TALLINNA TEHNIKAÜLIKOOL

Infotehnoloogia teaduskond

Ivar Osila 193309IABB

Raoul Rocco Riigov 206416IABB

AJASTATUD AVALDAMISE FUNKTSIONAALSUSE LISAMINE CONTENTSTACKILE KASUTADES TEMPORALI

Bakalaureusetöö

Juhendajad: Viljam Puusep (Tallinna Tehnikaülikool)

Jeno Laszlo (Betsson Group)

Tallinn 2023

Autorideklaratsioon

Kinnitame, et oleme koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.
Autorid: Ivar Osila, Raoul Rocco Riigov

17.05.2023

Annotatsioon

Lõputöö eesmärgiks oli viia lõpule eelneva meeskonnaprojekti aine raames alustatud projekt ettevõttele Betsson Group, mis seisnes ajastatud avaldamise tarbeks spetsiaalse UI elemendi loomises Contentstacki sisuhaldussüsteemi. Ühtlasi oli kindlaks sihiks kasutada arendamisel Temporali. Lõputöö raames sooviti lõpetada UI elemendi loomine ning selle integreerimine eelnevalt arendatud lahendustega selle töötlemiseks.

Lõputöö tulemusena valminud projekt on ettevõttele POC Temporali kasutamisest arendustööks. Lahenduse abil on võimalik sisestada Contentstacki UI elementi informatsioon ajastatud avaldamise kohta, mis läbib mitmeid AWS teenuseid ja mida töödeldakse Temporali abil. Selle tulemusena kantakse ajastatud avaldamise vahemikud PostgreSQL andmebaasi.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 37 leheküljel, 5 peatükki, 49 joonist.

Abstract

The addition of timed publishing functionality to Contentstack using Temporal

This bachelor's thesis aims to build upon the previously started project for the Betsson Group which was carried out to create a custom UI element for Contentstack that would enable the feature of scheduled publishing. Additionally, the development process required the project to make use of Temporal to prove the compatibility of the framework for the development purposes.

The goal of this thesis was to finish the UI element and integrate it with the previously built solutions. The final version of the project serves as a POC for the company of using Temporal. It allows for timed scheduling information to be inserted into the custom UI element, which will be processed by multiple AWS services and Temporal workloads. As a result, the timeframes for timed scheduling will be posted into PostgreSQL database.

The thesis is in Estonian and contains 37 pages of text, 5 chapters, 49 figures.

Lühendite ja mõistete sõnastik

CMS	*Content management system,* sisuhaldussüsteem
POC	*Proof of concept*
DSL	*Domain specific language*
SQS	*Simple Queue Service*
AWS	*Amazon Web Services*
API	*Application Programming Interface*

Sisukord

Jooniste loetelu

1 Sissejuhatus

Lõputöö projekt on loodud ettevõttele Betsson Group, mis tegeleb veebikeskkonnas hasartmängude pakkumisega. Ettevõte on liikumas isetehtud CMS lahenduselt Contentstack tarkvara peale. Paraku ei paku Contentstack kõiki nõutavaid funktsionaalsuseid ja need tuleb juurde arendada. Projekti põhieesmärk on lisada ajastatud avaldamise (s*cheduled publishing*) võimalus. Kuna Contentstack on laiendatav ja omab API-d, siis on see probleem lahendatav. Projekt on ettevõttele POC, mille käigus katsetatakse ettevõttele uusi tehnoloogiaid. Uutest tehnoloogiatest kasutatakse töös Temporali. Lisaks Contentstackile ja Temporalile on töös kasutatud Dockerit, PostgreSQLi, PGAdminit, AWS SQSi, AWS Lambdat, AWS API gateway'd, AWS IAMi. Enamiku valitud tehnoloogiate puhul on saadud kasutamiseks juhised ettevõttepoolselt juhendajalt, mõne puhul valisime need juurde ise, et paremini projekti nõudeid täita.

Töö käigus olid töö valmistajad pidevas kontaktis ettevõttepoolse juhendajaga ning arendamisel lähtuti SCRUM metoodikast, mille osaks on igapäevased stand-up koosolekud. Nende käigus saadud tagasiside põhjal oli võimalik muuta valitud tehnoloogiaid või lahendusviise probleemidele.

Lõputöö valmis kujul peaks olema võimalik kasutajal sisestada teatav hulk ajalisi andmeid sündmuse avaldamise kohta Contentstacki, mida töödeldakse ning mille töötlemise järel salvestub sündmuste vahemike hulk PostgreSQL andmebaasi.

1.1 Probleem ja projekti eesmärk

Projekti eesmärk on lisada Contentstack CMSile ajastatud avaldamise võimalus. Projekti eesmärk on lahendus praegusele probleemile, mis on ajastatud avaldamise võimaluse puudumine Contentstack'is. Probleem on ettevõtte tegevusala arvestades oluline, sest tihtipeale on ettevõttel vaja avaldada mängusid, *banner*eid või *promotion*eid. Lisaks proovime seda probleemi lahendada kasutades Temporali. Temporal on platvorm, mis võimaldab luua vastupidavaid ja skaleeruvaid hajusrakendusi ja ei ole veel ettevõttes kasutusel.

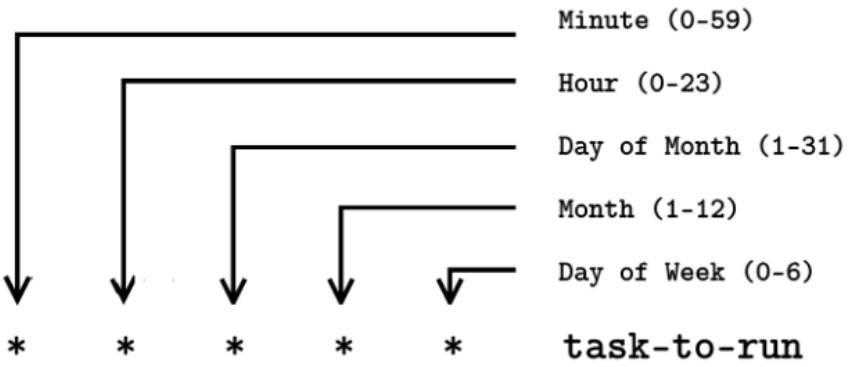
Ajastatud avaldamine peab võimaldama avaldada ja tühistada avaldamine mingil tulevasel ajal või etteantud intervallide tagant. Kasutajaliideses toimub see spetsiaalse kohandatud välja väljatöötamisega (*date picker*) Contentstack-i jaoks, mida siis saab kasutada sisu loomisel.

2 Metoodika

Ajastatud avaldamise puhuks ettevõttepoolse juhendaja poolt pakutud Cron job on laialdaselt kasutatud Linuxi- ja Unixipõhistes süsteemides, ent planeeritud ülesannete ja automatiseeritud tööde läbiviimine pole võõras ka teistes operatsioonisüsteemides ja tarkvaras. Cron job arenes välja tööde tegemiseks regulaarsete intervallide tagant ning see võimaldab automatiseerida ülesandeid ja panna ülesanded jooksma kindlal ajal. Nii on näiteks võimalik ajastada varunduste jooksutamist, süsteemi ressursside monitoorimist jpm. Kui on soov jooksutada midagi rohkem kui korra, tuleks selle automatiseerimiseks vaadata Croni suunas. Lähtuvalt ettevõtte eesmärgist liikuda Contentstacki sisuhaldussüsteemile, on Cron ajaliste andmete edastamiseks mõistlik valik kuna Contentstackist on võimalik edastada andmeid vaid stringina, tehes seda läbi payloadi.

Cron job pole paraku ajastatud avaldamise ülesande lahendamiseks ideaalne, sest tööde jooksutamine nõuab vaid töö algusaja määramist. Avaldamise puhul peab olema spetsifitseeritud ka töö lõppaeg. Selle väikese tüki funktsionaalsuse lisamiseks on mõistlik täiendada Croni veidi, lisades sellele välju ning muutes see DSLiks. DSL on hea vähese kogemusega programeerijatele, et hoida efektiivsust [1]. Ühtlasi on see kasulik kindlalt määratud väikeste probeemide lahendamiseks, nagu seda on ajastatud avaldamise puhul ajaperioodile viitamine, mitte pelgalt algusajale.

