Razvoj web aplikacije za upravljanje korisnički definiranim relacijskim podacima

Jurković, Kristijan

Sveučilište Jurja Dobrile u Puli
Tehnički fakultet u Puli

KRISTIJAN JURKOVIĆ

RAZVOJ WEB APLIKACIJE ZA UPRAVLJANJE KORISNIČKI DEFINIRANIM RELACIJSKIM PODACIMA

Završni rad

Pula, Rujan, 2022. godine

Sveučilište Jurja Dobrile u Puli
Tehnički fakultet u Puli

KRISTIJAN JURKOVIĆ

RAZVOJ WEB APLIKACIJE ZA UPRAVLJANJE KORISNIČKI DEFINIRANIM RELACIJSKIM PODACIMA

Završni rad

JMB: 0303007494, izvanredni student

Studijski smjer: Računarstvo

Predmet: Dinamičke Web Aplikacije

Znanstveno područje: Tehničke znanosti
Znanstveno polje: Računarstvo
Znanstvena grana: Informacijski sustavi
Mentor: doc. dr. sc. Nikola Tanković

Pula, Rujan, 2022. godine

Tehnički fakultet u Puli

IZJAVA O AKADEMSKOJ ČESTITOSTI

Ja, dolje potpisani Kristijan Jurković, kandidat za prvostupnika računarstva ovime izjavljujem da je ovaj Završni rad rezultat isključivo mojega vlastitog rada, da se temelji na mojim istraživanjima te da se oslanja na objavljenu literaturu kao što to pokazuju korištene bilješke i bibliografija. Izjavljujem da niti jedan dio Završnog rada nije napisan na nedozvoljeni način, odnosno da je prepisan iz kojega necitiranog rada, te da ikoji dio rada krši bilo čija autorska prava. Izjavljujem, također, da nijedan dio rada nije iskorišten za koji drugi rad pri bilo kojoj drugoj visokoškolskoj, znanstvenoj ili radnoj ustanovi.

Student _____

U Puli, 12.09.2022.

Tehnički fakultet u Puli

IZJAVA O KORIŠTENJU AUTORSKOG DJELA

Ja, Kristijan Jurković dajem odobrenje Sveučilištu Jurja Dobrile u Puli, kao nositelju prava iskorištavanja, da moj završni rad pod nazivom

RAZVOJ WEB APLIKACIJE ZA UPRAVLJANJE KORISNIČKI DEFINIRANIM RELACIJSKIM PODACIMA

koristi na način da gore navedeno autorsko djelo, kao cjeloviti tekst trajno objavi u javnoj internetskoj bazi Sveučilišne knjižnice Sveučilišta Jurja Dobrile u Puli te kopira u javnu internetsku bazu završnih radova Nacionalne i sveučilišne knjižnice (stavljanje na raspolaganje javnosti), sve u skladu s Zakonom o autorskom pravu i drugim srodnim pravima i dobrom akademskom praksom, a radi promicanja otvorenoga, slobodnoga pristupa znanstvenim informacijama. Za korištenje autorskog djela na gore navedeni način ne potražujem naknadu.

U Puli, 12.09.2022.

Potpis _____

Sadržaj

1. Uvod

Softverska rješenja temelje se na digitalizaciji i unaprjeđenju poslovnih procesa. Relacijski modeli kao jedno od softverskih rješenja služe opisivanju stvarnog svijeta u digitalnom obliku te kako bi probleme prebacili u programsku domenu gdje ih možemo matematički riješiti (Pavlić, 2011).

Relacijski modeli su tradicionalno modelirana apstrakcija stvarnog svijeta kako bi dobili ekvivalent problema kojeg možemo riješiti programskim putem. Međutim kod izmjene poslovnih procesa nije ih jednostavno promijeniti zbog količine podataka koje relacije sadrže. Potrebne su migracije podataka u nove strukture te je s time moguć gubitak podataka zbog nedovoljnog planiranja ili jednostavno nepažnje programera. Uvođenje novih relacijskih modela kao i inicijalno modeliranje procesa, zahtijeva prijenos znanja domene s krajnjeg korisnika na razvojnog inženjera te uvijek može doći do pogreške u komunikaciji kod analize, planiranja i izvođenja implementacije (Hoffer, et al., 2012).

Jedan od mogućih rješenja su dinamički modeli. Dinamički modeli prebacuju modeliranje procesa na krajnjeg korisnika te se može uštedjeti vrijeme razvoja, izbjeći potencijalne probleme kod migracije podataka i prijenos domenskog znanja. Dinamičke relacije kreira sam korisnik koji je odgovoran za poslovni proces dok kroz definiciju validacije koju korisnik sam definira za svaki pojedinačni tip, programsko rješenje može izvršavati provjere nad unesenim podacima (Tanković, et al., 2012).

U ovom završnom radu napravljena je usporedba relacijskih i dinamičkih modela, istaknute su prednosti i mane dinamičkih modela te je kao primjer kreiran projektni zadatak kojim se pokazuje implementacija platformski nezavisnih modela od strane korisnika.

2. Relacijski modeli

Tradicionalne relacijske baze podataka su organizirane i uređene cjeline međusobno povezanih podataka. Često ne prate modele aplikacija koje koriste složenije tipove podataka te se razlikuju od modela koji su realizirani objektno orijentiranim jezicima. Kako bi se relacijski model razvio moraju se identificirati i istražiti posljedice koje će implementacija imati na sam dizajn modela (Hoffer et al., 2012).

