



UNIVERZITET U NOVOM SADU
FAKULTET TEHNIČKIH NAUKA U
NOVOM SADU



NENAD ZELENVIĆ E5-33-2019

SISTEM ZA UPRAVLJANJE IMOVINOM U ELEKTRODISTRIBUTIVNOJ MREŽI

DOKUMENTACIJA PROJEKTOG ZADATKA
- MASTER AKADEMSKE STUDIJE -

—NOVI SAD, 2020—

SADRŽAJ

1	Uvod	1
2	Arhitektura	2
2.1	Simulator	2
2.2	Scada	2
2.3	Calculation Engine (CE)	2
2.4	NetworkModel Service (NMS).....	2
2.5	UI Adapter	3
2.6	Publish-Subscribe mehanizam (PubSub).....	3
2.7	Transaction Service	3
2.8	Korisnički interfejs (UI)	3
3	Standardi i modeliranje elektroenergetskog sistema	5
3.1	Import podataka na osnovu profila.....	5
3.2	Model podataka	6
3.3	Brisanje elemenata	8
4	Simulacija elektroenergetskog sistema sa kritičnom misijom (SCADA)	9
4.1	Integrity Check	9
5	Napredni računarski sistemi sa kritičnom misijom u elektroenergetskom sistemu (Calculation Engine).....	10
5.1	Alarmi u pozadini	10
5.2	Alarmi manipulacije	11
5.3	Prikaz alarma na UI	12
6	Sigurnost i bezbednost u Smart Grid sistemima	13
6.1	Planiranje remonta na osnovu broja operacija	13
7	Održavanje i kontrola kvaliteta elektroenergetskog softvera (Azure Service Fabric)	15
7.1	Uvod	15
7.2	Calculation Engine – Service Fabric aplikacija	15
7.3	Reliable Collections	16

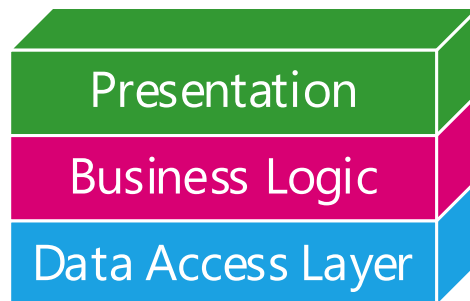
UVOD

Cilj svake uspješne kompanije jeste da smanjenjem troškova poveća svoj profit. To se može uraditi na više načina, ali jedan od najefikasnijih i najbržih jeste korišćenjem softverskog rešenja. Upravo razvoj jedne takve aplikacije, čijim korišćenjem se efikasno mogu smanjiti kapitalni troškovi i jeste cilj projekta.

Aplikacija bi trebalo da obuhvati tok kretanja opreme (prekidači i transformatori) kroz elektroenergetski sistem, a samim tim aplikacija se može koristiti u više timova u okviru jedna firme. Osim toga aplikacija treba da podrži komandovanje nad opremom, praćenje i čuvanje tih akcija, kao i generisanje raznih izveštaja. Čuvanje tih akcija nam omogućava jednu od najbitnijih funkcionalosti aplikacije, a to je predikcija datuma kad je potrebno poslati opremu na popravku. Na ovaj način se unapred može odrediti i isplanirati remot, što značajno smanjuje mogućnost dolaska do havarije i gubitaka napajanja kod velikog dela korisnika.

ARHITEKTURA

Arhitektura projekta je podeljena u tri sloja (Slika 0Error! Reference source not found.)



Slika 0 – Slojevi aplikacije

- Prezantacioni sloj predstavlja korisnički interfejs kroz koji korisnik ima uvid u dešavanja u sistemu i vrši interakciju sa istim
- Biznis logiku obuhvataju servisi: **PubSub**, **CE**, **Scada**, **NMS**, **UI Adapter**. Logika je sakrivena od korisnika i može da bude distribuirana jer je komunikacija između svih servisa preko mreže.
- Sloj podataka je *Azure SQL Server* kome servisi pristupaju koristeći razvijenu biblioteku za komunikaciju sa istorijskom bazom podataka.

Slika 0.1 je slikovita reprezentacija arhitekture celokupnog sistema.

SIMULATOR

Simulator predstavlja *third-party* aplikaciju koja služi za simuliranje vrednosti elemenata u polju. Vrednosti mogu da se zadaju ručno kliktanjem po korisničkom interfejsu ili nekom aplikacijom koja komunicira sa njim i šalje komande. Za komunikaciju se koristi TCP, a format poruke mora da poštuje MODBUS protokol.

SCADA

Scada je servis koji vrši periodičnu akviziciju stanja vrednosti na simulatoru koristeći MODBUS protokol. Kada primi trenutno stanje, šalje se razlika trenutnog i prethodnog stanja na Calculation Engine servis na interpretaciju.

CALCULATION ENGINE (CE)

Servis koji prima podatke sa Scada-e i interpretira njihovo značenje. Dužan je da proveri da li se neki od uređaja nalazi u alarmnom stanju i da zapisuje podatke u Azure SQL bazu podataka (na slici HIST). Za to vreme se pristigli podaci proslede na UI Adapter. Ako se utvrdi da je neki uređaj u alarmnom stanju, dodatne informacije se šalju na UI Adapter da bi se korisnički interfejs informisao o tom stanju. Na zahtev UI Adapter-a generiše izveštaje i šalje ih nazad.

