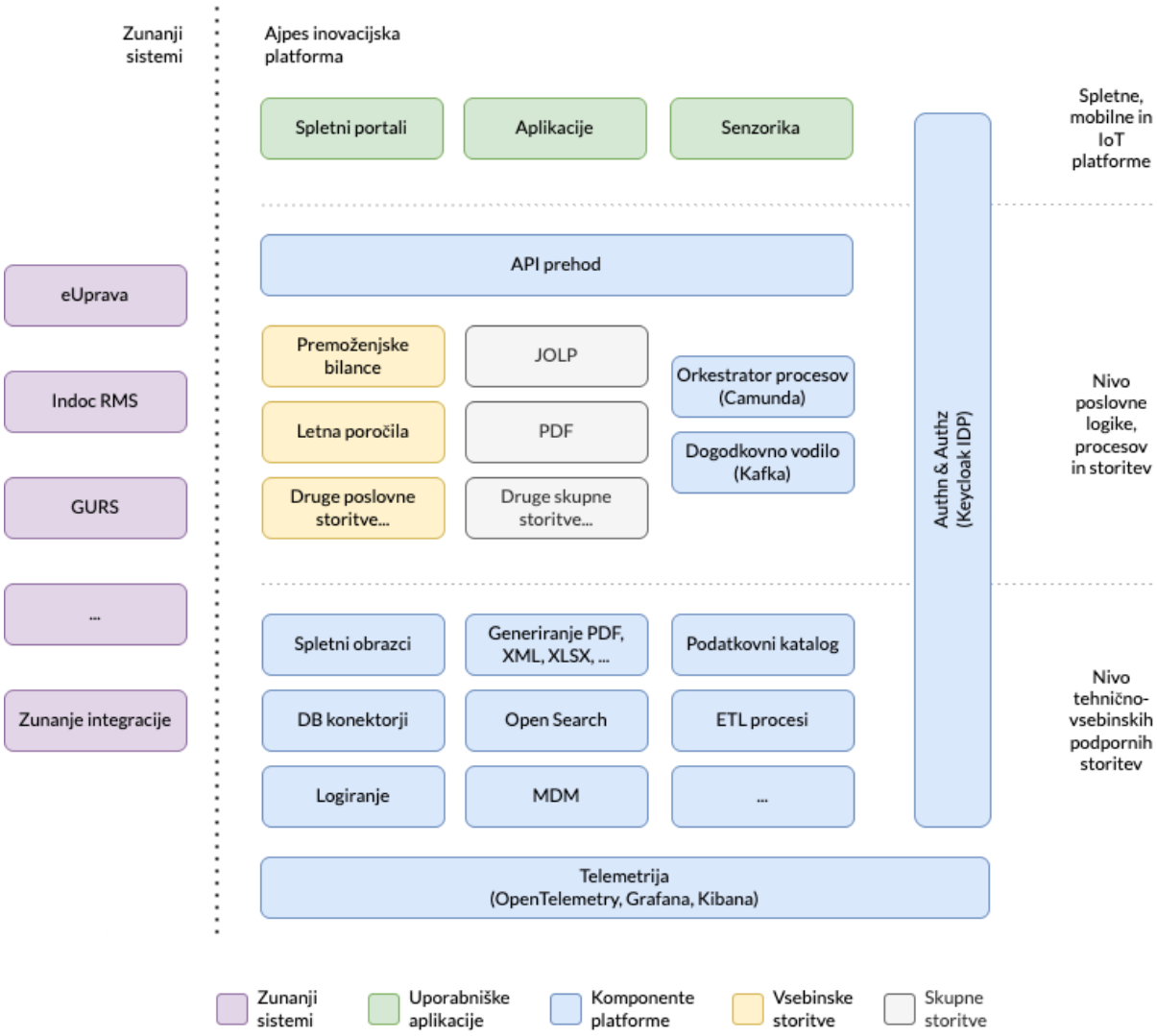
Tehnična arhitektura platforme

Na spodnji shemi je predstavljena tehnična arhitektura AJPES inovacijske platforme, na kateri se implementirajo rešitve za podporo poslovnim procesom.



Slika 1- arhitektura platforme

V nadaljevanju izpostavimo ključne lastnosti arhitekture, ki predstavljajo tudi osnovne zahteve za inženirje programske opreme.

Poslovna logika razklopljena od uporabniških vmesnikov

Nivo uporabniških vmesnikov (morebiti tudi senzorskih naprav) je ločen od pravil poslovne logike. Tehnično to pomeni, da so vmesniki za klice programskih funkcij, ki izvajajo operacije nad podatki, primarno dosegljivi preko HTTPS protokola (headless). Homogene programske funkcije so združene v vsebinske ali skupne poslovne storitve (mikrostoritve).

Dostop do poslovnih storitev je omogočen izključno preko API prehoda, ki primarno izvaja osnovne varnostne preveritve in služi kot fasada pred tehničnimi podrobnostmi implementacije in lokacije zalednih storitev. Enako velja tudi za klice med storitvami (kjer je to nujno potrebno in ni mogoče razklopiti sistemov s pomočjo dogodkov).