2.1. Razvoj relacijskih modela

Tradicionalni relacijski modeli baziraju se na neredudantnom skupu podataka koji opisuju stanje sustava prema strukturiranom načinu opisanom u shemi baze podataka. Svaki proces u stvarnom svijetu kojeg se pokušava digitalizirati, potrebno je analizirati, isplanirati moguće posljedice modela na sustav te implementirati tako da je lako proširiv te da jednostavno opisuje određeni objekt

Kod razvoja relacijskih modela koristi se jezik za opis podataka *Data Definition Language* (DDL) koji je skup sintaksnih pravila pomoću kojeg opisujemo entitete odnosno relacije u bazi podataka. S druge strane imamo jezik za rukovanje podacima *Data Manipulation Language* (DML) koji je računalni jezik pomoću kojega programeri mogu manipulirati podacima u entitetima (Pavlić, 2011).

3. Dinamički definirani modeli

Prelaskom na dinamičke modele prebacujemo analizu poslovnog procesa, identifikaciju te samu implementaciju modela na krajnjeg korisnika. Razlikujemo generativni *Model Driven Development* (MDD) koji ukazuje na ideju da se modeli mogu koristiti samo za opis domenskih problema, te interpretativni MDD koji se može promatrati kao skup platformski nezavisnih modela koji su interpretirani u runtime okruženju. U daljnjem razmatranju koristit će se interpretativni platformski nezavisni modeli. Meta podaci definiraju zasebno svaki objekt kroz tipove podataka koji će se u njemu nalaziti (Tanković, et al., 2012) (Overeem, et al., 2018).

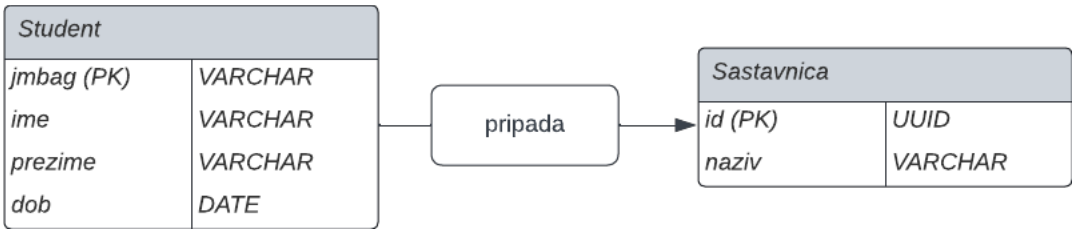
3.1. Potencijalni problemi dinamičkih modela

Postoji nekoliko potencijalnih problema kod kreiranja dinamičkih modela. Iako bi skratili vrijeme i trošak razvoja, zahtijevaju od krajnjeg korisnika poznavanje kompleksnosti modeliranja kako bi zadovoljio potrebe i zahtjeve poslovnog procesa kojeg korisnik želi modelirati. Modeliranje zahtijeva kreiranje modela prema ulazima krajnjeg korisnika korištenjem određenog alata i jezika za modeliranje. Kod definiranja alata bitno odabrati instrumente modeliranja koji su poznati krajnjem korisniku. (Tanković, et al., 2012)

3.2. Pohrana podataka

Korisnik, putem web sučelja može definirati meta model odabirom tipova podataka koje želi spremati, te validaciju nad podacima za određeni tip podatka. Spremanje meta modela i samih podataka definiranih meta modelom vrši se u JSON formatu. Uzmimo za primjer model Studenta s atributima: ime, prezime, datum rođenja, JMBAG, sastavnica_id. U tradicionalnom relacijskom modelu kreirali bismo relaciju Student kao što je prikazano na slici 3.1.

Slika 3.1. Relacijski model Student



Obrada podataka

Zatim se podaci mogu spremati u definirane relacije (tablica 3.1., tablica 3.2.)

Tablica 3.1. Podaci u relacijskom modelu Sastavnica

Sastavnica	
id	naziv
4a424158-c30b-4b22-9657-69b19cabb948	Tehnički fakultet

Obrada podataka

Tablica 3.2. Podaci u relacijskom modelu Student

Student				
jmbag	ime	prezime	dob	sastavnica_id
0303007494	Kristijan	Jurković	30.08.1985.	4a424158-c30b-4b22-9657-69b19cabb948

Obrada podataka

Dinamički meta model definira se u JSON formatu. S obzirom na to da se ne koristi tablična definicija dostupna u relacijskim bazama podataka, svaki od tipova dostupnih u meta modelu mora sadržavati naziv, tip podatka, validaciju podataka koji će se unositi te referencu ako ona postoji (slika 3.2.).