NETWORKMODEL SERVICE (NMS)

Servis koji čuva konektivnost mreže. Podatke o konektivnosti može da primi od **Importer** aplikacije i sa korisničkog interfejsa. Pristigle podatke validira i primenjuje ako su validni, u suprotnom informiše pošiljaoca o tome šta je bila greška u validaciji. Nakon primenjenog modela, ostali servisi mogu po potrebi da šalju zahteve i traže informacije o statičkom modelu.

UI ADAPTER

Specifičnost ovog servisa je u tome da služi kao most između korisničkog interfejsa i ostatka sistema. Korisnički interfejs šalje jedinstvene zahteve i dužnost ovog servisa je da koordiniše zahteve na jedan ili više servisa da bi u potpunosti odgovorio na zahtev korisničkog interfejsa. Tako korisnički interfejs i ostatak sistema nemaju deljene modele podataka.

PUBLISH-SUBSCRIBE MEHANIZAM (PUBSUB)

Da bi moglo da se podrži više od jednog korisničkog interfejsa i da to bude nezavisno od celog sistema koristi se publish-subscribe mehanizam koji će pristigle podatke da demultipleksira na svaki od korisničkih interfejsa tako da svi budu u sinhronizaciji.

TRANSACTION SERVICE

Funkcionalnost ovog servisa je koordinacija distribuirane transakcije kroz sistem. Kada podaci stignu na **NMS** i uspešno prođu validacije proslede se ostalim servisima. Nakon toga se servisi od interesa prijavljuju transakcionom servisu i kada se svi prijave **NMS** obavesti transakcioni servis da distribuirana transakcija može da se pokrene.

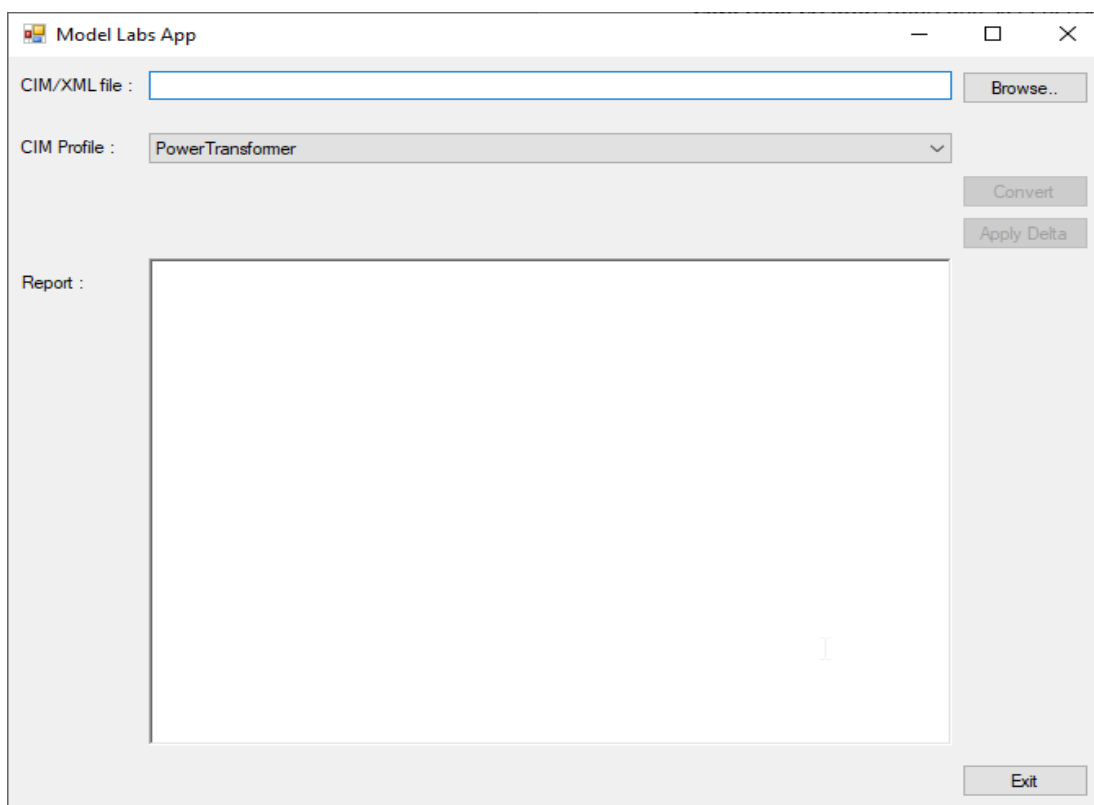
KORISNIČKI INTERFEJS (UI)

Korisnik za interakciju sa sistemom ima na raspolaganju korisnički interfejs. Funkcionalnosti mu omogućavaju manipulaciju uređajima koji se nalaze u sistemu, njihovo prebacivanje u funkciju i van funkcije, prikaz mnoštva izveštaja, popravku.



