i++) {
        var tbDatasource = self.ctx.datasources[i];

        var datasourceId = 'tbDatasource' + i;
        self.ctx.$container.append(„<div id='„ + datasourceId + „'
class='tbDatasource-container'></div>„);
        var   datasourceContainer   =   $('#'   +   datasourceId,
self.ctx.$container);

        datasourceContainer.append(„<div      class='tbDatasource-
title'>„ + tbDatasource.name + „</div>„);
        var    datasourceTitleCell    =    $('.tbDatasource-title',
datasourceContainer);
        self.ctx.datasourceTitleCells.push(datasourceTitleCell);

        var tableId = 'table' + i;
        datasourceContainer.append(„<table  id='„  +  tableId  +  „'
class='tbDatasource-table'></table>„);
        var table = $('#' + tableId, self.ctx.$container);

        // Create the header row for „Item:“ and „Quantity:“
        table.append(„<tr         style='height:         50px;'><td
class='tbDatasource-header'  style='font-weight:  bold;  font-size:
larger;  text-align:  left;  color:  black;  vertical-align:  middle;
padding-left:   15px;'><i>Item:</i></td><td   class='tbDatasource-
header'  style='font-weight:  bold;  font-size:  larger;  text-align:
right;  color:  black;  vertical-align:  middle;  padding-right:
15px;'><i>Quantity:</i></td></tr>„);

        for (var a = 0; a < tbDatasource.dataKeys.length; a++) {
            var dataKey = tbDatasource.dataKeys[a];
            var labelCellId = 'labelCell' + a;
            var cellId = 'cell' + a;
            table.append(„<tr><td  class='tbDatasource-label'  id='„
+ labelCellId + „' style='text-align: left; color: black; padding-
left: 5px;'>„ + dataKey.label + „</td><td class='tbDatasource-value'
```

```
id='„ + cellId + „' style='text-align: center; color:
black;'></td></tr„);
            var labelCell = $('#' + labelCellId, table);
            self.ctx.labelCells.push(labelCell);
            var valueCell = $('#' + cellId, table);
            self.ctx.valueCells.push(valueCell);
        }
    }

    self.onResize();
}



self.onDataUpdated = function() {
    var allZero = true;

    for (var i = 0; i < self.ctx.valueCells.length; i++) {
        var cellData = self.ctx.data[i];
        var textValue;

        if (cellData && cellData.data && cellData.data.length > 0)
{
            var tvPair = cellData.data[cellData.data.length - 1];
            var value = tvPair[1];

            if (cellData.dataKey.label === „BCP“) {
                self.ctx.labelCells[i].html(„Box Color Pattern“);
                try {
                    var parsedValue = JSON.parse(value);

                    if (Object.keys(parsedValue).length === 0) {
                        textValue = „0“;
                    } else {
                        textValue = „„;
                        for (var pattern in parsedValue) {
                            if
(parsedValue.hasOwnProperty(pattern)) {
                                textValue += „<i style='color:
grey;'>&quot;“ + pattern + „&quot;:</i> „ + parsedValue[pattern] +
„<br>„;
                            }
                        }
                        allZero = false;
                    }
                } catch (e) {
                    textValue = value.toString();
                }
```

```
                if (textValue === „0“) {
                    self.ctx.valueCells[i].parent().hide();
                } else {
                    self.ctx.valueCells[i].html(textValue);
                    self.ctx.valueCells[i].parent().show();
                }
            } else {
                if (isNumber(value) && value !== 0) {
                    allZero = false;
                    if (Number.isInteger(value)) {
                        textValue = value.toString();
                    } else {
                        var decimals = self.ctx.decimals;
                        var units = self.ctx.units;
                        if      (cellData.dataKey.decimals      ||
cellData.dataKey.decimals === 0) {
                            decimals = cellData.dataKey.decimals;
                        }
                        if (cellData.dataKey.units) {
                            units = cellData.dataKey.units;
                        }
                        textValue                                    =
self.ctx.utils.formatValue(value, decimals, units, true);
                    }
                    self.ctx.valueCells[i].html(textValue);
                    self.ctx.valueCells[i].parent().show();
                } else {
                    self.ctx.valueCells[i].parent().hide();
                }
            }
        } else {
            self.ctx.valueCells[i].parent().hide();
        }
    }