Enotna avtentikacija in avtorizacija

Vsi moduli in aplikacije, postavljene na platformi, uporabljajo enoten sistem za dodeljevanje in upravljanje identitet (IDP) Keycloak. Konkretno se uporablja standardni protokol Oauth 2.0, po katerem aplikacije kot del glave zahtev pošiljajo žetone (JWT), ki predstavljajo identiteto in pooblastila uporabnika ali sistema, ki je žeton pridobil. Žeton je podpisan s privatnim ključem IDP sistema, aplikacije pa veljavnost potrdijo z javnim ključem, katerega pridobijo od IDP sistema. Sam opisan protokol je že dobro poznan in podprt v knjižnicah za različne programske jezike – uporaba teh je obvezna.

Uporabniški vmesniki uporabljajo *authorization code grant* protokol za pridobivanje žetonov. Tehnično to pomeni, da je uporabnik preusmerjen na avtentikacijski proces znotraj Keycloak-a, ta pa je dinamično nastavljiv (recimo vklop 2FA, social login, federacija na MS Entra ID, ...). Po opravljeni avtentikaciji je uporabnik preusmerjen nazaj v izvorno aplikacijo s kodo, katero aplikacija izmenja za dostopne žetone, ki predstavljajo uporabnika. Uporabniški vmesniki po določenem času neaktivnosti (glede na kritičnost informacij) uporabnika avtomatsko odjavijo.

Zaledne storitve uporabljajo *client credentials grant* protokol za pridobivanje žetonov. Tehnično to pomeni, da je vsaki aplikaciji dodeljen *client id* in *client secret*. Zaledni sistem s parom teh poverilnic pridobi dostopne žetone v svojem imenu (t.i. *service account*).

Aplikacije pridobivajo žetone s pravicami v najmanjšem obsegu, potrebnem za svoje delovanje (princip omejevanja pravic).

Arhitektura prav tako vzpostavlja oz. predvideva t.i. okolje brez zaupanja (trust-less environment), kar v praksi pomeni, da vsak servis preverja vsako vhodno zahtevo (prisotnost veljavnih poverilnic in prisotnost zadostnih dovoljenj za izvedbo akcije). To velja tudi za klice znotraj notranjih storitev ter v okviru zasebnih in izoliranih omrežij.

Mikrostoritvena arhitektura

Mikrostoritev predstavlja sklop sorodnih funkcionalnosti, ki rešujejo določen vsebinski problem. Obseg storitve je lahko ozek na nivoju posamezne funkcije (npr. pretvorba valut po trenutnem tečaju) ali pa širši na nivoju sistema (npr. sistem za upravljanje matičnih podatkov organizacije).

Funkcionalnosti mikrostoritev so izpostavljene preko HTTPS ali GRPC protokolov (po potrebi lahko tudi drugih), uporabniški vmesniki pa so implementirani ločeno ter služijo kot prožilci poslovnih funkcij s strani uporabnikov ali drugih sistemov.

Posamezne storitve primarno komunicirajo s širšim sistemom tako, da objavijo dogodek na dogodkovnem vodilu Apache Kafka. S tem znižujemo stopnjo sklopljenosti sistema. V primeru dileme, ko se je težko odločiti ali je klic res potreben, je potrebno razmisliti o ustreznosti postavljenih mej storitev. Včasih je potrebno dve storitvi združiti, po navadi takrat, ko se naše razumevanje problematike poglobi.

Storitve v grobem ločimo v dve kategoriji:

- nivo poslovne logike, procesov in storitev,
- nivo tehnično-vsebinskih podpornih storitev.

Storitve poslovne logike izvajajo primarne (jedrne) poslovne funkcije, ki predstavljajo delovanje organizacije (npr. upravljanje letnih finančnih ali premoženjskih poročil). Te storitve dalje delimo na vsebinske in skupne storitve s ciljem optimizacije razvojnih sredstev in poenostavitve arhitekture. Razvijalci so dolžni tekom razvojnega procesa refaktorirati sistem skladno z arhitekturo (npr. prepoznati priložnost za uporabo ali uvedbo skupne storitve).

Tehnično-vsebinske podporne storitve omogočajo izvajanje splošnih funkcionalnosti, ki se lahko uporabijo v različnih domenah (npr. upravljanje z matičnim katalogom storitev ali skupnimi šifranti). Nabor teh storitev je na voljo na platformi za vse vsebinske storitve. Nekatere od teh imajo tudi pripadajoče bogate uporabniške vmesnike in tako delujejo že kot zaključene celote. Cilj razvijalcev je skladno z optimizacijo stroškov razvoja širiti nabor teh storitev in poskušati vsebinske težave reševati z že znanimi orodji in tehnologijami (t.j. tehnično-vsebinskimi podpornimi storitvami).