3.2. Dinamički model Student

```json
{
  "id": "545e9d78-eca5-4fb9-86f3-f07919b834bb",
  "name": "Student",
  "content": [{
      "label": "jmbag",
      "type": "string",
      "validation": "[0-9]{10}"
    },
    {
      "label": "ime",
      "type": "string",
      "validation": "[A-Za-z]+"
    },
    {
      "label": "prezime",
      "type": "string",
      "validation": "[A-Za-z]+"
    },
    {
      "label": "dob",
      "type": "date",
      "validation": null
    },
    {
      "label": "sastavnica",
      "type": "uuid",
      "validation": null,
      "references": "sastavnica"
    }
  ]
}
```

Obrada podataka

3.3. Validacija podataka

Validacija podataka kod dinamičkih modela se može vršiti na više načina. Programska implementacija može predefinirati validacije za određeni tip podatka te ponuditi korisniku odabir. Ukoliko takve validacije nisu usko specifične za podatke u poslovnom procesu korisnika, svakom tipu u meta modelu moguće je dodijeliti regularni izraz uz sam tip podatka kojim će se provjeravati ispravnost ulaznih podataka.

Kod implementacije sustava potrebno je pripaziti na sintaksu regularnih izraza s obzirom da programski jezici podržavaju, odnosno ne podržavaju određene sintakse; ukoliko se za primjer promatra programski jezik GoLang, podržana sintaksa će biti RE2 što je podskup izraza podržanih od PCRE biblioteke[1].

3.4. Izmjene poslovnih procesa i konzistentnost podataka

Izmjenama poslovnih procesa moraju se definirati novi meta modeli koji će zadovoljavati nove zahtjeve procesa. Prednost dinamički generiranih modela sastoji se u tome da uvođenje novih modele ne utječe na postojeće modele i podatke vezane za te modele.

Podaci su vezani za meta definiciju putem jedinstvenog identifikatora, te se sam meta model ne mijenja uvođenjem novih pravila, već se kreira novi model koji će biti definirati novu strukturu podataka, te osiguravati njihovu konzistentnost.

Jednostavna izmjena postojećih meta modela je moguća dok god ne postoje podaci vezani za meta model. Ukoliko podaci postoje za određeni meta model, potrebno je definirati izmijenjeni meta model, te migracijske skripte za postojeće podatke koji će opisati poveznice između podataka definiranih na starom meta modelu u odnosu na novi meta model.

[1] https://swtch.com/~rsc/regexp/regexp3.html#caveats

4. Arhitektura programskog rješenja

Projektni zadatak, koji je dio ovog rada, je programsko rješenje za kreiranje meta modela i unos podataka te je izvedeno kroz mikroservise u programskom jeziku GoLang te zapakirano u sustav za upravljanje spremnicima Docker (containers). Dostupno je na Github repozitoriju https://github.com/kjurkovic/airtable. Korisničko sučelje isprogramirano je pomoću biblioteke React u jeziku Javascript koristeći biblioteku Bootstrap CSS za definiranje vizualnih stilova. Za svaki mikroservis definirane su datoteke YAML tipa za lakše objavljivanje rješenja na različitim okruženjima Kubernetes.