    var   headerRow   =   self.ctx.$container.find(„.tbDatasource-
header“).parent();
    if (allZero) {
        headerRow.hide();
    } else {
        headerRow.show();
    }

    var noItemsRow = self.ctx.$container.find(„.no-items-row“);
    if (allZero && !noItemsRow.length) {
```

```
        noItemsRow = $(„<tr class='no-items-row'><td colspan='2'
style='text-align: center; font-weight: bold; color: black; font-
size: 20px; padding: 10px;'>Please select your items!</td></tr>„);

self.ctx.valueCells[0].parent().parent().append(noItemsRow);
    } else if (!allZero) {
        noItemsRow.remove();
    }

    function isNumber(n) {
        return !isNaN(parseFloat(n)) && isFinite(n);
    }
}


self.onResize = function() {
    var datasourceTitleFontSize = self.ctx.height / 8;
    if (self.ctx.width / self.ctx.height <= 1.5) {
        datasourceTitleFontSize = self.ctx.width / 12;
    }
    datasourceTitleFontSize   =   Math.min(datasourceTitleFontSize,
20);
    for (var i = 0; i < self.ctx.datasourceTitleCells.length; i++)
{
        self.ctx.datasourceTitleCells[i].css('font-size',
datasourceTitleFontSize + 'px');
        self.ctx.datasourceTitleCells[i].css('color', 'black');
    }
    var valueFontSize = self.ctx.height / 9;
    var labelFontSize = self.ctx.height / 9;
    if (self.ctx.width / self.ctx.height <= 1.5) {
        valueFontSize = self.ctx.width / 15;
        labelFontSize = self.ctx.width / 15;
    }
    valueFontSize = Math.min(valueFontSize, 18);
    labelFontSize = Math.min(labelFontSize, 18);

    for (i = 0; i < self.ctx.valueCells.length; i++) {
        self.ctx.valueCells[i].css('font-size',   valueFontSize   +
'px');
        self.ctx.valueCells[i].css('height', valueFontSize * 2.5 +
'px');
        self.ctx.valueCells[i].css('padding',     '0px     '     +
valueFontSize + 'px');
        self.ctx.valueCells[i].css('text-align', 'right');

        self.ctx.labelCells[i].css('font-size',   labelFontSize   +
'px');
```

```
        self.ctx.labelCells[i].css('height', labelFontSize * 2.5 +
'px');
        self.ctx.labelCells[i].css('padding',     '0px     '     +
labelFontSize + 'px');
    }
}

self.onDestroy = function() {}
```

Pripomoček »Display Order History«, koda:

```
self.onInit = function() {
    self.ctx.valueCells = [];
    self.ctx.labelCells = [];

    for (var i=0; i < self.ctx.datasources.length; i++) {
        var tbDatasource = self.ctx.datasources[i];
        var datasourceId = 'tbDatasource' + i;

        self.ctx.$container.append(„<div id='„ + datasourceId + „'
class='tbDatasource-container'></div>„);
        var   datasourceContainer   =   $('#'   +   datasourceId,
self.ctx.$container);

        var tableId = 'table' + i;
        datasourceContainer.append(„<table  id='„  +  tableId  +  „'
class='tbDatasource-table'><col                  width='30%'><col
width='70%'></table>„);
        var table = $('#' + tableId, self.ctx.$container);
        for (var a = 0; a < tbDatasource.dataKeys.length; a++) {
            var dataKey = tbDatasource.dataKeys[a];
            var labelCellId = 'labelCell' + a;
            var cellId = 'cell' + a;
            table.append(„<tr><td  id='„  +  labelCellId  +  „'>„  +
dataKey.label + „</td><td id='„ + cellId + „'></td></tr>„);
            var labelCell = $('#' + labelCellId, table);
            self.ctx.labelCells.push(labelCell);
            var valueCell = $('#' + cellId, table);
            self.ctx.valueCells.push(valueCell);
        }
    }
    self.onResize();
}

self.onDataUpdated = function() {
    for (var i = 0; i < self.ctx.valueCells.length; i++) {
        var cellData = self.ctx.data[i];
```