Nabor nekaj tipičnih podpornih storitev:

- dinamični spletni obrazci (upravljanje obrazcev in enoten zajem podatkov),
- generiranje standardnih dokumentov (PDF, XLSX, Word, ...),
- hitro iskanje (po nestrukturirani vsebini),
- upravljanje z matičnimi podatki (Master Data Management),
- podatkovni katalog (nabor podatkovnih produktov).

Dogodkovno-vodena arhitektura

Posamezne storitve za komunikacijo s širšim sistemom primarno uporabljajo dogodke. Dogodek (oz. poslovni dogodek) je opis dejstva, ki se je zgodilo in je nespremenljivo. Dogodke poimenujemo v pretekliku, po navadi se navezujejo na neko entiteto.

Nekaj primerov:

- naročilo oddano (order submitted),
- obrazec oddan (form submitted),
- prejeto plačilo (payment received),

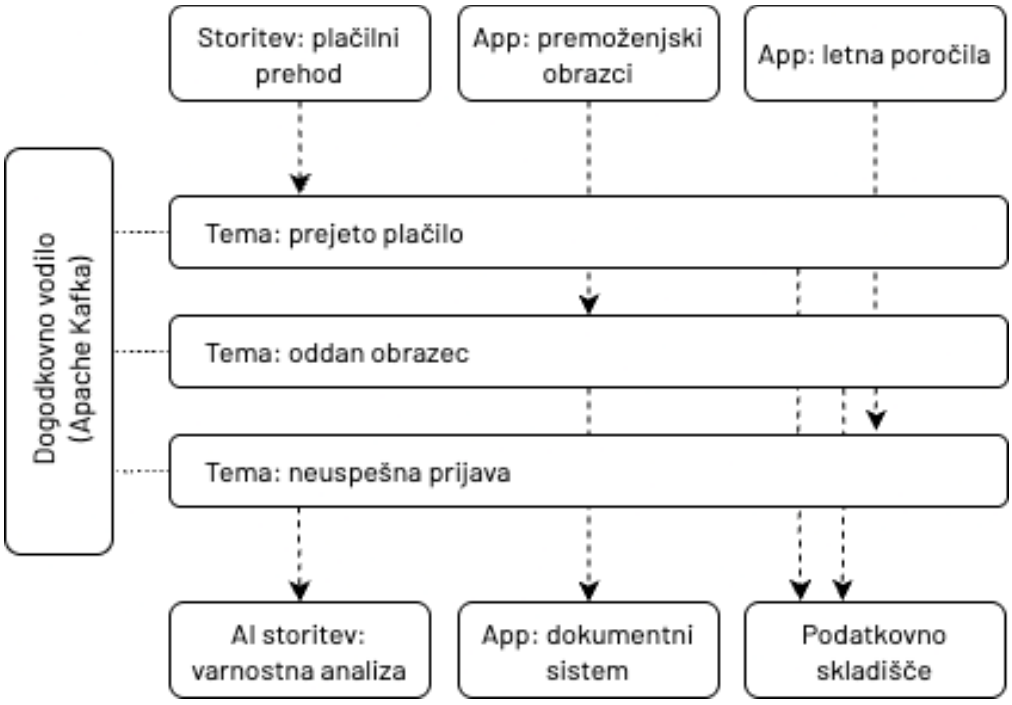
- neuspešna prijava (login failed).

Zahteva po opisu poslovnih dogodkov vodi inženirje sistemov, da že pri zasnovi poskušajo ugotoviti, kaj so tisti poslovni učinki, katere njihovi sistemi producirajo. S tem se viša nivo ujemanja tehnične in vsebinske perspektive na poslovni sistem. Iz tega sledi, da je tudi nadaljnja komunikacija in vzdrževanje bolj enostavno.

Vsebina dogodkov je teoretično lahko poljubna, v praksi pa v dogodke najbolj pogosto zapisujemo reference na entitete, ki so z dogodkom povezane (npr. ID osebe namesto njenih podatkov). Dogodki se objavljajo na dogodkovnem vodilu Apache Kafka, v posameznih t.i. temah (topicih). V posamezno temo praviloma piše samo en proizvajalec (producer), bere pa lahko en ali več konzumentov (consumer).

Uporaba dogodkovne arhitekture znižuje sklopljenost sistema in omogoča večji nadzor nad pretokom podatkov. Prav tako nam omogoča vrsto drugih prednosti glede razširljivosti in skalabilnosti sistema.

Ključno je, da na storitve gledamo kot oddajnike podatkovnih tokov (npr. tok plačil, oddanih obrazcev, ...). Te podatkovne tokove lahko nato spremljamo in obdelujemo neodvisno od izvora (ki, kot rečeno, proizvaja nespremenljiva poslovna dejstva). Takšen razmislek nas vodi do implementacije živih/dinamičnih sistemov, saj zaradi nizke sklopljenosti dopušča hitro integracijo in prilagoditve brez kritičnega vpliva na operativne sisteme.



Slika 2 - shema dogodkovne arhitekture

Na shemi lahko vidimo primer dogodkovno-vodene arhitekture, z vidika podatkovnih tokov. Puščice nakazujejo tok podatkov, tehnično konzumenti spodaj neprestano poslušajo (so naročeni) na nove dogodke (native Kafka protokol) – pull metoda.