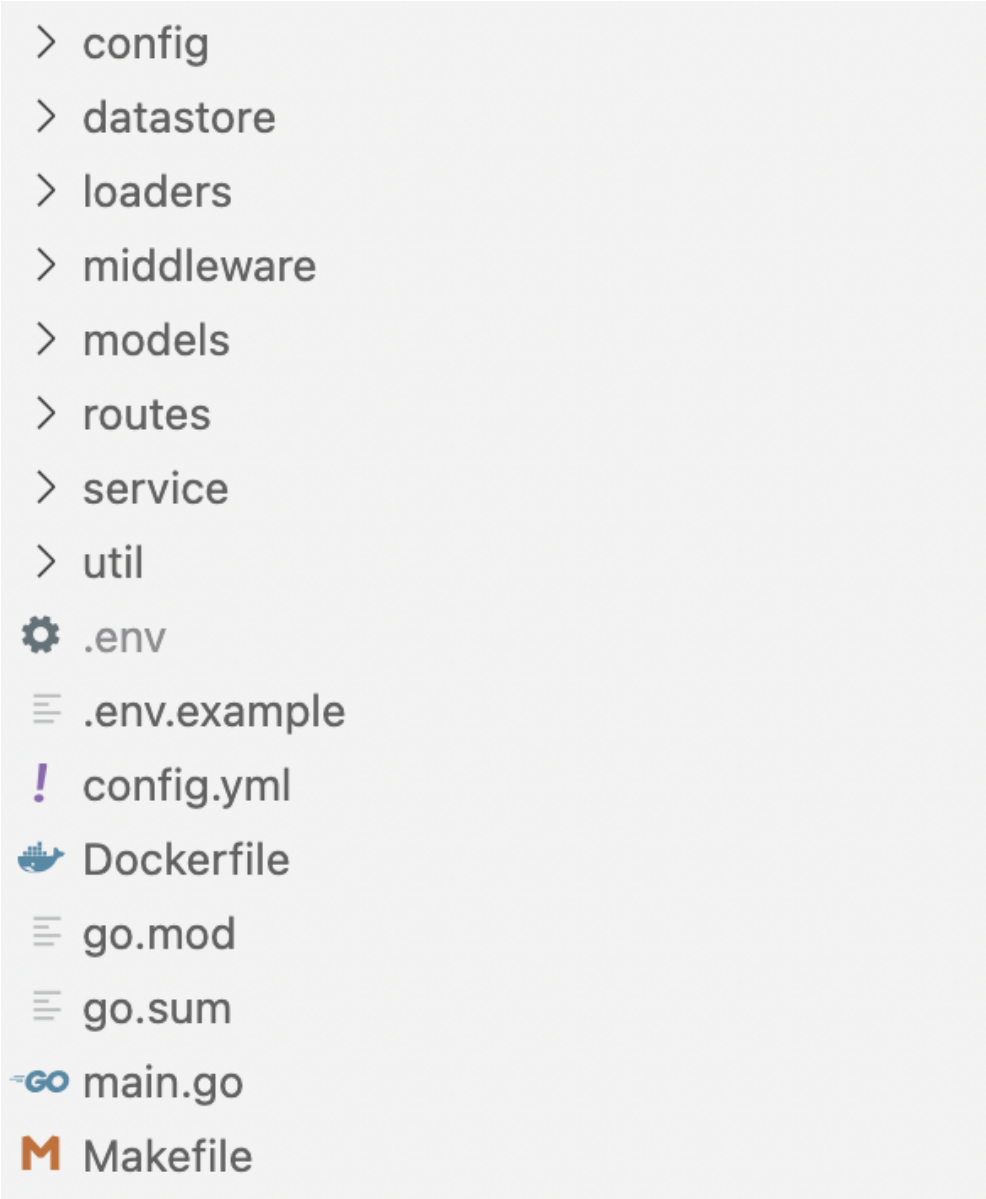
4.1. Mikroservisi

Programsko rješenje sastoji se od sedam mikroservisa koji međusobno komuniciraju HTTP(S) protokolom:

- Audit
- Auth
- Workspace
- Meta
- Data
- Notification
- Frontend

Struktura paketa svakog mikroservisa u GoLang jeziku je ista izuzev servisa *Notification* koji nema datastore paket obzirom da služi slanju email poruka. Mikroservis je podijeljen u 8 osnovnih paketa kao što je vidljivo na slici 4.1.

4.1. Struktura paketa mikroservisa



Obrada podataka

Main.go je datoteka koja sadrži ulaznu funkciju u program i inicijalizira web servis pomoću Loader obrasca definiranih za bazu podataka i aplikaciju u loaders paketu. Config paket sadrži YAML parser, te aplikacijski model za konfiguraciju mikroservisa. Parser uzima config.yml datoteku dostupnu u početnom direktoriju servisa, te iz YAML formata pretvara u GoLang strukture prema zadanoj definiciji aplikacijskog modela. Datastore paket definira repozitorije za pristup određenim relacijama u bazi podataka i inicijalizira SQL relacijske tablice prema modelima definiranim u models paketu. U middleware paketu drže se funkcije koje web servis koristi kod usmjeravanja HTTP zahtjeva prema određenoj funkciji kako bi se prije izvršavanja funkcije za specifični zahtjev izvršila provjera ili dodatna logika (primjerice provjerila prava pristupa korisnika koji radi zahtjev određenim resursima). Paket koji sadrži logiku za usmjeravanje HTTP zahtjeva zove se u routes, dok za poslovnu logiku koristimo service paket. Util paket sadrži funkcije koje ne pripadaju nijednom od navedenih paketa, ali se koriste u različitim paketima kao generičke funkcije, npr. hash funkcije za kreiranje refresh tokena.

4.1.1. Auth

Autentifikacija korisnika implementirana je kroz normu OAuth2 gdje klijent zahtijeva od poslužitelja pristup koji mu server omogućuje izdavanjem Access Tokena. Implementacija Access Tokena koristi JWT (JSON Web Token) koji definira kompaktnu i sigurnu metodu prijenosa podataka u JSON formatu jer je digitalno potpisana. Svaki drugi mikroservis koristi istu implementaciju JWTa, te je kao varijabla okruženja svakog mikroservisa definiran isti JWT tajni ključ čime izbjegavamo dodatne zahtjeve ostalih mikroservisa prema Auth servisu za dohvat korisničkih podataka.

Auth servis se sastoji od HTTP metoda i putanja za prijavu, registraciju, autentifikaciju putem refresh tokena, te dohvat i spremanje korisničkih podataka koji su navedeni u nastavku:

- POST **/auth/register**
  - Zahtjev: { firstName: string, lastName: string, email: string, password: string }
  - Odgovor: HTTP 200 OK { accessToken: string, refreshToken: string, accessTokenExpiresAt: long, refreshTokenExpiresAt: long }
- POST **/auth/login**
  - Zahtjev: { email: string, password: string }
  - Odgovor: HTTP 200 OK { accessToken: string, refreshToken: string, accessTokenExpiresAt: long, refreshTokenExpiresAt: long }
- POST **/auth/refresh-token**
  - Zahtjev: { refreshToken: string }
  - Odgovor: HTTP 200 OK { accessToken: string, refreshToken: string, accessTokenExpiresAt: long, refreshTokenExpiresAt: long }
- GET **/users**
  - Odgovor: HTTP 200 OK { content: { id: string, firstName: string, lastName: string, email: string}, pagination: { items: long, totalItems: long, page: long, totalPages: long } }
- GET **/users/:id**
  - Odgovor: HTTP 200 OK { id: string, firstName: string, lastName: string, email: string}
- PATCH **/users/:id**
  - Zahtjev: { id: string, firstName: string, lastName: string, email: string}
  - Odgovor: HTTP 202 Accepted
- PATCH **/users/:id/password**
  - Zahtjev: { password: string }
  - Odgovor: HTTP 202 Accepted
- DELETE **/users/:id**
  - Odgovor: HTTP 202 Accepted

4.1.2. Audit

Audit logiranje je proces dokumentiranja aktivnosti unutar programskog rješenja koji se koristi u nekoj organizaciji. Audit logovi snimaju pojavljivanja određenih događaja, vrijeme u koje se događaj odvio, odgovornu osobu koja je događaj inicirala i entitet odnosno objekt koji je zahvaćen tim događajem. Revizijom audit logova može se pratiti aktivnost korisnika, te mogu otkriti kršenje sigurnosti sustava.