```
        if (cellData && cellData.data && cellData.data.length > 0)
{
            var tvPair = cellData.data[cellData.data.length - 1];
            var value = tvPair[1];
            var textValue;

            if (isNumber(value)) {
                var decimals = self.ctx.decimals;
                var units = self.ctx.units;
                if          (cellData.dataKey.decimals          ||
cellData.dataKey.decimals === 0) {
                    decimals = cellData.dataKey.decimals;
                }
                if (cellData.dataKey.units) {
                    units = cellData.dataKey.units;
                }
                textValue    =    self.ctx.utils.formatValue(value,
decimals, units, true);
            } else {
                textValue = value;
            }

            if (self.ctx.labelCells[i].text() === „ORDER HISTORY:“)
{
                self.ctx.labelCells[i].css('color', 'black');
                self.ctx.labelCells[i].css('font-weight', 'bold');

                // Check and format „X, X, X, X, X, X, X, X, X“
pattern (where X is any letter)
                var pattern = /([A-Z]), ([A-Z]), ([A-Z]), ([A-Z]),
([A-Z]), ([A-Z]), ([A-Z]), ([A-Z]), ([A-Z])/g;
                if (pattern.test(textValue)) {
                    textValue  =  textValue.replace(pattern,  'Box
Color Pattern („$1, $2, $3, $4, $5, $6, $7, $8, $9“)');
                }

                // ... [Rest of the ORDER HISTORY format remains the
same]
                textValue  =  textValue.replace(/Model1_Quantity/g,
„Engraving Model 1“);
                textValue  =  textValue.replace(/Model2_Quantity/g,
„Engraving Model 2“);
                textValue                                          =
textValue.replace(/Spare_Part_Quantity/g, „Spare Part“);
                textValue  =  textValue.replace(/ID_Narocila/g,  „ID
Naročila“);
                textValue   =   textValue.replace(/Ime_Narocnika/g,
„Ime Naročnika“);
```

```
                var dateTimeRegex = /(\d{1,2}\/\d{1,2}\/\d{4},
\d{1,2}:\d{1,2}:\d{1,2} (AM|PM))/g;
                var matches = textValue.match(dateTimeRegex);
                if (matches) {
                    matches.forEach(function(dateTimeStr) {
                        var formattedDateTime = „<span
style='color: gray;'><b><i>„ + dateTimeStr + „</i></b></span>„;
                        textValue = textValue.replace(dateTimeStr,
formattedDateTime);
                    });
                }

                textValue = „<span style='color: black;'>„ +
textValue + „</span>„;
                textValue = textValue.replace(/\|\|/g, „<br/>„);
            }
            self.ctx.valueCells[i].html(textValue);
        }
    }

    function isNumber(n) {
        return !isNaN(parseFloat(n)) && isFinite(n);
    }
}


self.onResize = function() {
    var valueFontSize = self.ctx.height/9;
    var labelFontSize = self.ctx.height/9;
    if (self.ctx.width/self.ctx.height <= 1.5) {
        valueFontSize = self.ctx.width/15;
        labelFontSize = self.ctx.width/15;
    }
    valueFontSize = Math.min(valueFontSize, 18);
    labelFontSize = Math.min(labelFontSize, 18);

    for (var i = 0; i < self.ctx.valueCells.length; i++) {
        self.ctx.valueCells[i].css('font-size',
valueFontSize+'px');
        self.ctx.valueCells[i].css('height',
valueFontSize*2.5+'px');
        self.ctx.valueCells[i].css('padding', '0px ' +
valueFontSize + 'px');
        self.ctx.labelCells[i].css('font-size',
labelFontSize+'px');
        self.ctx.labelCells[i].css('height',
labelFontSize*2.5+'px');
```