Eristada võib kaht liiki DSLe: sisemised ja välimised DSLid. Sisemist DSLi võib pidada üldotstarbelise keele erilisel viisil kasutamiseks. Sisemise DSLi skripti võib nõnda jooksutada vabalt ka üldotstarbelises keeles, kuid see kasutab vaid teatud funktsioone, et saada hakkama väikese aspektiga kogu süsteemis. Tulemusena peaks jääma tunne, et tegemist ongi suuresti kohandatud keelega. Välimine DSL on eraldiseisev keel rakenduse põhikeelest. Tavaliselt on sellisel DSLil kohandatud süntaks, ent võib kasutada ka mingi teise keele süntaksit. Välimise DSLi skripti parsitakse tavaliselt koodiga põhirakenduses, kasutades tekstiparsi tehnikaid [2]. Vaadates nende kahe DSL liigi ja kirjelduse otsa, võib valminud DSLi pidada välimiseks DSLiks.



Joonis 1. Cron jobi väljade ülesehitus koos tähenduste ja võimalike väärtustega.

AWSis kasutatud teenustest oli tegu kõigi näol projekti valmistajate jaoks uute teenustega. Nende projektile lisamise eesmärgiks oli muuta võimalikuks Contentstackist andmete kätte saamine ja nende võimalikult optimaalsel kujul hoiustamine. Ühtlasi pakkusid need võimalust esimeseks kokkupuuteks mõndade AWSi poolt pakutavate funktsionaalsustega, mida ka tulevikus võib vaja minna. Hoiustamise esmaseks tarbeks on loodud API Gateway abil Webhook API, kuhu tuleb Contentstackist välja salvestamisel edastatav payload, ning mille abil see edastatakse SQS'ile. SQS järjekordasid on kasutuses kaks, mille vahele on lisatud Lambda funktsioon. Selle abil ekstraktitakse tervest esimesse järjekorda jõudvast payloadist vajalik DSL string ning asetatakse see teise järjekorda, mis on mõeldud vaid DSLide talletamiseks.

Contentstack on headless CMS ehk sisuhaldussüsteem, mis on spetsiaalselt ehitatud selleks, et toetada mitmeid erinevaid seadmeid ja platvorme ning millel puudub kasutajaliides. Sisuhalduse toimingud tehakse API kaudu, mis võimaldab kasutajatel sisu kaugjuhtida. APIga tööd tehes konkreetse sisu muutmiseks salvestatakse UI elemendis kasutaja valikud ning edastatakse webhooki abil payloadina AWS teenustesse.

Webhook võimaldab erinevate rakenduste omavahelise suhtluse. Selle kaudu on võimalik saata ühest rakendusest teise andmeid kindla sündmuse toimumisel ning ka Contentstackis on täpselt võimalik paika panna, millise välja muutmisel või salvestamisel webhook välja saadetakse. API ja webhooki erinevus seisneb selles, et API puhul on vaja teha päring andmetele, aga webhook teavitab kui mingi sündmus toimub. Payload kujutab endast webhooki poolt saadetavaid andmeid ning sisaldab vajalikku DSLi. Webhook on seadistatud saatma andmeid AWSi API gatewayle, sest otse SQSile payloadi saatmine osutus oodatust keerukamaks. Gateways on siiski seadistatud ühendus SQSiga, mis oli soovitud viis andmete salvestamiseks. SQS on teenus, mis võimaldab hajusalt töötavatel

rakendustel edastada sõnumeid asünkroonselt, kasutades järjekordi. See teenus võimaldab rakendustel edastada sõnumeid ja teateid üksteisele ilma otsest ühendust loomata. SQSi saab integreerida teiste AWS teenustega nagu seda on Lambda. Lambda on serveriteta arvutusteenus, mis võimaldab käivitada koodi ilma virtuaalsete masinate või füüsiliste serveriteta. Lambda võimaldab rakenduste arendajatel käivitada koodi automaatselt vastavalt konkreetsetele sündmustele või ajakavale, mis võib sisaldada näiteks andmete muudatusi, API-päringuid, sõnumeid, failide värskendusi ja palju muud. Meie lahenduse puhul käivitatakse kood Lambdas vastavalt pärast payloadi jõudmist esimesse SQS järjekorda. Lambda töötleb payloadis sisalduvaid andmeid nii, et alles jääb vaid DSL, mis kantakse teise SQSi järjekorda.

Omades SQS järjekorras DSLi stringi ja selle töötlemiseks vastavat consumerit, omame loogikaga seonduvat võimekust vastavate ajavahemike loomiseks andmebaasi. Andmebaasina kasutasime vastavalt ettevõttepoolse juhendaja soovitustele PostgreSQL andmebaasi ja sellega töötamiseks PGAdmini andmebaasiserverite haldustööriista. Mõlema puhul on seadistus tehtud Dockeri konteinerite abil, et tagada projektiga seonduvate tabelite taasloomise võimalus erinevates keskkondades.

DSL consumeri loogika on kantud Temporali workloadi, mis käivitatake manuaalselt läbi terminali ning mille järel populeeritakse loodud andmebaasis tabelid vastavalt SQSis olevale DSLile. Pärast workloadi käivitamist populeerib see andmebaasi vastavalt viimasele SQS sõnumile automaatselt.

2.1 Objekti detailne kirjeldus

DSL'i keerukam osa kujutab endast CRON expressionit või sellele vastavat modifikatsiooni. CRON'i kasutatakse, et triggerida teatavaid sündmuseid kindlatel aegadel ning see koosneb peamiselt kuuest väljast, millega defineeritakse ära, kui tihti midagi toimub. See võimaldab sättida sündmuseid toimuma kindlal aastal, kindlas kuus, valitud kuu- või isegi nädalapäevadel, ent jääb hätta, kui sündmus toimub iga teatava minutite või tundide vahemike tagant. Näiteks võimaldab CRON ajatabelisse märkida sündmuseid, mis toimuvad iga 3 tunni tagant, ent seejuures teeb seda nii, et sündmused toimuvad alati vaid 00.00, 03.00, 06.00 jne. Üksik CRON suudab kujutada hästi ükskõik, millist punkti ajas, sest on võimalik defineerida tunnid ja minutid, millal sündmus toimub,

ent CRON'i nõrkuseks on ka see, et see ei tegele ajavahemikega, vaid punktidega ajas. Loodud DSL tegeleb ajavahemikega ning võimaldab sättida sündmusi toimuma keerulisematel kellaaegadel ükskõik millise intervalliga. Ühtlasi on defineeritud päevade arv, kui kaua DSL kestab. Seega on võimalik selle abil sisestada toimuma näiteks viie minutiline sündmus alates kella 4.22'st iga kolme tunni ja 30 minuti tagant 21ks päevaks. Osa näidete puhul on DSL parseri algusaeg testimise mõttes seatud detsembrisse 2022, mistõttu algavad kuupäevad alates sellest ajast. DSL'i töötlemine on asetatud Temporali workloadi. Temporali puhul on tegu tehnoloogiaga, millega polnud varasemalt kokku puutunud. Temporali võtmeelementideks on Workerid, Workloadid ja Activity'd. Workloadide käivitamine toimub läbi HTTP kutse, mille abil saame käivitada kõik vastavas Workloadis asuvad Activity'd. Workeri käivitamine toimub esialgu lokaalselt ning just Worker on see, kes suhtleb Temporali clusteriga ning tagastab sinna Workloadide käivitamisel tekkivat infot.

Temporal annab rakendustele garantii, et käivitatav kood lõpetab oma töö ilma viperusteta, sõltumata sellest, kas vahepeal mõni teenus on olnud maas või mitte. See sarnaneb andmebaasisüsteemide garantiile, et töö tehakse täielikult või üldse mitte. Sarnasus seisneb selles, et kõik toimub platvormi tasandil ja arendajad ei pea kirjutama nii öelda kaitsvat koodi, mis kaitseks rikete eest. Tulemuseks on puhtam kood.
Algus- ja lõpuaegade säilitamiseks kasutame PostgreSQL andmebaasi. Peamine on see, et sealt saaks kätte kindla kasutaja poolt sisestatud sündmused ning kõik nende sündmuste toimumisvahemikud. PostgreSQL päringute integreerimine Temporali workloadidesse oli üks keerukamaid elemente kogu projektis, sest nõudis palju katsetamist ning erinevate kihtide omavahel toimima panemist.
Contentstacki UI tarbeks on loodud element, mida hoitakse Contentstackis endas. See tähendab, et elemendi loomiseks vajalikud HTML, JS ja CSS koodijupid on kõik ühes failis, mis pole puhta koodi põhimõtteid silmas pidades just parim, ent positiivse külje pealt võimaldab eraldada muust projektist osa, mis on seotud vaid Contentstackiga ning asetada see otse sinna.

```
'* * * * MON,WED_4 22 5 33_21'
```