Razlike između audit logova i normalnih sistemskih logova su informacije koje sadrže, njihova svrha te njihova nepromjenjivost.

Audit servis nije dostupan krajnjem korisniku za kreiranje zapisa, već je dostupan samo servisima unutar Kubernete, dok je dohvat audit podataka ograničen samo na administratore sustava. Servisi koji rade audit logove kreiraju HTTP zahtjev s potrebnim podacima prema Audit servisu.

Dostupni zahtjevi prema Audit servisu:

- POST **/logs**
    - Zahtjev { userId: string, type: string, obj: json }
    - Odgovor: HTTP 201 Created
- GET **/audit/:id?type={type}&page={page}&size={size}**
    - Odgovor: HTTP 200 OK { content: { id: string, userId: string, type: string, obj: string}, pagination: { items: long, totalItems: long, page: long, totalPages: long }

4.1.3. Workspace

Workspace servis sadrži putanje za CRUD operacije nad radnim okruženjima, te mogu služiti kako bi se meta modeli i njihovi podaci mogli grupirati u logičke cjeline. Prije kreiranja meta modela, potrebno je kreirati radno okruženje u kojem će se meta model nalaziti.

Dostupni zahtjevi prema Workspace servisu:

- POST **/workspace**
    - Zahtjev: { name: string }
    - Odgovor: HTTP 200 OK { id: string, name: string }
- GET **/workspace**
    - Odgovor: HTTP 200 OK [{ id: string, name: string }]
- PUT **/workspace/:id**
    - Zahtjev: { name: string }
    - Odgovor: HTTP 200 OK { id: string, name: string }
- DELETE **/workspace/:id**
    - Odgovor: HTTP 204 No Content

4.1.4. Meta

Meta servis služi za definiranje i upravljanje meta modelima. Izveden je kroz relacijski entitet. SQL entitet sadrži osnovne podatke meta objekta: id, naziv, datum kreiranja, datum izmjene te JSON kolumnu u koju se sprema definicija meta objekta.

Meta modeli se definiraju na korisničkom sučelju putem formi, te su dostupne korisniku koji ih je kreirao za daljnje uređivanje.

Dostupni zahtjevi prema Meta servisu:

- POST **/meta**
    - Zahtjev: { workspaceId: string, name: string: fields: [{ type: string, validation: string, label: string }], isPublic: boolean }
    - Odgovor: HTTP 201 Created
- PUT **/meta/:id**
    - Zahtjev: { workspaceId: string, name: string: fields: [{ type: string, validation: string, label: string }], isPublic: boolean }
    - Odgovor: : HTTP 202 Accepted
- GET **/meta/user/:id**
    - Odgovor: HTTP 200 OK { content: { id: string, userId: string, workspaceId: string, fields: [{id: string, metaId: string, label: string, type: string, validation: string}], name: string, isPublic: boolean }, pagination: { items: long, totalItems: long, page: long, totalPages: long }
- GET **/meta/:id**
    - Odgovor: HTTP 200 OK { id: string, userId: string, workspaceId: string, fields: [{id: string, metaId: string, label: string, type: string, validation: string}], name: string, isPublic: boolean }
- DELETE **/meta/:id**
    - Odgovor: HTTP 202 Accepted

Data servis se koristi za spremanje podataka vezanih za meta modele. Validacija podataka vrši se kod slanja podataka od strane korisnika na servis u skladu s definiranom validacijom u meta modelu. U nastavku se navode dostupni zahtjevi prema Data servisu:

- POST **/data**
    - Zahtjev: { content: json }
- Odgovor: HTTP 202 Accepted **/data/:metaId**
    - Odgovor: { content: { id: string, metaId: string, content: json }, pagination: { items: long, totalItems: long, page: long, totalPages: long }

4.1.5. Notification

Servis za notifikacije koristi se za slanje emailova kod određenih događaja u sustavu. Koristi se Sendgrid API kroz biblioteku kako bi se email poruke proslijeđivale relevantnom korisniku. Servis nije dostupan izvan Kubernetes klastera, te ga mogu koristiti samo servisi unutar klastera putem HTTP zahtjeva.

4.2. Kubernetes klaster

Za pokretanje cjelokupnog rješenja u produkcijskom okruženju koristi se Kubernetes klaster. Mikroservisi programskog rješenja su u klasteru definirani kao tip NodePort Service čime se mapira svaki servis na port range specificiran service-node-port-range zastavicom (Kubernetes, 2022). Za svaki NodePort Service može se definirati određeni port putem nodePort ključa u yml datoteci za određeni mikroservis (slika 4.1.)