Izpostavimo še, da je lahko posamezna storitev naročena na več tem, kot na zgornjem primeru podatkovno skladišče. S tem odpremo velik nabor morebitnih uporab takšne arhitekture z vidika obdelovanja podatkov (tudi v realnem času).

Orkestrator procesov

Sistemi na platformi podpirajo različne poslovne procese z izvajanjem funkcij poslovne logike. Za potrebe tega dokumenta proces opredelimo kot zaporedje korakov, ki se običajno izvaja v daljšem časovnem obdobju in vključuje enega ali več sistemskih in/ali človeških deležnikov.

Z vidika tehnične izvedbe to pomeni, da delegiramo odločitve o tem, kaj je naslednji korak v procesu na orkestratorja procesov Camunda. Proces je modeliran v standardu BPMN in nameščen v bazo orkestratorja. Ta skrbi za hrambo trenutnega stanja posameznih instanc procesov (kje v procesu smo) kar pomeni, da bistveno znižamo kompleksnost poslovnih rešitev. V storitvah posledično ni več implementiranih procesnih odločitev, temveč zgolj validacije in izračuni poslovnih dejstev ter hranjenje rezultatov v trajne hrambe (npr. podatkovne baze).

Orkestrator procesov Camunda podpira večino elementov, predpisanih v standardu BPMN, ter omogoča modeliranje in izvajanje kompleksnih procesnih diagramov. Za modeliranje se uporablja Camunda Modeler (odprtokodna namizna aplikacija).

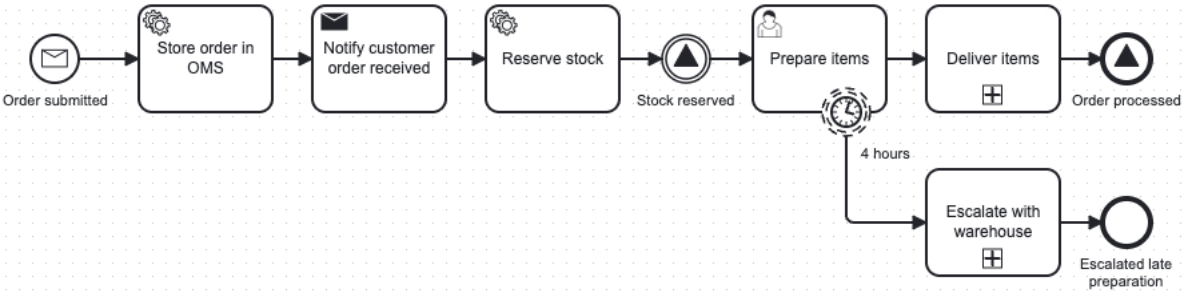
Procesi so tipično sestavljeni iz:

- sistemskih opravil in
- človeških opravil.

Orkestrator procesov za posamezno instanco procesa hrani spremenljivke, katere lahko skozi življenjski tok procesa dodajamo in spreminjamo. Vrednosti spremenljivk orkestrator uporabi pri odločanju v vejitvah (pogojnih stavkih), uporabljamo pa jih lahko pri implementaciji opravil.

Orkestrator objavi na voljo opravila, ki jih je potrebno obdelati (podobno kot v primeru Apache Kafka, kjer se objavijo na temah), zunanje storitve pa ta opravila prevzemajo, obdelajo in orkestratorju vrnejo rezultat v obliki nabora novih/posodobljenih spremenljivk.

Do funkcionalnosti orkestratorja se dostopa preko REST API-ja, ki ponuja možnosti za prevzem in zaključevanje nalog, pridobivanje informacij in statistike o procesih, migracije instanc procesov iz starih v nove verzije, ...