Joonis 2. DSL string kujutamaks ajavahemike 4.22-5.33 jada esmaspäeviti ja kolmapäeviti 21 päeva jooksul.

```
Start date: Mon Dec 05 2022 04:22:00 GMT+0200 (Eastern European Standard Time) End date: Mon Dec 05 2022 05:33:00 GMT+0200 (Eastern European Standard Time)
Start date: Wed Dec 07 2022 04:22:00 GMT+0200 (Eastern European Standard Time) End date: Wed Dec 07 2022 05:33:00 GMT+0200 (Eastern European Standard Time)
Start date: Mon Dec 12 2022 04:22:00 GMT+0200 (Eastern European Standard Time) End date: Mon Dec 12 2022 05:33:00 GMT+0200 (Eastern European Standard Time)
Start date: Wed Dec 14 2022 04:22:00 GMT+0200 (Eastern European Standard Time) End date: Wed Dec 14 2022 05:33:00 GMT+0200 (Eastern European Standard Time)
Start date: Mon Dec 19 2022 04:22:00 GMT+0200 (Eastern European Standard Time) End date: Mon Dec 19 2022 05:33:00 GMT+0200 (Eastern European Standard Time)
Start date: Wed Dec 21 2022 04:22:00 GMT+0200 (Eastern European Standard Time) End date: Wed Dec 21 2022 05:33:00 GMT+0200 (Eastern European Standard Time)
```

Joonis 3. Joonisel 2 kujutatud DSL stringi parsimise tulemusena konsooli väljastatavad ajavahemikud.

```
'* * * * *_4 22 5 33_21';
```

Joonis 4. DSL string kujutamaks ajavahemike 4.22-5.33 jada iga päev 21 päeva jooksul.

```
Start date: Sat Dec 03 2022 04:22:00 GMT+0200 (Eastern European Standard Time) End date: Sat Dec 03 2022 05:33:00 GMT+0200 (Eastern European Standard Time)
Start date: Sun Dec 04 2022 04:22:00 GMT+0200 (Eastern European Standard Time) End date: Sun Dec 04 2022 05:33:00 GMT+0200 (Eastern European Standard Time)
Start date: Mon Dec 05 2022 04:22:00 GMT+0200 (Eastern European Standard Time) End date: Mon Dec 05 2022 05:33:00 GMT+0200 (Eastern European Standard Time)
Start date: Tue Dec 06 2022 04:22:00 GMT+0200 (Eastern European Standard Time) End date: Tue Dec 06 2022 05:33:00 GMT+0200 (Eastern European Standard Time)
Start date: Wed Dec 07 2022 04:22:00 GMT+0200 (Eastern European Standard Time) End date: Wed Dec 07 2022 05:33:00 GMT+0200 (Eastern European Standard Time)
Start date: Thu Dec 08 2022 04:22:00 GMT+0200 (Eastern European Standard Time) End date: Thu Dec 08 2022 05:33:00 GMT+0200 (Eastern European Standard Time)
```

Joonis 5. Joonisel 4 kujutatud DSL stringi parsimise tulemusena konsooli väljastatavad ajavahemikud.

```
'*/30 */3 * * *_4 22 4 29_21';
```

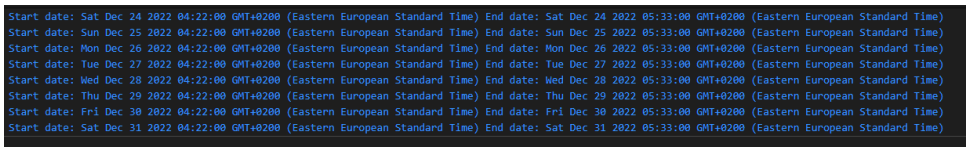
Joonis 6. DSL string kujutamaks 7-minutuliste ajavahemike jada iga 3 tunni ja 30 minuti tagant 21 päeva jooksul.

```
Start date: Tue May 17 2022 07:52:00 GMT+0300 (Eastern European Summer Time) End date: Tue May 17 2022 07:59:00 GMT+0300 (Eastern European Summer Time)
Start date: Tue May 17 2022 11:22:00 GMT+0300 (Eastern European Summer Time) End date: Tue May 17 2022 11:29:00 GMT+0300 (Eastern European Summer Time)
Start date: Tue May 17 2022 14:52:00 GMT+0300 (Eastern European Summer Time) End date: Tue May 17 2022 14:59:00 GMT+0300 (Eastern European Summer Time)
Start date: Tue May 17 2022 18:22:00 GMT+0300 (Eastern European Summer Time) End date: Tue May 17 2022 18:29:00 GMT+0300 (Eastern European Summer Time)
Start date: Tue May 17 2022 21:52:00 GMT+0300 (Eastern European Summer Time) End date: Tue May 17 2022 21:59:00 GMT+0300 (Eastern European Summer Time)
Start date: Wed May 18 2022 01:22:00 GMT+0300 (Eastern European Summer Time) End date: Wed May 18 2022 01:29:00 GMT+0300 (Eastern European Summer Time)
Start date: Wed May 18 2022 04:52:00 GMT+0300 (Eastern European Summer Time) End date: Wed May 18 2022 04:59:00 GMT+0300 (Eastern European Summer Time)
```

Joonis 7. Joonisel 6 kujutatud DSL stringi parsimise tulemusena konsooli väljastatavad ajavahemikud.

```
'* * * 12 *_4 22 5 33_21'; //every day during dec 4.22-5.33
```

Joonis 8. DSL string kujutamaks ajavahemike jada 4.22-5.33 detsembris 21 päeva jooksul.



Joonis 9. Joonisel 8 kujutatud DSL stringi parsimise tulemusena konsooli väljastatavad ajavahemikud.

Parseri CRON osa parsimise loogika pärineb ‘cron-parser’ sõltuvusest, ning soovi korral, sisestades mitmed DSLid, ühendatakse soovitud ajavahemikud kasutades ‘merge-ranges’ sõltuvust. Praeguse POC käigus kasutasime algselt rohkem ‘merge-ranges’ sõltuvust, kuid kuna projekti hilisemas faasis muutsime CRONi DSLiks, ei oma see enam nii suurt tähtsust, sest suudame ajavahemike infot edastada DSL abil. Siiski võib seda sõltuvust vaja minna, kui soovitakse ühendada mitmeid DSLide ajavahemikke, ehk võimalus arendamiseks eksisteerib. Praegu on ettevõttepoolse juhendaja soovitusel võetud suund töödelda üht kindlat DSLi. DSL töötlemise kood on laias laastus testimata, läbi on viidud vaid katsed erinevatele piirväärtustele, mis võivad kuupäevade töötlemisel eksisteerida. Avastatud on ka koodijuppe, mis ei tööta, ent mille tööle saamine on projekti oluliseks osaks, juhuks kui DSL peaks kasutusele minema. Parandatud vigade osas on suudetud DSL parseris parandada viga, mis tekkis kellaaegade töötlemisel, mis sisaldasid sama minuti ja tunninäitajaid, nt 15.15. Paraku pole praegu võimalik töödelda DSLi, mis jookseb väga mitmetel nädalapäevadel, näiteks esmaspäevast reedeni. Lisatud funktsionaalsustest on võimalik lisada ajavahemikke soovitud kellaaegadele, mis ei ole täistunnil või pooltunnil. Seda funktsionaalsust CRON ei võimalda. Ühtlasi defineeritakse üldine ajavahemik, mille jooksul soovitud ajavahemikud peavad eksisteerima. Projekti arenduse koha pealt peab DSL töötlemisega kindlasti veel vaeva nägema ja looma ühiktestid ajavahemikega seotud piirväärtustele, et vältida võimalike kasutajate omakasupüüdlikke tegevusi veebilehel, mis võivad olla seotud promotionite ajastatud avaldamisega.

```
var parser = require('cron-parser');
var mergeRanges = require('merge-ranges');
```

Joonis 10. DSL parsija loomiseks kaasatud teegid.

2.2 Tehnilise teostuse põhjendus

Tehnilise külje pealt olime suures osas sunnitud kasutama ettevõtte poolt soovitud lahendusi. Lahendused on sellise projekti puhul üsna optimaalsed. Temporali kasutamine selles projektis on tehtud uuringu eesmärgiga, ent on mitmed tugevad küljed, mida see tehnilisele lahendusele lisab. Temporal annab meile võimaluse luua projekti, mille sisu loogika on kaitstud muudatuste eest. Temporal pakub vigadele vatupidavuse läbi selle, et haldab automaatselt workflow'de taastamist vigade korral ning kindlustab, et iga samm kindlas workflow's on täidetud täpselt korra. Ühtlasi pakub see monitoorimisvahendeid workflow'de oleku kohta ning võimaldab jälgida individuaalseid töid ning nendega kaasnevaid vigasid. Temporal erineb hajusrakendusest, sest tavaline hajusrakendus ei paku kaasaehitatud workflow orkestratsiooni ja veatolerantsust. Temporali näol ongi tegu spetsiaalselt hajutatud workflow'de arendamiseks mõeldud abivahendiga. Temporalis on library'd ja API'd mis vähendavad koordinatsiooni ja oleku halduse keerukust hajutatud süsteemides, mis teeb lihtsaks hajutatud workflow'de ehitamise ja jooksutamise.