4.1. NodePort Service

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  type: NodePort
  selector:
    app.kubernetes.io/name: MyApp
  ports:
    - port: 80
      targetPort: 80
      nodePort: 30007
```

https://kubernetes.io/docs/concepts/services-networking/service/

Notification mikroservis definiran je kao ClusterIP Service čime ograničavamo njegovu dostupnost samo servisima unutar klastera dok vanjski pristup nije dozvoljen. ClusterIP je u Kubernetesu defaultni tip servisa te je odabran ukoliko nijedan drugi tip nije izričito specificiran u yaml datoteci (slika 4.2.)

4.2. ClusterIP Service

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app.kubernetes.io/name: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

https://kubernetes.io/docs/concepts/services-networking/service/

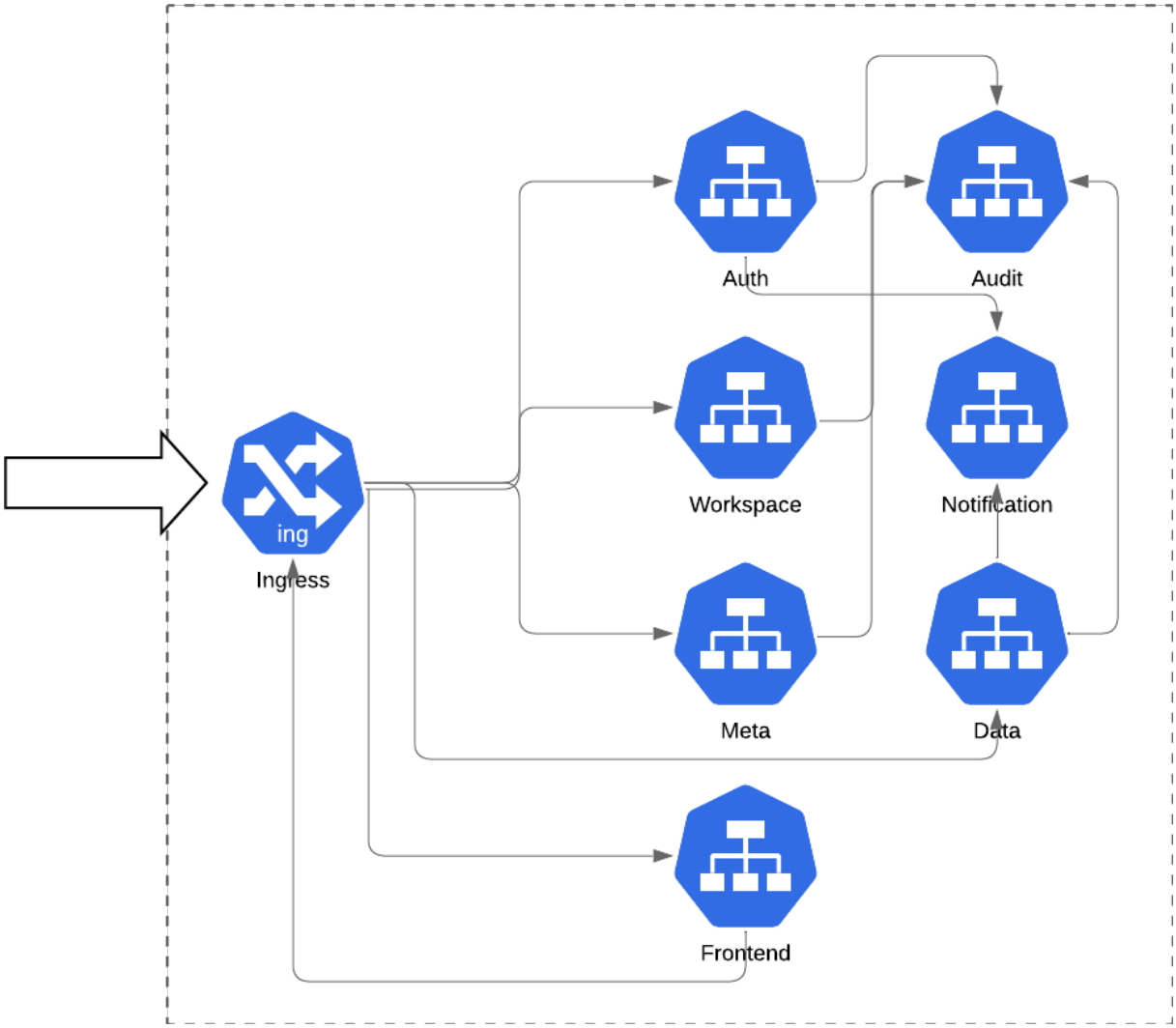
Resurs Ingress koristi se za konfiguriranje pristupa servisima unutar klastera putem pravila definiranih u yaml datoteci (slika 4.3.). Svaki od mikroservisa ima pristup definiran putem regularnog izraza.

4.3. Definicija Ingress pravila