IMPORT PODATAKA NA OSNOVU PROFILA

Kada dobijemo CIM profil u obliku RDFS fajla, potrebno je taj fajl konvertovati u odgovarajući oblik tako da ga programsko okruženje (Visual Studio) može očitati i da se može integrisati u .NET okruženje. Takav oblik bi bio *dll*. Takođe taj *dll* uvlačimo u projekat kao referencu i time dobijamo pristup svim klasama koje čine osnovni model podataka projekta. Koristimo built-in tool za konverzaciju zvani **CIMProfileCreator**. Uporedno sa generisanjem *dll*-a potrebno je i napraviti odgovarajući *cim xml* koji će odgovarati sa modelom projekta. Nakon uvlačenja *dll*-a i generisanja *cim xml*-a sa odgovarajućim podacima možemo da radimo import podataka u sistem korišćenjem aplikacije **ModelAppLabs** (Slika 0.1.1). Uz pomoć ove aplikacije koristimo generisani *dll* i *cim xml* da bi dobili *ConcreteModel* koji se u toku import-a konvertuje u *Delta* objekat. *Delta* objekat predstavlja osnovu komuniciranja servisa sistema u toku transakcije. U slučaju da nismo uneli vrednosti nekog atributa u *cim xml*-u te vrednosti će biti postavljene sa nekim „default“ vrednostima. U kratkim crtama import podataka kao jedan nabavka opreme i primenjuje se u model svakog servisa kroz distribuiranu transakciju.



Slika 0.1 – ModelAppLabs aplikacija

MODEL PODATAKA

U projektu **Asset Management** se nalazi implementacija sledećih klasa:

Iz paketa *Core*:

- *ConductingEquipment*
- *ConnectivityNodeContainer*
- *Equipment*
- *EquipmentContainer*
- *IdentifiedObject*
- *Maintenance* (konkretna)
- *PowerSystemResource*
- *Substation* (konkretna)
- *Warehouse* (konkretna)

Iz paketa *Meas*:

- *Discrete* (konkretna)
- *Measurement*

Iz paketa *Wires*:

- *Breaker* (konkretna)
- *PowerTransformer* (konkretna)
- *ProtectedSwitch*
- *Switch*

Svaka klasa čije se instanciranje očekuje, odnosno konkretna klasa, dobija odgovarajuću vrednost u *DMSType* enumeraciji (Listing 0.1).

```
1. public enum DMSType : short
2. {
3.     MASK_TYPE = unchecked((short)0xFFFF),
4.
5.     DISCRETE = 0x0001,
6.     BREAKER = 0x0002,
7.     POWERTRANSFORMER = 0x0003,
8.     SUBSTATION = 0x0004,
9.     WAREHOUSE = 0x0005,
10.    MAINTENANCE = 0x0006
11. }
```

Listing 0.1 - Definisanje konkretnih klasa

Svaki tip resursa u modelu se jednoznačno identifikuje odgovarajućim *ModelCode*-om tj. vrednošću od 64 bita. Svaka klasa (apstraktna ili konkretna) dobija svoj *ModelCode* kao i svaki atribut tih klasa. Vrednost *ModelCode*-a nosi u sebi niz informacija vezanih za resurs. Najviših 32 bita opisuje nasleđivanje gde klasa „dete“ uvek ima jednu cifru više od „roditeljske“ klase. Narednih 16 bita opisuje da li je klasa apstraktna ili ne. Ukoliko je klasa apstraktna vrednost je postavljena na nulu, u suprotnom vrednost odgovara vrednosti *DMSType*-a za tu konkretnu klasu. Kada je u pitanju *ModelCode* koji je dodeljen klasi, najnižih 16 bita su uvijek nula. Veći deo

vrednosti *ModelCode*-a koji je dodeljen atributu (viših 48 bita) je identično *ModelCode*-u koji je dodeljen klasi kojoj atribut pripada. Najnižih 16 bita su rezervisani za opis atributa gde viših 8 bita označavaju redni broj atributa u klasi, dok preostalih 8 bita označavaju tip atributa u klasi (int, bool, long, string, List<int>....).

Iako je *mrId* jedinstven za svaki entitet koji se kreira, njegova upotreba za identifikaciju entiteta može da uspori rad servisa jer je u pitanju string. Zbog toga se uvodi novi jedinstveni generator tipa long (poređenje brojeva je daleko brže od poređenja string-ova) tj. *GID*. Globalni identifikator je definisan u klasi *IdentifiedObject* koja se nalazi u hijerarhiji nasljeđivanja svih ostalih klasa. Kreira se na osnovu tri podatka, a to su: *SistemId* (16 bita) koji je za nas uvijek nula jer koristimo jedan sistem, *DMSType* (16 bita) odgovara tipu entiteta za koji se kreira globalni identifikator, brojač (32 bita) za svaki tip entiteta postoji odgovarajući brojač koji obezbeđuje jedinstvenost globalnog identifikatora po tipu. Veza entitet i brojač se čuva u istorijskoj bazi i pre svake transakcije vrši se provera tj. čitanje iz baze te veze da ne dolazi do ponovnog pojavljivanje istog *GID*-a.