DSL on loodud eesmärgiga maksimeerimaks funktsionaalsust, mida kasutajal võib vaja minna. Ehkki tihti võib minna vaja kõigest ajastatud avaldamist, milles on defineeritud täistundidega avaldamine, on meie lahendus veidi abstraktsem ja kasutatav rohkemates kohtades kui ainult selles projektis spetsiifiliselt.

Contentstack'i eelis tavalise CMS süsteemi ees on see, et seal saab teha komponente, mida kiirelt kasutada mitmete lehtede vahel. See teeb lihtsamaks erinevate muudatuste sisseviimise veebilehe ülesehituses.

3 Realisatsioon

3.1 Nõuded

1. Contentstack on CMS, mida ettevõttes kasutatakse, aga sellel puudub täiustatud ajastatud avaldamise võimalus. See tuleb arendada.

2. POC, mis luuakse, on eksperiment, et näha, kas me saame kasutada Temporali antud probleemi lahendamiseks.

3. Contentstack kandeid (*entry*) tuleb näidata kindlate ajavahemike tagant ja ainult määratud päevadel.

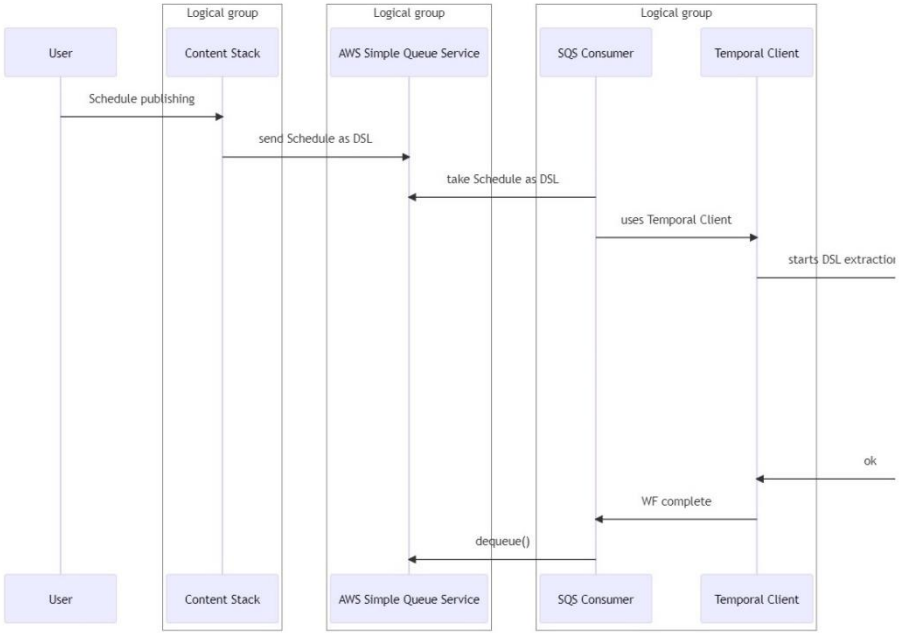
4. Avaldamise informatsiooni konfigureerimise võimalus Contentstackis.

5. Lahendus peaks muutma pideva manuaalse avaldamise ja mitteavaldamise ebavajalikuks.

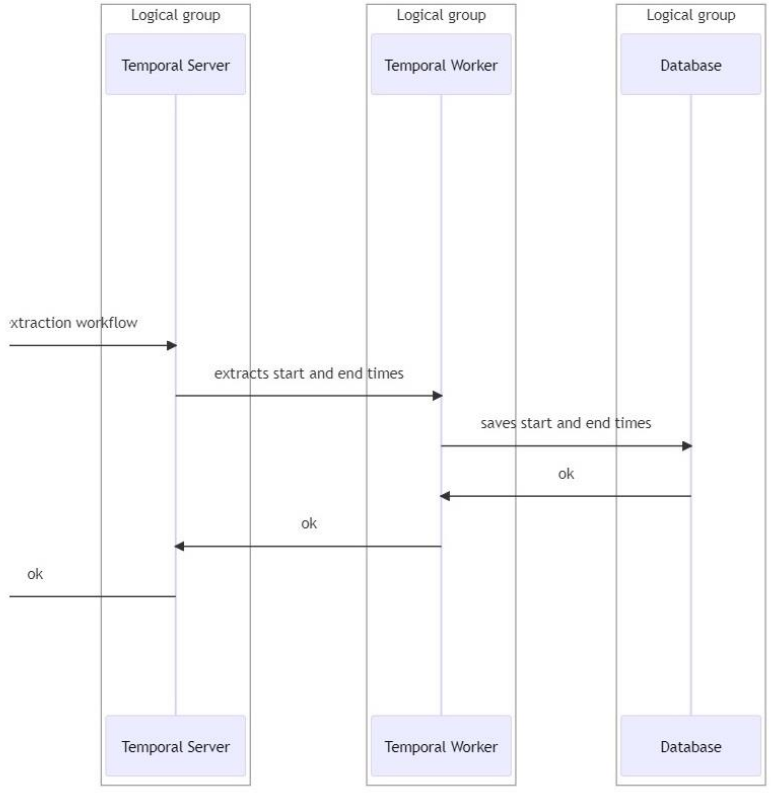
6. Avaldamise tulemusel peab tekkima kanal (*feed*), mis näitab, kas kanne peab teatud ajavahemikus olema nähtaval või mitte.

3.2 Üldine arhitektuur

Lahendus hõlmab CMSi kasutajaliidest ajastamise info sisestamiseks. Seejärel liigub see info ootejärjekorda (*queue*). Eraldi teenus võtab ajastamise info ja dokumendi tuvastmise koodi sealt ootejärjekorrast ning algatab *workflow*. *Workflow* sisestab avaldamise alguse ja lõpu ajad koos dokumendi koodiga andmebaasi. Andmebaasis olevat avaldamise infot on võimalik eraldi teenuse kaudu pärida.



Joonis 11. Üldine andmevooskeem.



Joonis 12. Üldise andmevooskeemi jätk.

3.3 Contentstack

Contentstack on headless CMS, mis põhimõtteliselt tähendab, et kogu sisu asub CMSis ja sisu jagamiseks tuleb kasutada API-d. Erinevus tavalisest CMSist on, et sellega ei ole ühendatud ühtegi kindlat front end kihti. API-d kasutades saab sama sisu kasutada nii veebilehel, mobiilses rakenduses, CRM-is jne [3].
Contentstack eelised:

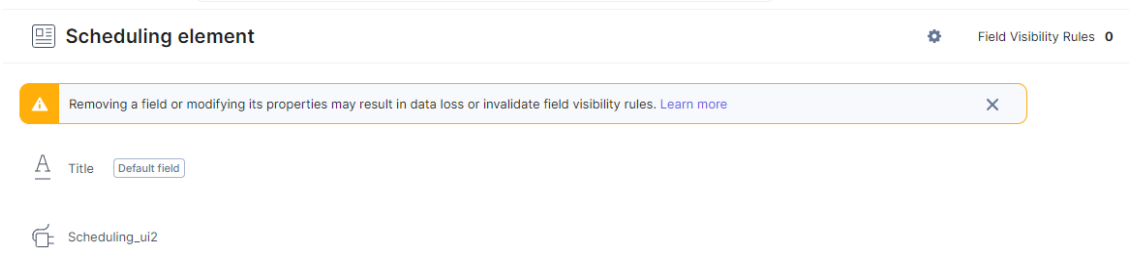
- Kui sisus toimub muutus, siis ei pea seda muutust läbi viima kõigis kasutatavates süsteemides, piisab muudatusest ühes tsentraalses kohas
- Content As A Service arhitektuur võimaldab kiiremini arendada erinevaid kanaleid, mille abil sisu hallatakse
- Modulaarne sisu, mis ei ole otseselt seotud ühegi kasutajaliidesega
- Erinevad mängude ja meedia kompaniid juba kasutavad Contentstacki edukalt [3]