```
rules:
  {{ if .Values.ingress.hostEnabled }}
  - host: {{ .Values.ingress.host  }}
    http:
  {{ else }}
  - http:
      {{ end }}
      paths:
        - path: /(auth|user)(/|$)(.*)
          pathType: Prefix
          backend:
            service:
              name: auth-service
              port:
                number: 80
        - path: /audit(/|$)(.*)
          pathType: Prefix
          backend:
            service:
              name: audit-service
              port:
                number: 80
        - path: /workspace(/|$)(.*)
          pathType: Prefix
          backend:
            service:
              name: workspace-service
              port:
                number: 80
        - path: /meta(/|$)(.*)
          pathType: Prefix
          backend:
            service:
              name: meta-service
              port:
                number: 80
        - path: /data(/|$)(.*)
          pathType: Prefix
          backend:
            service:
              name: data-service
              port:
                number: 80
```

Komunikacija između servisa pokazana je na slici 4.4. Svaki servis je moguće skalirati horizontalno bez dodatnih podešavanja samog servisa. Kubernetes podržava horizontalno skaliranje, te je potrebno definirati samo broj replika koje će servis imati. Za komunikaciju se koristi HTTP/1.1. protokol, stoga će load balanceri koji se koriste za proslijeđivanje normalno raditi bez dodatnih implementacija mesh servisa, kao što bi bio slučaj kod korištenja HTTP/2 protokola.

4.4. Tok podataka između mikroservisa

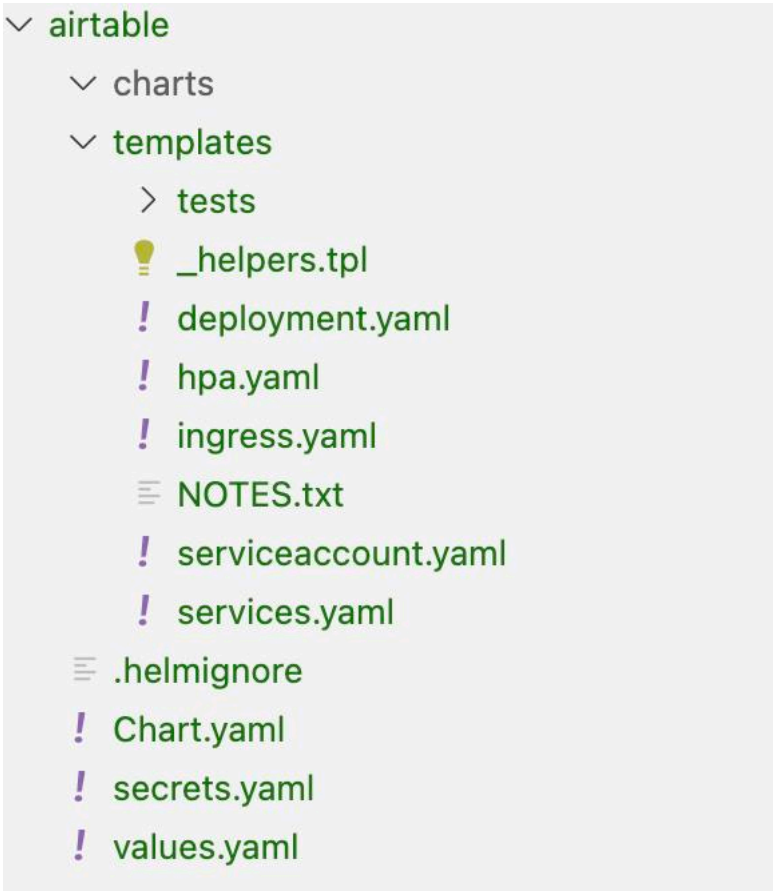


4.3. Helm karte

Helm je softver odnosno paketni menadžer za Kubernetes aplikacije. Pomoću yaml datoteka, koje se nazivaju Charts, Helm definira način na koji će se aplikacije na Kubernetes klasterima instalirati odnosno na koji način će se podići verzija postojećih aplikacija.

Programsko rješenje ima kreirane Helm datoteke koje definiraju kreiranje Ingress kontrolera, mikroservisa i tajnih podataka. Tajni podaci koji se koriste kao varijable okruženja dostupne servisima u Kubernetes klasteru. Na slici 4.5. prikazane su datoteke koje Helm koristi za instalaciju odnosno izbacivanje nove verzije programskog rješenja na Kubernetes klaster (Helm, 2022).

4.5. Struktura Helm direktorija



Obrada podataka

Instalacija programskog rješenja vrši se naredbom

helm install airtable . -f values.yaml

u ${projectRoot }/scripts/airtable direktoriju.

4.4. Proširivanje sustava

Programsko rješenje izvedeno je kroz mikroservise. Proširivanje sustava dodatnim funkcionalnostima moguće je dodavanjem novih mikroservisa. Ukoliko se zahtijeva komunikacija s postojećim servisima potrebno je kreirati klijentske biblioteke za komunikaciju. Sve konfiguracijske varijable potrebno je dodati kroz config.yaml odnosno putem varijabli okruženja kako bi se te postavke mogle konfigurirati kroz tajne podatke u Kubernetes klasteru za produkcijsko okruženje.

Direktorij scripts sadrži datoteku docker-compose.yml u kojoj je potrebno konfigurirati novi servis kako bi bio dostupan za pokretanje kroz Docker u razvojnom okruženju. Potrebno je i definirati Helm chart kako bi se novi servis mogao uspješno instalirati na postojeći Kubernetes klaster.

5. Zaključak

Razvoj softvera temeljen modelima (*Model Driven Development)* služi kako bi se unaprijedila produktivnost razvojnih inženjera, te poboljšala kvaliteta softvera. Interpretativni MDD, gdje se koriste platformski nezavisni modeli, orijentiran je na pojednostavljenu reprezentaciju kompleksnih sustava, te se može koristiti za automatizaciju jednostavnijih procesa.

Modeliranje sustava je kompleksan proces. Postoje različiti alati za modeliranje, čak i oni koji krajnjem korisniku daju mogućnost relativno jednostavnog modeliranja procesa. Međutim, svaki od tih alata zahtijeva poznavanje kompleksnosti modeliranja, što otvara mogućnosti pogreške kod definiranja modela. Jedan od sljedećih koraka za kreiranje dinamičkih modela od strane korisnika je kreiranje intuitivnijih i pojednostavljenih korisničkih sučelja za izradu kompleksnijih sustava čime bi se proces modeliranja mogao više približiti krajnjim korisnicima.

6. Literatura

Hoffer, J. A., Venkataraman, R. & Topi, H., 2012. *Modern Database Management (11th Edition).* s.l.:an.

Tanković, N., Vukotić, D. & Žagar, M., 2012. *Rethinking Model Driven Development: Analysis and Opportunities.* s.l.:an.