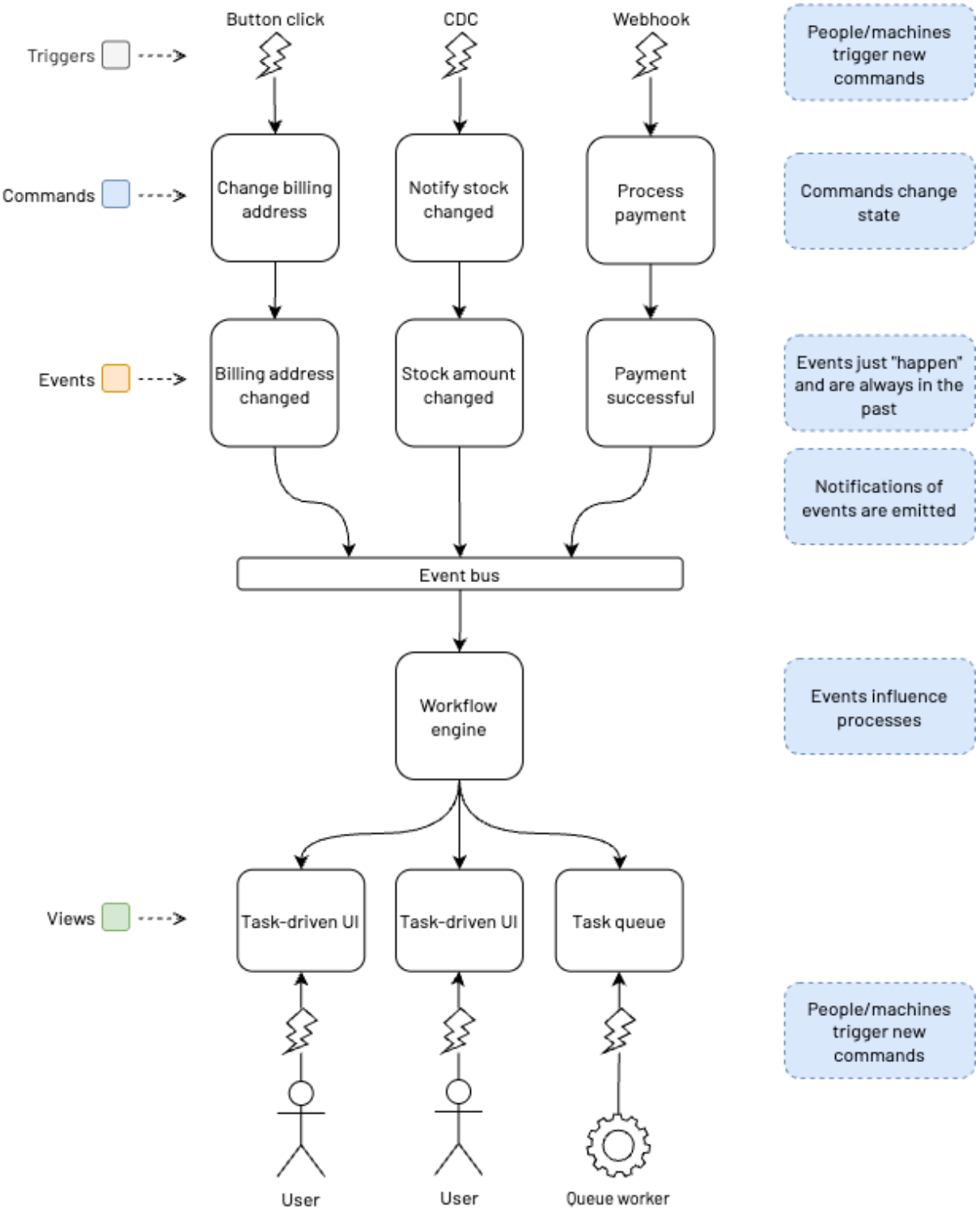


Slika 3 - primer procesa v BPMN

V zgornjem primeru je prikazan model preprostega procesa za naročilo. Opis delovanja nekaj glavnih elementov, ki predstavljajo tudi zahteve za tehnične karakteristike poslovnih storitev:

- proces se začne s sporočilom *Order submitted*, katerega lahko proži poljubni zunanji sistem (sporočilo vsebuje tudi spremenljivke, na primer podatke o naročilu);
- orkestrator ob prejemu sporočila prepozna, da je naslednji korak sistemsko opravilo *Store order in OMS*, zato posodobi stanje procesa v bazi (s tem tudi objavi na razpolago sistemsko opravilo);
- sistemsko opravilo prevzame zunanja poslovna storitev, zadolžena za integracijo s sistemom OMS – podrobnosti orkestratorja ne zanimajo, storitev po tem, ko izvede poslovno logiko, vrne rezultat v obliki spremenljivk;
- orkestrator nato na podlagi novih spremenljivk preveri, kaj bo naslednji korak, ter instanco procesa premakne naprej; glede na vrsto naslednje aktivnosti (bodisi sistemsko ali človeško opravilo) je odvisno, kateri sistem bo zaključil opravilo;
- ko proces preide do aktivnosti *Prepare items* (ki je človeško opravilo), se podrobnosti le-tega lahko prevzamejo preko API vmesnika orkestratorja in prikažejo bodisi v namenskih aplikacijah, bodisi v generičnih seznamih opravil;
- ko uporabniki preko uporabniških vmesnikov opravijo delo, poslovne storitve oddajo dogodke, ki prožijo spremembe v procesu.

Razširjen primer uporabe konceptov dogodkovnega vodila in orkestratorja procesov je prikazan na spodnji shemi.



Slika 4 - koncept uporabe orkestratorja in dogodkov

Dodatne usmeritve za razvijalce

Glede na opisano inovacijsko platformo AJPES bodo razvijalci pri razvoju IT rešitev sledili ciljem modernih poslovnih sistemov:

- privzeta varnost v vseh pogledih,
- vrhunska uporabniška izkušnja (tudi za notranje zaposlene),
- osredotočenost na proces,
- dolgoročno vzdržen razvoj,
- splošne dobre prakse modernega razvoja programske opreme.