3.3.1 Contentstacki elemendid

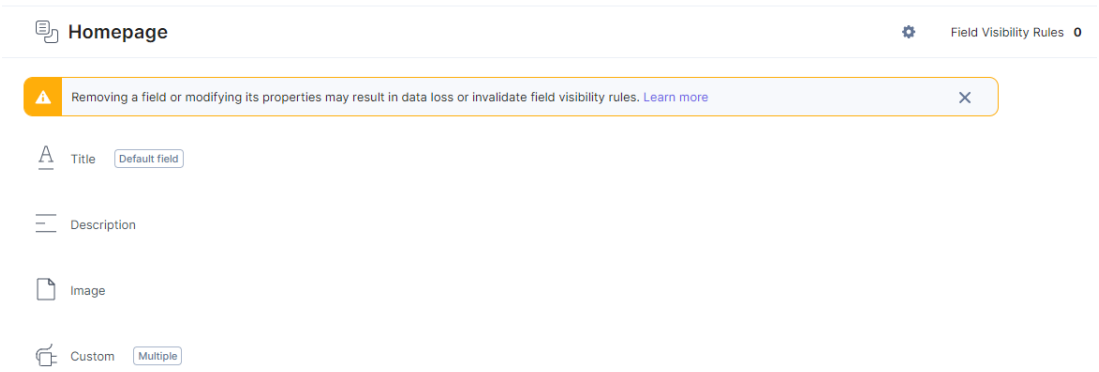
Ettevõte - emaüksus, mis kapseldab sama ettevõtte stackid, et projektide haldus oleks lihtsam [4].

Stack - konteiner, mis hoiab veebisaidiga seotud sisu (*entry*'sid) ja varasid. Toimib ka koostööruumina, kus mitu kasutajat saavad koos töötada sisu loomise, muutmise, kinnitamise ja avaldamisega. Projekti valmistajad on kaasatud Betsson Groupi poolt vastavasse POC prototüübi stacki, kus toimus Contentstackiga seotud arendus [4].

Content type - võimaldab määratleda lehe või digitaalse vara osa struktuuri. Aitab luua sisu aluse. Selle kindla projekti puhul piisas kontenttüübi loomisel vaid lisada eelnevalt loodud avaldamise element, ent keerulisematel puhkudel võib kontenttüüp muuta veebilehel muudatuste tegemise kergemaks [4].



Joonis 13. Kontenttüüp Contentstackis, et kuvada loodud UI elementi.

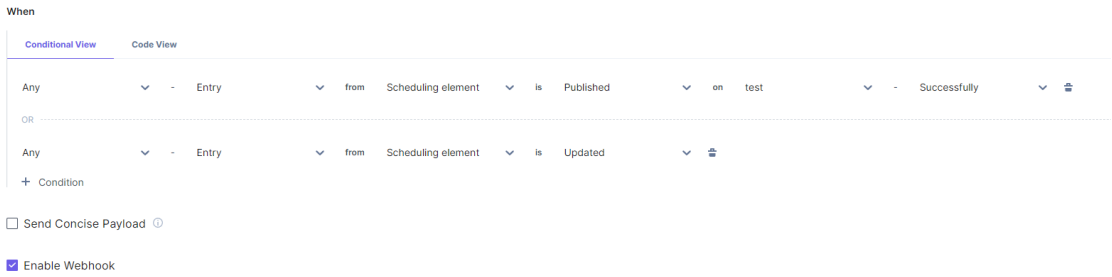


Joonis 14. Kontenttüübi võimalused keerulisema veebilehe ülesehituse puhul. Kontenttüübid muudavad veebilehe elementide haldamise lihtsaks.

Entry - tegelik sisuosa, mis on loodud ühe määratletud kontenttüübi abil. Kirje loomiseks täidavad sisuhaldurid lihtsalt andmed kontenttüübi väljadesse. Pealkirja ja avaldamise elemendi väljadesse on lisatud testandmed [4].

Joonis 15. Kontenttüübi põhjal loodud ja testandmetega täidetud Entry.

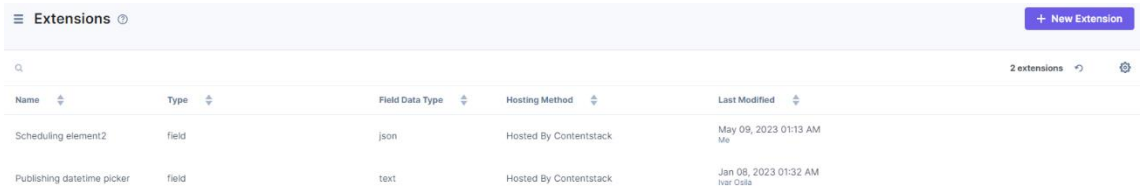
Webhook - Contentstackis nimetatakse kasutaja poolt määratletud HTTP tagasikutset (callback) webhook'iks. See on automatiseeritud süsteem, mis saadab reaalajas teavet kolmandate osapoolte rakendustele või teenustele [4]. Testimise huvides on Webhook seadistatud saatma teavet nii Entry uuendamisel kui avaldamisel.



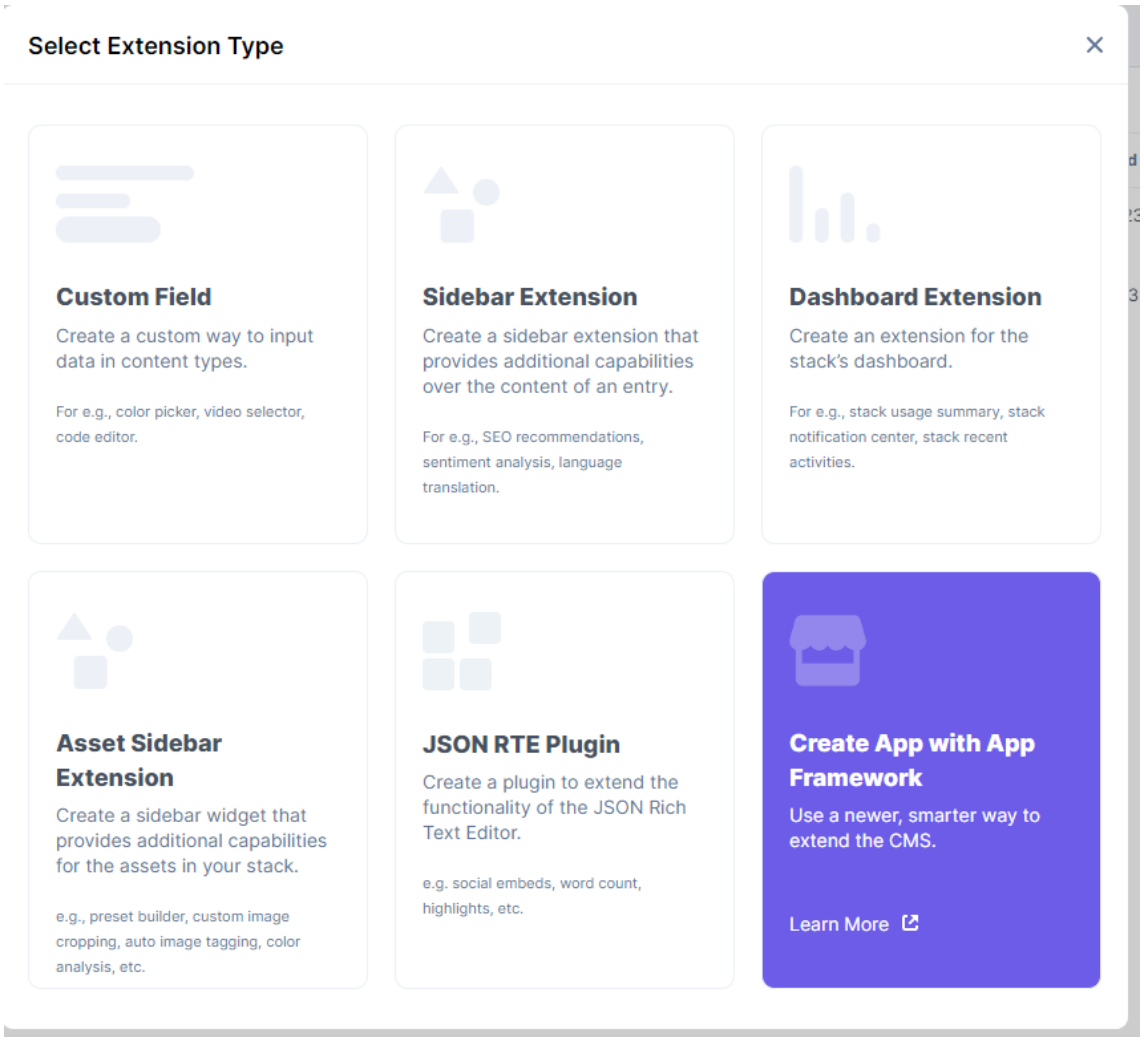
Joonis 16. Webhooki konfiguratsioon Contentstackis.