Bilo koju klasu i njene attribute iz internog modela prikazujemo u formi resursa pridržavajući se određenih standarda. Standard formuliše upite i odgovore u vidu resursa, *property*-ja i vrednosti. Resurs je svaki objekat koji poseduje jedinstveni identifikator. *Property* je neki aspekt resursa koji se može opisati. Asocijacije između resursa se kreiraju preko *property*-ja koji su tipa *Reference*. *Property* opisuje atribut nekog objekta i sastoji se iz dva dela: *Id* (svaki property je jednoznačno određen preko *ModelCode*-a) i *Value* (sadrži vrednost atributa). *ResourceDescription* opisuje objekat i može da sadrži sve ili samo odabrane *property*-je. Globalni identifikator objekta koji se opisuje predstavlja identifikator *ResourceDescription*-a. Takođe postoji i *Delta* objekat s kojim servisi komuniciraju u toku transakcije. *Delta* u sebi sadrži listu dodatih, ažuriranih i obrisanih *ResourceDescription*-a.

Svaka klasa u modelu se sastoji od privatnih atributa, njihovih javnih *property*-ja, konstruktora koji kao parametar prima globalni identifikator i predefinisanih metoda koje se implementiraju na odgovarajući način.

Metoda *Equals()* proverava jednakost atributa koji pripadaju roditeljskoj klasi i u slučaju potvrdnog odgovora proveravaju se atributi tekuće klase.

Sledeće metode se koriste za manipulaciju atributima klase:

- *HasProperty()* – proverava da li prosleđena vrednost *ModelCode*-a pripada nekom od atributa klase, prvo tekuće, a zatim roditeljske ukoliko ne pripada tekućoj
- *GetProperty()* – služi za konverziju atributa u *Property* objekat
- *SetProperty()* – postavlja vrednosti atributa na osnovu *Property* objekta

Klase koje imaju reference implementiraju metode koje omogućavaju servisu manipulaciju nad referencama:

- *IsReferenced()* – implementira se u slučaju postojanja liste referenci u klasi i ova metoda proverava da li postoji neki entitet koji referencira instancu ove klase
- *AddReference()* - implementira se u slučaju postojanja liste referenci u klasi i koristi se da bi se dodala vrednost u listu referenci
- *RemoveReference()* - implementira se u slučaju postojanja liste referenci u klasi i koristi se da bi se uklonila vrednost iz liste referenci

Klasa *Container* služi za grupisanje entiteta istog tipa i sadrži mapiranja globalnih identifikatora na njihove entitete. Metoda *CreateEntity(long globalId)* obezbeđuje kreiranje konkretnih klasa gde se prvo nađe tip entiteta na osnovu globalnog identifikatora, a zatim se kreira klasa.

BRISANJE ELEMENATA

Na korisničkom interfejsu pruža se mogućnost brisanja elemenata, odnosno trafostanice, laboratorije, skladišta, transformatora i prekidača. Brisanje trafostanica i prekidača je omogućeno samo iz laboratorije, kada su poslani na popravku. Selektovani objekat se mapira u *ResourceDescription*, koji se zatim dodaje u *Deltu* i šalje **NMS**-u da bi se izbrisao iz modela. Prilikom brisanja objekta ukoliko on posjeduje listu referenci u sebi, takođe se briše i svaki element iz te liste.

SIMULACIJA ELEKTROENERGETSKOG SISTEMA SA KRITIČNOM MISIJOM (SCADA)

U sistemu za upravljanje imovinom u elektrodistributivnoj mreži gde se vrše veoma česta paljenja i gašenja opreme (prekidači, transformatori), veoma je bitno da servis koji rukovodi sa ovom operacijom obavlja svoj posao u mili/nano sekundama i da ga obavlja precizno. Ime tog servisa u projektu je **SCADA** i ona mora da zadovoljava sve prethodne kriterijume. Takođe jako bitno stvar je ta da **SCADA** mora biti „glupa“ tj. da obavlja samo tu jednu operaciju (paljenje/gašenje) i sa tom osobinom ona ustvari ni ne zna šta kontroliše već samo radi ono što joj se kaže. S obrizom da ona ne sadrži nikakve podatke o opremi i njihovim stanjima, za to je potrebno da neki drugi servis preuzme tu obavezu. Ta operacija se zove *IntegrityCheck*, a u sledećem poglavlju objasnićemo koji servis to implementira tu operaciju, kako i kada **SCADA** kontaktira taj servis.

INTEGRITY CHECK

Servis koji implementira ovu operaciju je **Calculation Engine** i ova operacija je ustvari jedan od mikroservisa **Calculation Engine-a** (detaljnije u 7. poglavlju). Takođe pošto **Calculation Engine** kontroliše rad *Integrity Check-a* on i čuva stanja opreme tj. znamo da li je prekidač trenutno upaljen ili ugašen. Jedina svrha *Integrity Check-a* jeste da **Calculation Engine** vrati **SCADA** listu svih prekidača sistema i njihova trenutna stanja zatim **SCADA** sa tom listom može da ažurira **Modbus** simulator.