V nadaljevanju opišemo lastnosti sistemov, katere morajo le-ti izkazovati, da ustrezajo pričakovanim standardom in višajo možnosti za dosego prej omenjenih ciljev. Za izvedbo sistemov na način, da bodo izkazane navedene lastnosti, so odgovorni razvijalci in inženirji programske opreme.

Sledenje UX smernicam

Inženirji v sodelovanju z eksperti za oblikovanje vzdržujejo sistem načrtovanja in oblikovanja uporabniške izkušnje (design system). Razvijalci implementirajo uporabniške vmesnike skladno z usmeritvami in sredstvi, ki jih določa design system.

Cilj sistema je zagotoviti enotno uporabniško izkušnjo skozi celotno organizacijo – to vključuje tako vmesnike programske opreme, kot tudi dokumentacijo, navodila, itd.

Sistem je v praksi vzpostavljen kot repozitorij:

- semantičnega modela entitet z opisom atributov,
- modela akcij nad entitetami, ki predstavljajo operacije v aplikacijah,
- barvne in oblikovne sheme organizacije (CGP, marketing usmeritve, ...),
- gradnikov uporabniškega vmesnika (gumbi, vnosna polja, ...),
- komponent, ki so vizualizacija domenskih entitet in akcij nad njimi,
- tipičnih ekranov za implementacijo vmesnikov.

Razvojni proces razvijalcev je prilagojen sprotnemu vzdrževanju (oz. vzpostavitvi) design sistema. Ključno za razumevanje namena sistema je, da stremimo k cilju vzpostavitvi ponovljivih uporabniških tokov nad dobro-definiranimi domenskimi entitetami in akcijami. Hkrati se zavedamo, da je uporabniški vmesnik z vidika obdelovanja informacij zgolj implementacijska podrobnost - iste akcije nad neko entiteto lahko na primer poganja človek preko uporabniškega vmesnika ali pa AI agent. Ne glede na modaliteto se zaledni konceptualni model ne sme spreminjati – lahko pa se spreminja uporabniška izkušnja za izvedbo željenih ciljev.

Za tehnično izvedbo in repozitorij se uporablja kombinacija orodja Figma (ali primerljivega) ter centralnega vira dokumentacije entitet, akcij in uporabniških tokov (npr. Confluence, GitLab).

Uporaba ogrodij za uporabniške vmesnike

Razvijalci uporabniške vmesnike implementirajo s pomočjo odprtokodnih JavaScript ogrodij in ne implementirajo svojih rešitev. Primarno se uporablja ogrodje PrimeVue, s prilagojenimi osnovnimi komponentami na oblikovno shemo organizacije.

Uporabniški vmesniki so implementirani kot SPA aplikacije, ki same ne vsebujejo poslovne logike, z zalednimi sistemi pa komunicirajo preko HTTPS protokola (AJAX klici). Praviloma so vsi klici usmerjeni na API prehod, za lažje obvladovanje in organizacijo.

Aplikacije se neposredno ne ukvarjajo z avtentikacijo uporabnikov, temveč delegirajo to funkcionalnost na Keycloak. Ko se uporabnik uspešno identificira, ga Keycloak preusmeri nazaj v aplikacijo s kodo, katero aplikacija pri Keycloaku izmenja za dostopni žeton. Ta žeton aplikacija nato pošilja z vsako zahtevo na zaledne storitve.

Keycloak prav tako podpira enotno prijavo (SSO), kar pomeni, da aplikacija ob »zagonu« preveri, ali v brskalniku že obstaja SSO piškotek. Če obstaja, uporabniku ni potrebno ponovno izkazovati identitete.

Tehnično so vsi ti postopki že podprti v osnovni JavaScript knjižnici, katero vzdržuje Keycloak in v adaptacijah za različna ogrodja (tudi Vue). Razvijalci morajo tako zgolj vključiti delovanje v svoje implementacije.

Avtomatsko testiranje

Za razvoj aplikacij se uporablja pragmatičen TDD (Test-Driven Development).

Ne zasledujemo 100% pokritosti s testi, temveč pokritost tistih delov sistema, ki so kritični in kjer pričakujemo večje število sprememb (po navadi se to kaže v približno 85% pokritosti s testi). Testiranje uporabljamo kot orodje za pisanje boljše kode, pripravljene na dolgoročni razvoj.

Za kritične dele sistemov se izvede tudi obremenitveno testiranje (load testing) s pomočjo orodij kot so K6, da se že tekom razvoja zazna ozka grla in ne-optimalne sekcije kode in/ali poizvedb.

Za posamezne module razvijalci pripravijo poročilo o testiranju v jUnit formatu - na razpolago so odprtokodne knjižnice, ki se preprosto vključijo v projekte in generirajo poročila. Poročilo izpostavi najmanj metrike o celotni pokritosti kode s testi ter pokritosti različnih poti v kodi (izraženo v odstotkih).

Samo izvajanje testov, pripravo poročil in upoštevanje metrik se pozneje tudi vključi v CI/CD postopke.