Extension - Contentstack pakub oma kasutajatele eelnevalt loodud, kasutamiseks valmis laiendusi, mida saab kasutada kontenttüüpides. Contentstackis saab luua nelja tüüpi laiendusi: kohandatud väljad (Custom Fields), külgriba laiendused (Sidebar Extensions), töölaualaiendused (Dashboard Extensions) ja JSON RTE pistikprogrammid (JSON RTE Plugins).

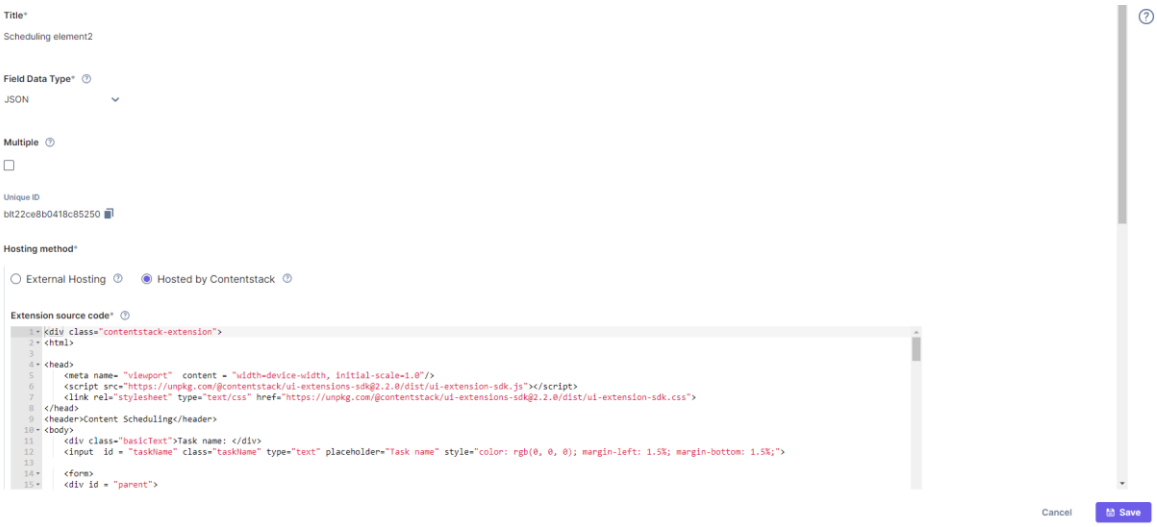
Contentstack pakub teatud eelnevalt loodud, kasutamiseks valmis laiendusi, mida saab oma kontenttüüpides kasutada. Samuti saab luua kohandatud laienduse, kas kirjutades koodi otse Contentstacki andmehoidlasse või hostides selle valitud URL-il ja kasutades seda URLi laienduse seadistamisel Contentstackis. Kohandatud väljad on spetsiifilised ainult ühele stackile ehk neid ei saa kasutada ega jagada mitme stacki vahel [4]. Avaldamise UI puhul on kasutatud võimalust luua kohandatud väli (custom field) HTMLi, JSi ja CSSi abil ning kasutatud koodi Contentstacki hoidlasse talletamist. Contentstackis pole veel võimalik kohandatud välja detailsem avamine, mis muudab mahukama välja, nagu seda on ajastatud avaldamise väli, vaatamise ja muutmise ebameeldivamaks, sest peab pidevalt väljas kerima.



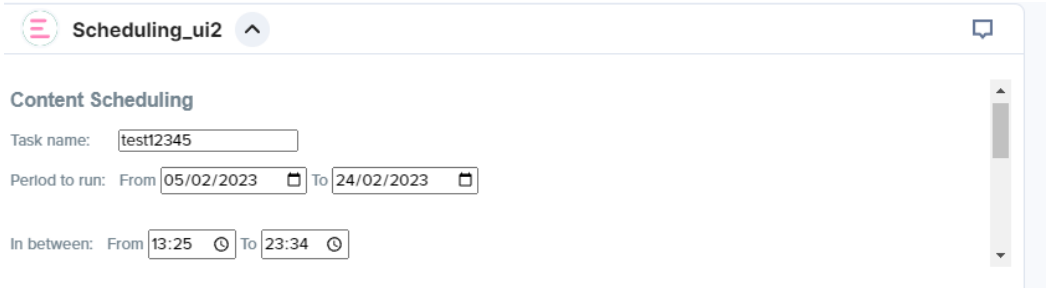
Joonis 17. Loodud laiendusväljade nimekiri Contentstackis. Nimekirjas olev ’Scheduling element2’ kujutab endast loodud kohandatud UI välja



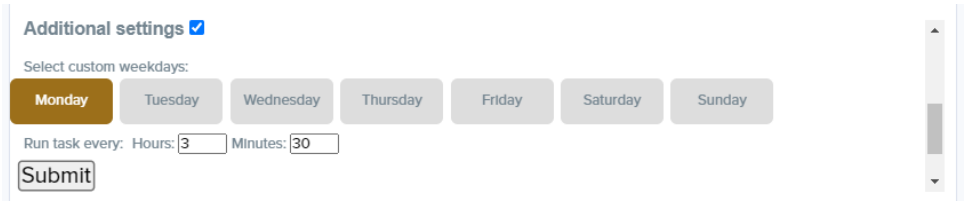
Joonis 18. Laiendusvälja loomise vaade.



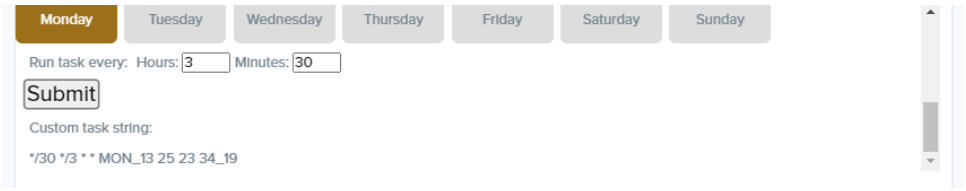
Joonis 19. Kohandatud laiendusvälja täpsem vaade.



Joonis 20. Kohandatud laiendusvälja visuaal Entry's.



Joonis 21. Kohandatud laiendusvälja visuaal Entry's #2.



Joonis 22. Kohandatud laiendusvälja visuaal Entry's #3.

Tokenid - Contentstackil on tarne-tokenid (Delivery Tokens) ja haldustokenid (Management Tokens). Neid tokeneid kasutatakse API-kõnede autoriseerimiseks [4].

3.4 AWS

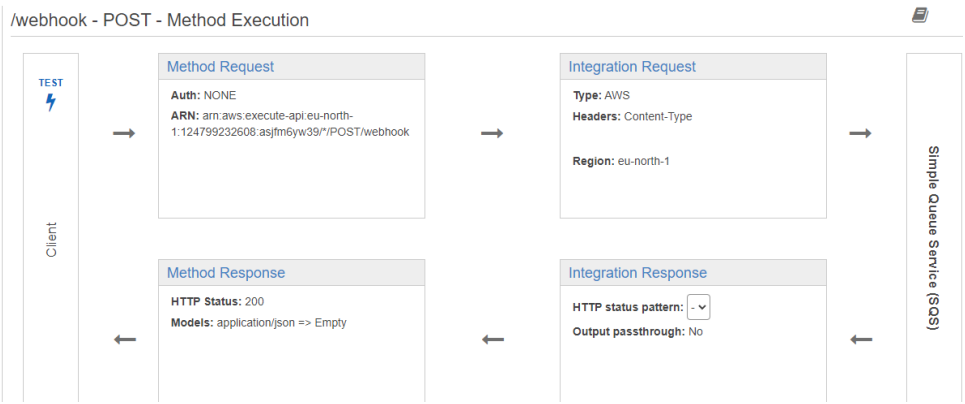
AWS pakub ettevõtetele ja üksikisikutele on-demand arvutusplatvorme ja API-sid. Arvutusplatvormid võivad hõlmata mitmesuguseid teenuseid, sealhulgas andmebaase, võrgustike juhtimist, turvalisuse tagamist, rakenduste käivitamist ja haldamist, veebiarendust ja mobiilirakenduste loomist. Sellised platvormid võimaldavad ettevõtetel või organisatsioonidel keskenduda oma rakenduste arendamisele ja käitamisele, sest arvutusressursid on välja töötatud ja hallatud kolmandate osapoolte poolt. Need teenused töötavad Amazoni serveriparkide baasil. AWS-i pilvandmetöötlusplatvormide abil saab Amazon pakkuda oma klientidele erinevaid tarkvaratööriistu ja töötlemisvõimalusi. AWS pakub mitmesuguseid teenuseid, mis on loodud vastama oma klientide vajadustele. Amazon pakub oma teenustele mitmeid hinnamudeleid, nagu näiteks on-demand hinnamudel, spot instances hinnamudel, reserved instances

hinnamudel ja saving plan hinnamudel [5]. Nende on-demand hinnamudel võimaldab klientidel kiiresti juurde pääseda erinevatele teenustele. Lisaks sellele pakub Amazon mitmeid muid tarkvaratööriistu ja teenuseid, nagu näiteks andmete ladustamine ja rakenduste levitamine [6].