SCADA poziva *Integrity Check* samo jednom, u procesu inicijalizacije tj. pokretanja. Ako provera prođe bez grešaka na konzoli se ispisuje pozitivna poruka i ažurira se **Modbus** simulator sa novim vrednostima. U suprotnom slučaju ispisuje se negativna poruka i ne dolazi do ažuriranja **Modbus** simulator-a. Do negativne poruke može doći usled nekozistentnih baza tj. ako **Calculation Engine** ne sadrži dobre podatke o digitalnim signalima ili ako dođe do prekida veze između ova dva servisa.

NAPREDNI RAČUNARSKI SISTEMI SA KRITIČNOM MISIJOM U ELEKTROENERGETSKOG SISTEMU (CALCULATION ENGINE)

S obzirom da zadatak projekta je upravljanje imovinom u elektrodistributivnoj mreži, kao u takvoj postoji oprema koja se posle nekog vremena kvari (prekidači i transformatori). Za lakše upravljanje tom opremom potrebno je obezbediti adekvatan sistem za alarmiranje tj. upozoravanje korisnika o mogućim kvarovima opreme.

Sistem radi sa dve vrste alarma:

- Alarmi *manipulacije* – Ovakve alarme koriste samo prekidači i predstavljaju vrstu alarma koji se aktiviraju na otvaranje/zatvaranje prekidača. Postoje i podvrste:
 - Alarmi na osnovu kataloške vrednosti limita prekidača
 - Alarmi na osnovu broja kratkih spojeva
 - Alarmi na osnovu prediktovane vrednosti limita prekidača
- *Vremenski* alarmi – Ove alarme koriste i prekidači i transformatori i označavaju vreme kada će oprema da se pokvari. Podvrste:
 - Alarmi na osnovu vremena kojeg provedu na polju
 - Alarmi na osnovu vremena definitivnog kvara tj. nema popravka nakon isteka ovog vremena

U daljem tekstu poglavlja pričaćemo o alarmima manipulacije, načinu konstruisanja alarma u pozadini projekta i o načinu prezentovanja alarma na grafičkom interfejsu.

ALARMI U POZADINI

Način na koju su zamišljeni i konstruisani alarmi je jako slično sa *ModelCode*-vima. Svaki alarm se sastoji od 40 bitne vrednosti, gde svakih 8 bita su rezervisane za svaku podvrstu. Prvih 8 za kratke spojeve, drugih 8 za katalošku vrednost limita, trećih 8 za vreme provedeno u polju, četvrtih 8 za vreme do definitivnog kvara i petih 8 za prediktovanu vrednost limita prekidača. Na svakih tih 8 bita se mogu pojaviti četiri vrednosti koje ukazuju na trenutnu vrednost podvrste tog alarma, a to su:

- *Normal (x0)* - vrednost kada oprema nije u alarmu i nije potrebno obeštavanje korisnika o opasnosti. Svaki put kad kupimo opremu ili je popravimo vrednost alarma te opreme postavimo na *Normal*
- *High (x1)* - vrednost opreme kada je procenat njene iskorišćenosti preko 80% i tada je potrebno obavestiti korisnika o opasnosti
- *HighHigh (x2)* - vrednost opreme kada je procenat njene iskorišćenosti preko 95% i tada je potrebno obavestiti korisnika o opasnosti
- *UsedUp (x3)* - vrednost opreme kada je procenat njene iskorišćenosti preko 100% i tada je potrebno obavestiti korisnika o opasnosti

x promenljiva zavisi od podvrste, ako se radi o kratkim spojevima tu ide 1, ako se radi o kataloškoj vrednosti limita onda ide 2 i tako dalje...

Način sa kojim prelazimo iz jednog stanja alarma u drugo stanje jeste *Bitwise operacije (AND i OR)*. Celokupna logika kreiranja alarma je dovela do toga da za jedan prekidač ili transformator je dovoljna samo jedna vrednost alarma koju s vremenom menjamo, a ne njih više. Tako da npr. je moguće da se desi da je prekidač prešao 80% iskorišćenosti za katalošku vrednost limita i 100% iskorišćenost vremena provedenog u polju i s ovom logikom je omogućeno da se te dve informacije čuvaju u jednoj vrednosti i sa *Bitwise operacija* možemo te vrednosti odrediti.

Pošto je **Calculation Engine** dinamični servis, on ima glavnu ulogu u kontolisanju rada alarma. Kada se desi *High*, *HighHigh* ili *UsedUp* stanje alarma **Calcualtion Engine** vrši promenu vrednosti tog alarma, njegovo čuvanje u dinamičnim kolekcijama, upis u bazu i obaveštavanje klijenata. Klijenti se obaveštavaju tako što **Calculation Engine** pošalje poruku ka **UI Adapter**-u, zatim ta poruka dolazi do **PubSub**-a i od **PubSub**-a ka svim ostalim aktivnim klijentima.