Pavlić, M., 2011. *Oblikovanje baze podataka.* Rijeka: an.

Overeem, M., Jansen, S. & Fortuin, S., 2018. *Generative versus Interpretive Model-Driven Development: Moving Past 'It Depends'.* s.l.:an.

Kubernetes, 2022. *Kubernetes.* [Mrežno]
Available at: https://kubernetes.io/docs/concepts/services-networking/service/
[Pokušaj pristupa 8 9 2022].

Helm, 2022. *Helm.* [Mrežno]
Available at: https://helm.sh/docs/topics/charts/
[Pokušaj pristupa 8 9 2022].

Sažetak

Razvoj softverskih rješenja se oslanja na dokumentiranju i digitalizaciji postojećih poslovnih procesa. Formulari korišteni u poslovnim procesima modeliraju se prenošenjem postojećih atributa formulara u relacijske baze podataka. Svaka izmjena rješenja zbog izmjene poslovnih procesa mora osigurati konzistentnost i sigurnost postojećih podataka, te zahtjeva određeno vremensko razdoblje kako bi se novi poslovni procesi implementirali.

Relacijski modeli su tradicionalno modelirana apstrakcija stvarnog svijeta kako bi dobili ekvivalent problema kojeg možemo riješiti programskim putem. Međutim, kod izmjene poslovnih procesa nije ih jednostavno promijeniti zbog količine podataka koje relacije sadrže. Potrebne su migracije podataka u nove strukture, te je s time moguć gubitak podataka zbog nedovoljnog planiranja ili jednostavno nepažnje programera. Uvođenje novih relacijskih modela kao i inicijalno modeliranje procesa, zahtijeva prijenos znanja domene s krajnjeg korisnika na razvojnog inženjera, te uvijek može doći do pogreške u komunikaciji kod analize, planiranja i izvođenja implementacije.

Jedan od mogućih rješenja su dinamički modeli koji prebacuju modeliranje procesa na krajnjeg korisnika te se programsko rješenje brine o konzistenciji podataka vezanih za definiciju modela (meta model). Nisu potrebne migracije već jednostavno nova definicija modela koja će opisati novi odnosno unaprijeđeni poslovni proces. Nastavno na definiciju modela, prikazan je postupak kreiranja servisa za meta modeliranje i unos podataka pomoću Helm rješenja za menadžment Kubernetes klastera koji su horizontalno skalabilni (Tanković, et al., 2012).

Cilj istraživanja ovog rada je usporedba relacijskih i dinamičkih modela, te prijenos izrade modela na samog korisnika čime bi se izbjegla nadogradnja sustava i skratio vremenski period implementacije novih procesa.

Ključne riječi: Relacijski modeli, Dinamički modeli, Poslovni procesi, Korisničko modeliranje

Abstract

The development of software solutions relies on the documentation and digitization of existing business processes. Forms used in business processes are modelled by transferring existing form attributes to relational databases. Any change to the solution due to changes in business processes must ensure the consistency and security of existing data and requires a certain period to implement new business processes.

Relational models are traditionally modelled abstraction of the real world to obtain the equivalent of a problem that can be solved programmatically. However, when changing business processes, it is not easy to change them due to the amount of data that relations contain. Data migration to new structures is required, with that data loss is possible due to insufficient planning or simply carelessness of the programmer. The introduction of new relational models, as well as the initial modelling of the process, requires the transfer of end users' domain knowledge to the development engineer, and there can always be errors in communication during analysis, planning and execution of implementation.

Dynamic models are one of possible solutions for this. Dynamic models transfer process modelling to the end user, and the software solution takes care of the data consistency related to the definition of the model (meta model). No migrations are needed, but simply a new definition of the model that will describe the new or improved business process. Following the definition of the model, the process of creating services for meta modelling and data entry using the Helm solution for managing a Kubernetes cluster, which are horizontally scalable, is presented.

The subject of research in this work is the comparison of relational and dynamic models and the transfer of model creation to the end user, which would avoid upgrading the system and shorten the time of implementing new processes.

Keywords: Relational models, Dynamic models, Business processes, User modeling