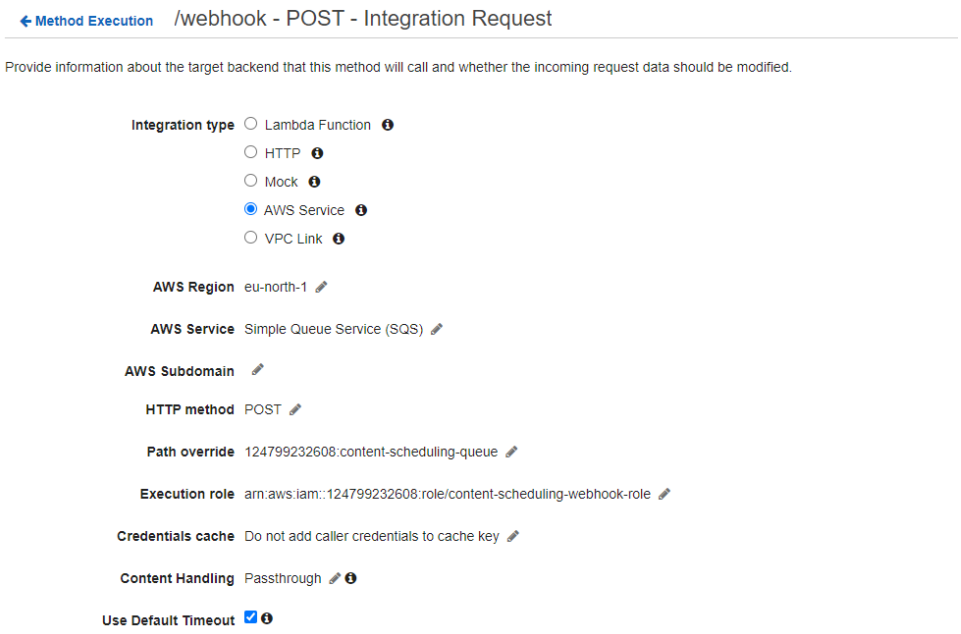
3.4.1 AWS Elemendid

API Gateway - API Gateway on üks AWSi serverivaba komponent. Seda kasutatakse RESTful ja WebSocket API-de loomiseks ja haldamiseks backendis oleva funktsionaalsuse ees.

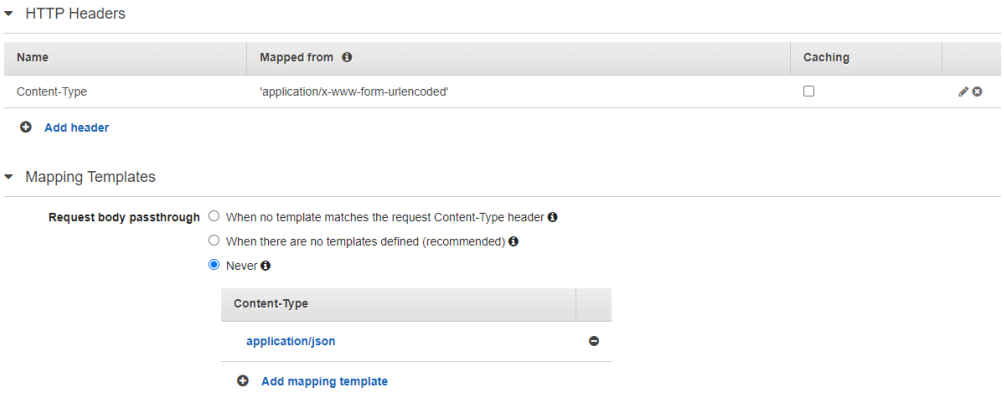
On mitmeid põhjuseid, miks API Gateway kasutamiseks. Esimene neist on rakenduse või teenuse implementatsiooni eraldamine kliendist. See võimaldab olla suurema paindlikkusega äriloogika ja töötlemise ehituse suhtes ning kliendil ei ole vaja mõista aluseks olevaid andmestruktuure või salvestamiskihti.
Amazon API Gateway võib toimida teenuse proksina. See funktsionaalsus võimaldab luua HTTP lõpp-punkti, et paljastada AWS teenus backendina. See võimaldab rakendada enda RESTful API definitsioone koos kõigi täiendavate veebipõhiste turvameetmete ja liikluse juhtimise eelistega. Samuti saab arvutuskihi täielikult vahele jätta. Näiteks võib tuua soovi kui frontend veebirakendus peab salvestama mõningaid andmeid, nagu näiteks kommentaarid teemas või vestluses olevad sõnumid. DynamoDB kasutamine teenuse proksi integratsioonipunktina võimaldab rakendada CRUDisarnast API-d, millele saab otseselt liidestuda kasutajaliides.



Joonis 23. AWS API Gateway üldine vaade.



Joonis 24. API Gateway integratsioonipäringu detailsem vaade. Integratsioonitüüp on sätestatud toimima AWS teenusele, milleks projektis on SQS. Loodud on roll, mis saab päringuid läbi viia.



Joonis 25. API Gateway integratsioonipäringu detailsem vaade #2. Sisutüübi määramine annab võimaluse lugeda webhookiga saadetavat saadavat sisu.

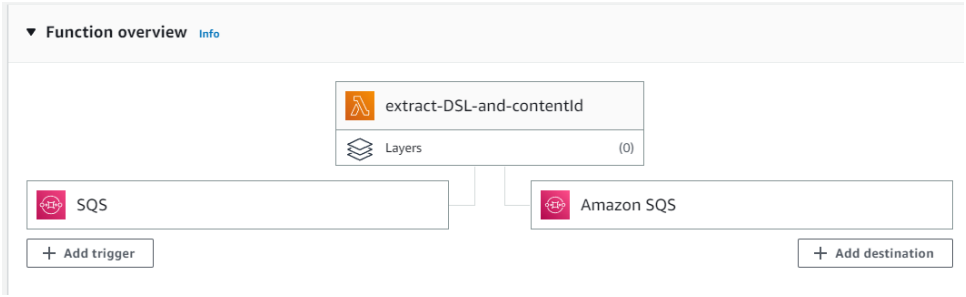
Lambda - AWS Lambda on täielikult hallatav teenus, mis liigitub serverivabade funktsioonide teenustena (FaaS) kategooriasse. See on osa sündmustepõhisest arhitektuurimustrist ja on parim viis saavutada serverivaba arhitektuuri. FaaS mudel võimaldab kasutada või rentida arvutusressurssi diskreetse tüki koodi käivitamiseks. See kood hõlmab ühte funktsiooni, mis võib moodustada ühe osa suuremast rakendusest [7].

AWSis olev Lambda on teenus, mis pakub FaaSi. Loodud koodi saab paigaldada ja käivitada eelnevalt määratletud sündmuse alusel.