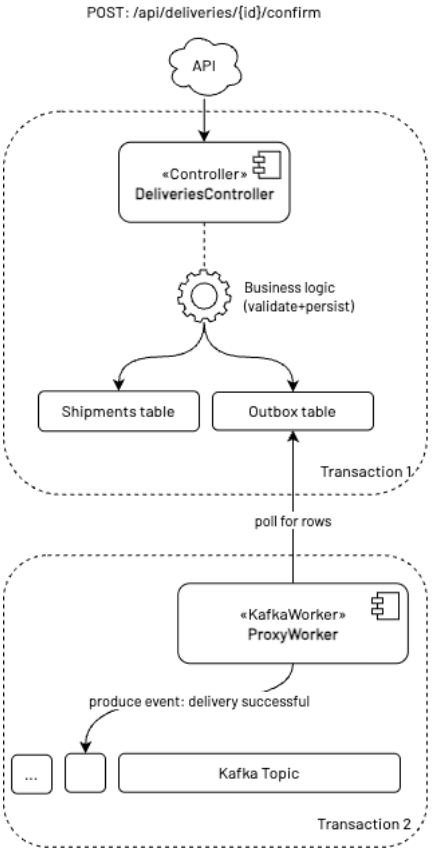
Strukturirani in zanesljivi dogodki

Aplikacije objavljajo poslovne dogodke na dogodkovno vodilo Apache Kafka.
Shema dogodkov sledi Cloud Events specifikaciji, ki predpisuje nabor obveznih in poljubnih polj. S tem dosežemo standardno shemo za izmenjavo podatkov skozi celotno platformo. Tehnično to pomeni, da v glavo Kafka sporočil dodamo polja predpisana s strani Cloud Events, vsebina pa ostane enaka.

Serializacija sporočil pred pošiljanjem v Kafko je bodisi v tekstovnem JSON formatu, bodisi v binarnem AVRO formatu. Kjer se pričakuje večja količina podatkov, je priporočen AVRO zaradi bolj optimalne hrambe na disku.

Ne glede na serializacijski format, aplikacije uporabljajo register shem (Confluent Schema Registry), ki hrani repozitorij shem, vezanih na posamezne Kafka teme. Aplikacije se povežejo na register shem in pred serializacijo skladno s shemo validirajo sporočilo in ga ne zapišejo v Kafko, v primeru, da so podatki pomanjkljivi.

Aplikacije skrbijo za zanesljivost pošiljanja dogodkov – vsaka aplikacija zagotovi, da se bo dogodek v Kafko poslal najmanj enkrat. Tehnično se za izvedbo uporabi »outbox« vzorec, shematsko prikazan v nadaljevanju.



Slika 5 - outbox vzorec

Iz zagotovila, da bo vsak zapis v Kafko poslan najmanj enkrat, prav tako sledi, da morajo biti aplikacije, ki poslušajo sporočila, biti odporne na podvojena sporočila (od vsebine je odvisno, kakšna strategija se uporabi za obravnavo takšnih sporočil).

Dolgoročna učinkovitost

Aplikacije upoštevajo dolgoročnost uporabe, predvsem z vidika velikosti zbranih podatkov in hitrosti delovanja.

Pričakovano je, da se izvajanje vseh funkcij (razen vnaprej dogovorjenih izjemah) končajo v manj kot 100ms, merjeno neposredno na strežniku, kjer se funkcija izvaja (praviloma je ta čas veliko nižji v razponu 20-50ms). Omrežni dostopi in motnje z vidika končnega uporabnika te čase po navadi še povečajo za cca. 50ms (torej čas prenosa rezultata preko omrežja do naprave uporabnika).

Ti performančni nivoji morajo biti doseženi tudi po dolgoročni uporabi aplikacij. V praksi to pomeni, da razvijalci predhodno razmišljajo o ustreznih indeksih v primeru relacijskih podatkovnih baz. V primerih, ko je predvideno iskanje po nestrukturirani vsebini, se je potrebno opredeliti do ustreznosti različnih tipov indeksov v relacijskih bazah ali pa se že na začetku uporabi Open Search, ki je na voljo na platformi. V primeru uporabe Open Search se prav tako razmisli o optimalnih zapisih, ustreznem načinu indeksiranja in optimalno zapisanih poizvedbah (uporaba Open Search sama po sebi ne zagotavlja hitrosti).

Sistemi zagotavljajo tudi avtomatsko arhiviranje (umik) podatkov iz transakcijskih tabel v primerih, ko s časom in dolgoročno uporabo lahko količina podatkov vpliva na performance poizvedb. Takšno arhiviranje je sistemsko in se izvaja avtomatsko na določeno periodo.

Dokumentacija

Izvajalci posameznih sistemov vodijo najmanj ažurno dokumentacijo spletnih vmesnikov storitev, v formatu Open API 2.0 (Swagger). Dokumentacija se praviloma generira avtomatsko iz trenutno aktivne verzije kode (bodisi ad-hoc, bodisi tekom CI/CD postopka) in je dosegljiva razvijalcem, ki se želijo povezati s storitvijo. Iz varnostnih vidikov mora dokumentacija v produkcijskem okolju skrita za avtorizacijo.