```
        self.ctx.labelCells[i].css('padding',     '0px     '     +
labelFontSize + 'px');
    }
}

self.onDestroy = function() {
}
```

Pripomoček »Box Colour Pattern display«, koda:

```
self.onInit = function () {
    self.ctx.datasourceTitleCells = [];
    self.ctx.valueCells = [];

    var                          gridRows                          =
Math.ceil(self.ctx.datasources[0].dataKeys.length / 3);

    for (var i = 0; i < self.ctx.datasources.length; i++) {
        var tbDatasource = self.ctx.datasources[i];

        var datasourceId = 'tbDatasource' + i;
        self.ctx.$container.append(
            „<div id='„ + datasourceId +
            „' class='tbDatasource-container'></div>„
        );

        var datasourceContainer = $('#' + datasourceId,
            self.ctx.$container);

        datasourceContainer.append(
            „<div    class='tbDatasource-title'    style='display:
none;'>„ +
            tbDatasource.name + „</div>„
        );

        var tableId = 'table' + i;
        datasourceContainer.append(
            „<table id='„ + tableId +
            „' class='tbDatasource-table'></table>„
        );

        var table = $('#' + tableId, self.ctx.$container);

        for (var row = 0; row < gridRows; row++) {
            table.append(„<tr>„);

            for (var col = 0; col < 3; col++) {
                var dataIndex = row * 3 + col;
```

```
                if (dataIndex < tbDatasource.dataKeys.length) {
                    var dataKey = tbDatasource.dataKeys[dataIndex];
                    var cellId = 'cell' + dataIndex;
                    table.append(„<td id='„ + cellId +
                        „'                      class='tbCell'><div
class='tbValue'></div></td>„);
                    var valueCell = $('#' + cellId + ' .tbValue',
table);
                    self.ctx.valueCells.push(valueCell);
                } else {
                    // Add empty cells for the remaining positions
in the grid
                    table.append(„<td></td>„);
                }
            }

            table.append(„</tr>„);
        }
    }

    self.onResize();
}

self.onDataUpdated = function () {
    for (var i = 0; i < self.ctx.valueCells.length; i++) {
        var cellData = self.ctx.data[i];
        if (cellData && cellData.data && cellData.data.length > 0)
{
            var tvPair = cellData.data[cellData.data.length - 1];
            var value = tvPair[1];

            self.ctx.valueCells[i].html(„„);

            // Change background color based on value (R=Red,
G=Green, Y=Yellow)
            switch (value) {
                case 'R':
                    self.ctx.valueCells[i].css('background-color',
'#EA4C4C');
                    break;
                case 'G':
                    self.ctx.valueCells[i].css('background-color',
'#A1C45A');
                    break;
                case 'Y':
                    self.ctx.valueCells[i].css('background-color',
'#F1C550');
```

```
                    break;
                default:
                    self.ctx.valueCells[i].css('background-color',
'white');
                    break;
            }
        }
    }
}

self.onResize = function () {
    var valueCellSize = Math.min((self.ctx.width - 20) / 3,
(self.ctx.height - 20) / 3); // Slightly smaller cells with 20px
padding
    var valueFontSize = valueCellSize / 3;

    $('.tbCell').css({
        'width': valueCellSize + 'px',
        'height': valueCellSize + 'px',
        'border': '2px solid black',
        'box-sizing': 'border-box',
        'padding': 0 // Remove padding
    });

    $('.tbValue').css({
        'width': '100%',
        'height': '100%',
        'box-sizing': 'border-box',
        'font-size': valueFontSize + 'px',
        'line-height': valueCellSize + 'px',
        'text-align': 'center'
    });
}

self.onDestroy = function () {
}
```