ALARMI MANIPULACIJE


Kao što je ranije rečeno postoje tri podvrste alarma manipulacije i oni se odnose samo na prekidače:

- Alarmi na osnovu kataloške vrednosti limita prekidača se koriste kada dolazi do komandovanja prekidača. Prekidač se može komandovati sa **UI** ili **Modbus** aplikacije. Njegov limit tj. vrednost sa kojom se izračunava procenat iskorišćenosti je kataloška vrednost limita za komandovanje prekidača
- Alarmi na osnovu broja kratkih spojeva se koriste kada dolazi do kratkih spojeva nad prekidačem i dolazi do komandovanja prekidačem ali samo kroz **Modbus** aplikaciju. Njegov limit tj. vrednost sa kojom se izračunava procenat iskorišćenosti je kataloška vrednost limita za kratak spoj. Po nekom pravilo ta vrednost je manja u poredjenju sa kataloškom vrednosti za komandovanje.
- Alarmi na osnovu prediktovane vrednosti limita prekidača se koriste kada dolazi do komandovanja prekidača. Prekidač se može komandovati sa **UI** ili **Modbus** aplikacije. Njegov limit tj. vrednost sa kojom se izračunava procenat iskorišćenosti je prediktovana vrednost limita na osnovu ostalih prekidača koji se nalaze u laboratoriji i dolaze iz istog kataloga. Kad dodajemo novi prekidač u stanicu, limit je jednak kataloškoj vrednosti, a tek nakon određenog vremena odredi se predikovani limit i dodeli se. *Predikcija* se konstantno vrti i taj limit je dinamičan tj. promenljiv

Kada se vrši komandovanje za katalošu vrednost limita tada se računa procenat iskorišćenosti i za katalošku vrednost i za prediktovanu vrednost. Svaki put kad komandujemo sa prekidačem uvećavamo za jedan, i ta vrednost je dinamična. Ona se resetuje svaki put kad kupimo novi prekidač ili neki prekidač popravimo i vratimo nazad u stanicu. Komandovanje sa **UI** aplikacije možemo da vidimo na slikama (Slika 0.1 i Slika 0.2) takođe na Slika 0.3 možemo videti i kako izgledaju limiti na **UI** aplikaciji za katalošku vrednost i za kratki spoj.

[Dashboard](#) [Details](#)

Breaker2

Closed 

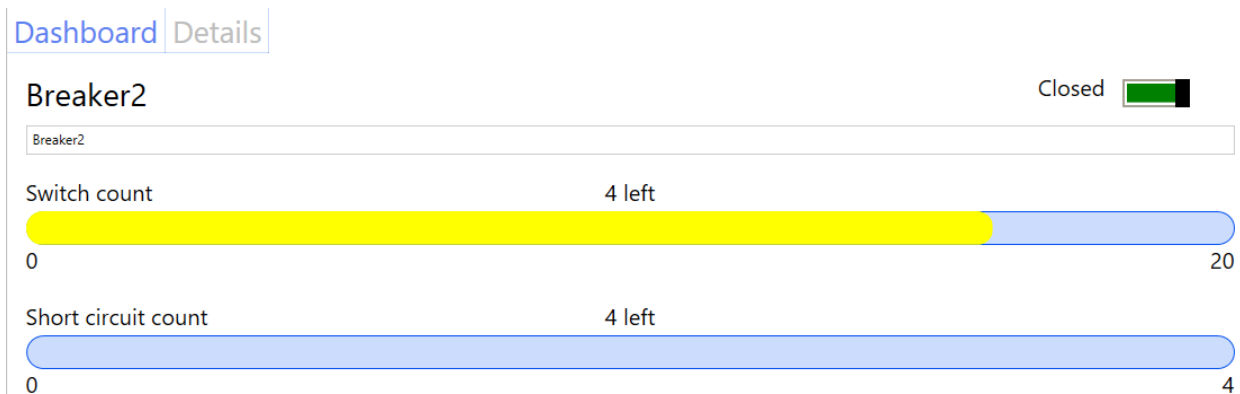
Slika 0.1 - Prekidač u zatvorenom stanju

[Dashboard](#) [Details](#)

Breaker2

Opened 

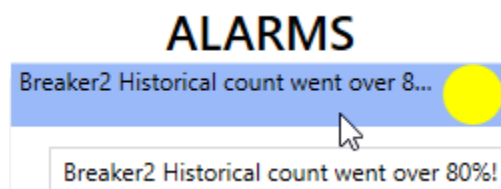
Slika 0.2 - Prekidač u otvorenom stanju



Slika 0.3 - Kataloški limiti i limiti kratkog spoja

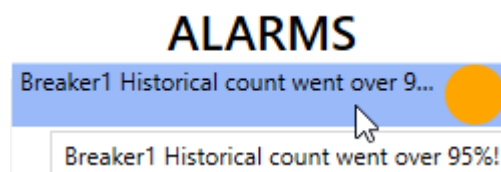
PRIKAZ ALARMA NA UI

Na Slika 0.1 možemo da vidimo izgled prozora kada prekidač uđe u *High* stanje.