Izvajalci prav tako vodijo procesno dokumentacijo v formatu BPMN, neodvisno od tega, ali je uporabljen orkestrator procesov ali ne. V praksi se takšna dokumentacija pripravi že tekom načrtovanja skupaj s poslovnimi uporabniki. V primeru uporabe Camunde pa so BPMN modeli del repozitorija kode, saj se preko CI/CD postopkov tudi nameščajo.

Varnost

Pri razvoju aplikacij se upošteva najmanj trenutno aktualni seznam OWASP Top 10 ranljivosti, katere se sistematično naslovi in preprečuje. Sorodno temu se prav tako upošteva OWASP priporočila za beleženje revizijske sledi in dnevniških datotek. Tekom samega razvojnega procesa se uporabljajo tudi statični analizatorji kode, ki že tekom programiranja opozarjajo na morebitne nevarnosti.

Enoten nadzor in telemetrija

Aplikacije omogočijo, da je njihovo interno delovanje možno spremljati z orodji za analizo in varnost, kot so Grafana, Kibana in Sentry. Tehnično to pomeni, da aplikacije uporabijo standard Open Telemetry in pripadajoče knjižnice.

V dogovoru z upravljalcem platforme se telemetrija bodisi aktivno pošilja v centralno točko, bodisi se izpostavi v obliki Prometheus metrik. Aplikacije prav tako hkrati s standardnimi operativnimi metrikami (npr. poraba pomnilnika in procesorja) po potrebi izpostavijo tudi vsebinske metrike, ki so lahko uporabne na analitičnih nadzornih ploščah (npr. število oddanih poročil).

Neodvisnost od platforme

Aplikacije so implementirane s ciljem, da jih je možno namestiti na poljubno končno strojno opremo in operacijski sistem. V praksi to pomeni, da se aplikacije zgradi s pomočjo orodja Docker, v slike, katere tečejo v orkestratorju zabojnikov Docker Swarm.

Razvijalcem se tako ni potrebno ukvarjati s podrobnostmi končnih izvajalnih okolij, morajo pa poskrbeti za karakteristike poganjanja v zabojnikih in avtomatsko upravljanem okolju:

- aplikacije imajo hiter zagon (faktor nekaj 10ms),
- shema relacijske podatkovne baze se osveži avtomatsko – ni ročnih posodabljanj baze (bodisi ob zagonu aplikacijskega zabojnika, bodisi kot ločen migracijski zabojnik),
- posamezni zabojniki so »stateless« - stanje (kot so dnevniške datoteke in podatki podatkovne baze, predpomnilnik) se hrani izven zabojnika,
- vedno je možno hkrati pognati več instanc istega zabojnika,
- podatki, ki morajo biti trajno hranjeni, se ohranijo po ponovnem zagonu zabojnika,
- aplikacija je pri pisanju dnevniških datotek zmožna zapisati tudi identifikator zabojnika v katerem teče (najbolj pogosto preko okoljskih spremenljivk).

Vse odvisnosti, ki jih aplikacija potrebuje za normalno delovanje (npr. namestitve grafičnih knjižnic ali šifrirnih algoritmov), morajo biti nameščene tekom izgradnje Docker slik. Dovoljena je uporaba javno dostopnih in priznanih (popularnih) osnovnih slik na registru DockerHub, priporočena pa je uporaba t.i. »chiseled« osnovnih slik, ki so varnostno »oskubljene« tako, da vsebujejo zgolj minimalen nabor programja, potrebnega za delovanje aplikacij.

Moderne razvojne prakse

Razvojne ekipe so pripravljene v vsakem trenutku namestiti novo verzijo aplikacije v produkcijo – to je možno z upoštevanjem modernih (ali pa pozabljenih) razvojnih praks:

- vedno obstaja samo ena verzija kode, ki se nedvoumno namešča na ciljno okolje,
- testno-voden razvoj zagotavlja, da koda zagotovo pravilno izvaja vsaj predvidene funkcionalnosti in ima zagotovo vsaj predvidene stranske učinke,

- procesno-vodeno razmišljanje zagotavlja, da implementiramo z vidika trenutnega razumevanja poslovnega procesa minimalno in točno zadostno potrebno podporo,
- upoštevanje standardov za telemetrijo in poročanje zagotavlja hiter cikel povratnih informacij v primeru težav ob nameščanju posodobitev,
- inženirji že v fazi razvoja poskrbijo, da je vsako namestitev hitro in enostavno povrniti na prejšnje stanje,
- namestitve na produkcijo so predvidljive, hitre in se lahko izvajajo tudi tekom delovnega dne.

Na platformi se uporablja sistem GitLab za vodenje kode in povezanih operativnih postopkov za nameščanje aplikacij. Tehnično konkretno se uporablja:

- repozitorije kode za različne module (Git),
- register Docker slik,
- register programskih paketov in knjižnic,
- MLOps ogrodje za upravljanje ML modelov.

Razvijalci poskrbijo za implementacijo CI/CD skripte, katero GitLab runnerji poženejo na ciljnih okoljih.