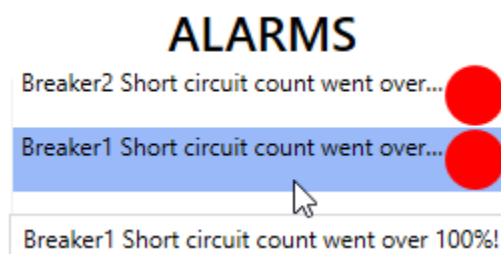
Slika 0.1 - Prikaz High alarm stanja

Na Slika 0.2 možemo da vidimo izgled prozora kada prekidač uđe u *HighHigh* stanje.



Slika 0.2 - Prikaz HighHigh alarm stanja

Na Slika 0.3 možemo da vidimo izgled prozora kada prekidač uđe u *UsedUp* stanje.



Slika 0.3 - Prikaz UsedUp alarm stanja

PLANIRANJE REMONTA NA OSNOVU BROJA OPERACIJA

Planiranje remonta igra važnu ulogu u svakom sistemu, pogotovo kad se radi o sistemu opravljanja elektroenergetke opreme. Efikasnim planiranjem može se poboljšati efikasnost elektroenergetskog sistema. Unapred možemo videti kada koju opremu treba poslati na popravku i na taj način isplanirati popravku, tako da krajnji korisnik najmanje trpi.

Remont na osnovu broja operacija odnosi se isključivo na prekidače i vezan je za *predikciju*. To znači da na osnovu prediktovane vrednosti limita prekidača možemo da odredimo približno vreme prestanka rada nekog prekidača. Na osnovu takvih informacija možemo da zamenimo pokravareni prekidač sa novim i da ga popravimo i sa time izbegavamo nepoželjne havarije i nezgode.

Prikaz ovog plana remonta u projektu prikazan je u vidu izveštaja. Primer jednog ovakvog izveštaja dat je na Slika 0.1.

Osnovni pojmovi na prozoru izveštaja su:

1. Objekat klase **ReportModel** koji se sastoji od sledećih polja
 - GID (globalni identifikator) – jedinstvena vrednost prekidača u celom sistemu
 - Name – naziv prekidača
 - Type – tip
 - Container – naziv kontejnera kome prekidač pripada
 - Price – cena prekidača
 - Catalog limit – kataloški limit prekidača
 - Predicted limit – prediktovani limit prekidača
 - Nominalno vreme isteka rada prekidača
 - Prediktovano vreme isteka rada prekidača
2. Klikom na dugme *Generate* šalje se zahtev na **UI adapter** koji prosleđuje to odgovarajućem servisu
3. Comboboxevi za odabir odgovarajućeg izveštaja i checkboxovi za odabir vremena izveštaja (3 meseca, 1 godina i 5 godina)
4. Klikom na dugme *Export* dobijamo mogućnost eksportovanja izveštaja u .pdf i .csv fajl

UVOD

Azure Service Fabric je distribuirana sistemska platforma koja olakšava pakovanje, raspoređivanje i upravljanje skalabilnim i pozdanim mikroservisima i kontejnerima. Service Fabric hostuje (podiže) mikroservise unutar kontejnera koji su raspoređeni i aktivirani širom klastera.

Ovi distribuirani mikroservisi rade sa velikom gustinom na zajedničkom skupu mašina, koji se nazivaju klasteri. Service Fabric omogućava lagano vreme izvršavanja koje podržava *stateless* i *stateful* mikroservise, što je i glavna podela mikroservisa. Glavna razlika između *stateless* i *stateful* se odgleda u tome kako se podaci čuvaju i njima. U slučaju *stateless* servisa oni se ni ne čuvaju, dok *stateful* čuva podatke. Svaki mikroservis predstavlja jedinstvenu celinu koja je razdvojena od ostalih mikroservisa, može da se skalira, testira i razvija nezavisno od ostalih mikroservisa.

CALCULATION ENGINE – SERVICE FABRIC APLIKACIJA

Calculation Engine predstavlja najkopleksniji servis sistema sa puno obaveza. Kao takav njegova implementacija u mikroservisnom okruženju je i očekivana i poželjna. Pošto servis ima puno obaveza potrebno je bilo odrediti odgovarajući broj mikroservisa koji će te poslove obavljati, a to su 5 *stateful* servisa i 4 *stateless* servisa.

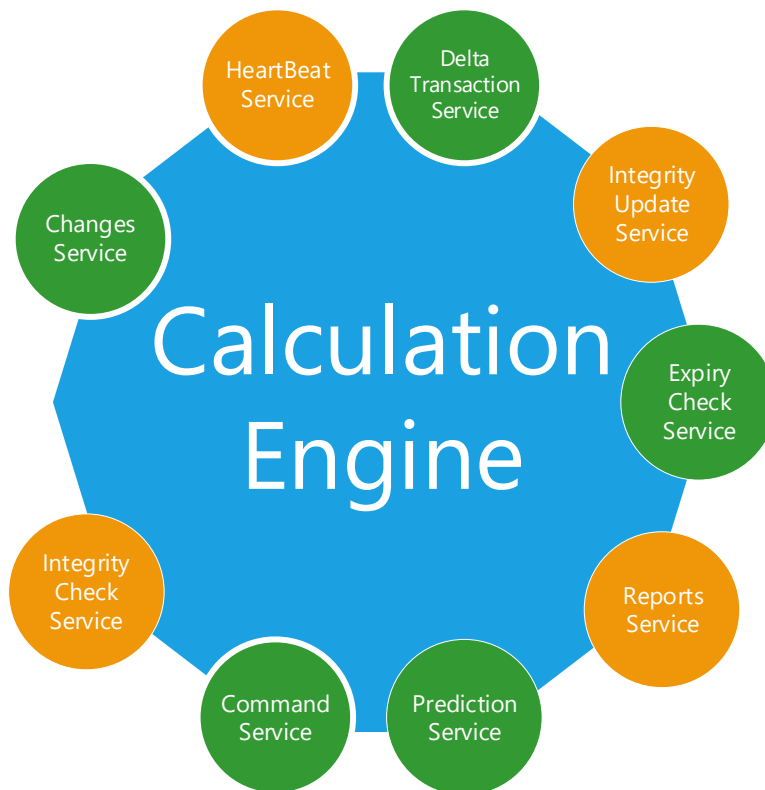
Stateful:

- *DeltaTransaction Service* – Koristi se u toku transakcije i prosledjuje odgovarajuće podatke ka ostalim mikroservisima
- *Changes Service* – Servis koji očitava promene sa **SCADE**
- *Expiry Check Service* – Provera vreme isteka opreme u stanicama
- *Prediction Service* – Servis koji služi za računanje predikcije
- *Command Service* – Služi za obaveštavanje **SCADE** u slučaju komandovanja sa **UI** aplikacije

Stateless:

- *HeartBeat Service* – Provera stanje Calculation Engine, da li je živ ili mrtav
- *IntegrityUpdate Service* – Vraća podatke koji se traže sa **UI** aplikacije
- *IntegrityCheck Service* – Proverava stanje digitalnih signala vezanih za prekidače
- *Reports Service* – Služi za vraćanje podataka potrebnih za izveštaje

Na Slika 0.1 možemo da vidimo kako **Calculation Engine** izgleda kada je razdvojen na mikroservise.



Slika 0.1 - Calculation Engine kao mikroservisna aplikacija

RELIABLE COLLECTIONS

Kao što je ranije rečeno *Stateful* servisi čuvaju svoja stanja unutar *Service Fabric* klastera. Sačuvana stanja se dele između svih čvorova u particiji, a njihova promena se replicira sa primarnog čvora na sve sekundarne čvorove automatski u okviru transakcija. Sva stanja se čuvaju u strukturama koje se nazivaju *Reliable Collections*. U projektu je korišćena *Reliable Dictionary* kolekcija. Zbog razvoja **Calculation Engine** kao *Service Fabric* aplikaciju i korišćenja mikroservisa prinudno je bilo prelazak na *Reliable Collections* sa običnih kolekcija (List, Dictionary...).

Prednosti *Reliable Collections* u odnosu na druge kolekcije su:

- *Replikacija* – promena stanja se repliciraju i time se postiže velika dostupnost.
- *Asinhroni API* – API-ji su asinhroni tako da se osigurava da thread-ovi ne budu blokirani prilikom IO operacija.
- *Transakcija* – Lako se može upravljati sa više *Reliable* kolekcija unutar servisa.
- *Postojanost* – Podaci se čuvaju na disku radi trajnosti čime se takođe i sprečavaju kvarovi npr. nestanak struje datacentra.

Reliable Collections nam omogućavaju pisanje visoko dostupnih, skalabilni i relativno tačnih cloud aplikacija. Ključna razlika između *Reliable Collections* i ostalih tehnologija visoke dostupnosti je ta što se stanja tj. Podaci čuvaju lokalno u servisnoj instanci i njihova dostupnost je na visokom nivou. *Reliable Collections* čuvaju svoja stanja sve kod je klaster aktivan. Na Slika 0.1 možemo videti implementaciju *Reliable Dictionary* u okviru *Command Servisa*.

U slučaju da nekom mikroservisu su potrebni podaci drugog mikroservisa, on šalje upit tj. uspostavlja se WCF veza između servisa i podaci se razmenjuju.

```
1. private readonly ReliableDictionary<long, CommandBreaker> breakers;  
2.  
3. private readonly ReliableDictionary<long, Discrete> discretetes;  
4.  
5. public CalculationModel(IReliableStateManager stateManager)  
6. {  
7.     this.breakers = new ReliableDictionary<long, CommandBreaker>(stateManager, "command_Breakers");  
8.     this.discretetes = new ReliableDictionary<long, Discrete>(stateManager, "command_Discretetes");  
9. }
```

Slika 0.1 - Implementacija Reliable Dictionary

Prilikom primene Delte, kad stigne Delta na **Calculation Engine** preko distribuirane transakcije ti podaci u okviru Delte koji stignu na *stateful* servise koji se nalaze na **Calculation Engine** se upisuju u *Reliable Collections* koje se nalaze tamo. Takođe svaki put kad dođe do dinamičnih promena (alarmi, manipulacija prekidača, predikcija) te promene i vrednosti vezane za njih se isto upisuju u *Reliable Collections*. *Reliable Collections* čuvaju te podatke sve dok se ne ugasi *Service Fabric* klaster, sa gašenjem klastera gube se podaci i stanja.