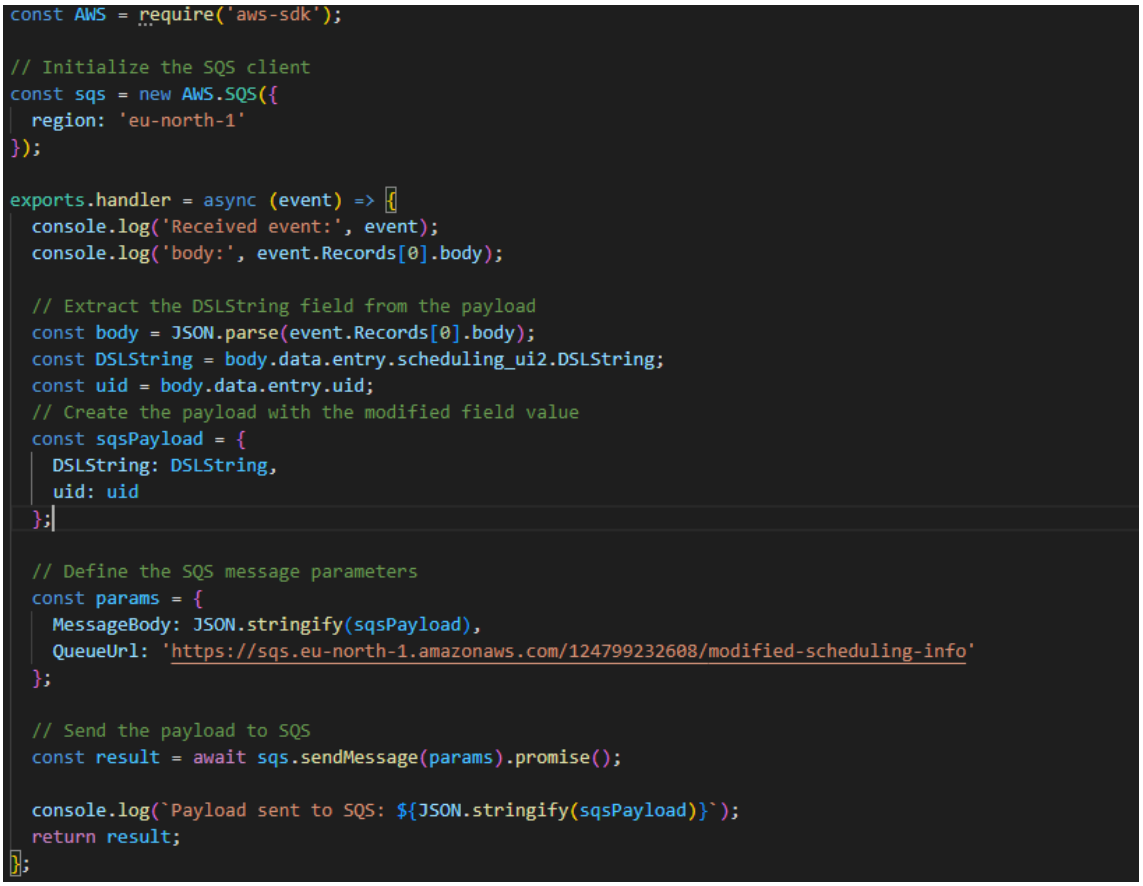
Funktsioon on kooditükk, mis käivitub alati, kui see käivitatakse mingi sündmuse poolt. Funktsiooni saab kirjutada mitmes erinevas programmeerimiskeeles ning koodiga saab kaasas kanda raamatukogusid ja sõltuvusi, mida on vaja koodi toimimiseks. Funktsioon käivitatakse AWSi poolt pakutud tööajal, platvormil ja majutusel. Serverivabades rakendustes võib vaadelda Lambda funktsioone vahelülina teenuste ja integratsioonikomponentide vahel. See on üks peamistest ehitusplokkidest AWS-is ja võimaldab sündmustepõhiste arhitektuuride toimimist suurel skaalal [7].

Lambda sobib hästi ka teisteks kasutusviisideks, mis on järgmised:
Backendi arvutused: Kui rakendused ja teenused vajavad mootorit äriloogika töötlemiseks. Selle näiteks on funktsioonide kasutamine API backendina. RESTful API-d on olemuselt sündmustepõhised ning sobivad päringu ja vastuse mudelisse.
Veebirakendused: alates serverivabast staatilisest veebisaidist kuni keerukamate rakendusteni [7].

Andmetöötlus: Lambda on ideaalne reaalajas töötluse jaoks ning asendab massitöötluse mustrid, käivitades andmete loomise punktis paralleelse töötlemise.
Operatsioonid ja automatiseerimine: On võimalik teostada automaatset parandust vastusena sündmustele, käivitada automatiseeritud ja ajastatud ülesandeid ning skaneerida oma infrastruktuuri vastavusastmete säilitamiseks.
Lisaks sellele saab kasutada Lambdat Chatbottide jaoks ja kasutusjuhte eksisteerib mitmeid teisigi. Lambda funktsioon võib sisaldada koodi, mis teeb teostusmudeli raames mis tahes ülesannet. Näiteks võib kasutada seda veebilehe oleku regulaarseks kontrollimiseks, failil töötlemiseks või reageerimiseks AWSis muutunud atribuudile [7].



Joonis 26. Lambda toimib vahelülina kahe SQS järjekorra vahel ning jätab webhookist alles vaid DSL stringi ja id. Lambda funktsiooni kood on suuruse tõttu lisatud AWSi .zip failina.



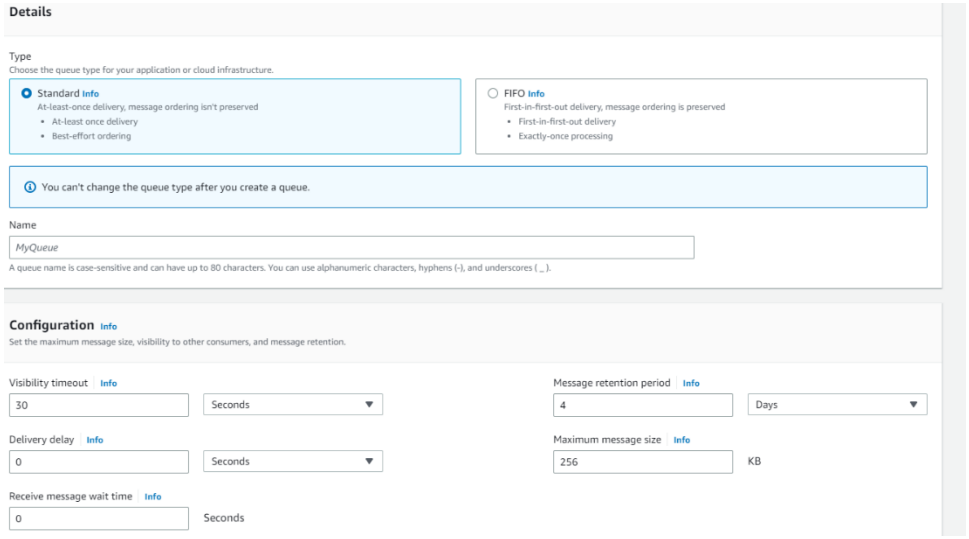
```
const AWS = require('aws-sdk');

// Initialize the SQS client
const sqs = new AWS.SQS({
  region: 'eu-north-1'
});

exports.handler = async (event) => {
  console.log('Received event:', event);
  console.log('body:', event.Records[0].body);

  // Extract the DSLString field from the payload
  const body = JSON.parse(event.Records[0].body);
  const DSLString = body.data.entry.scheduling_ui2.DSLString;
  const uid = body.data.entry.uid;
  // Create the payload with the modified field value
  const sqsPayload = {
    DSLString: DSLString,
    uid: uid
  };

  // Define the SQS message parameters
  const params = {
    MessageBody: JSON.stringify(sqsPayload),
    QueueUrl: 'https://sqs.eu-north-1.amazonaws.com/124799232608/modified-scheduling-info'
  };

  // Send the payload to SQS
  const result = await sqs.sendMessage(params).promise();

  console.log(`Payload sent to SQS: ${JSON.stringify(sqsPayload)}`);
  return result;
};
```

Joonis 27. Lambda funktsiooni kood, mis kasutab AWS SDK'd, et töödelda esimesse järjekorda saabuvat payloadi. Alles jäetakse vaid DSLString ja uid, mis edastatakse teise järjekorda.

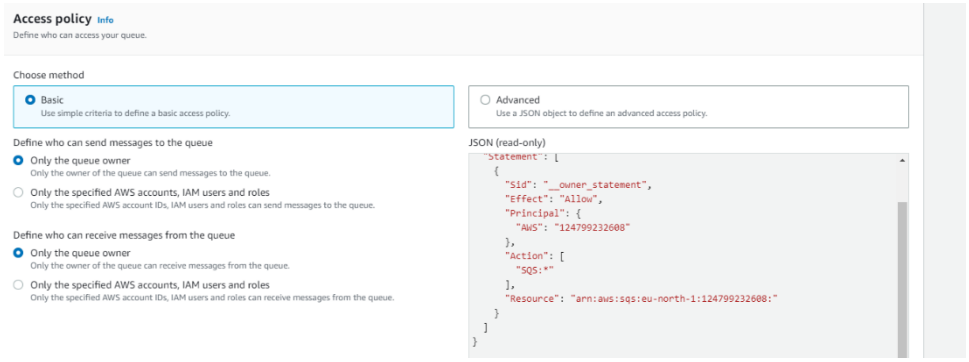
SQS - Simple Queue Service ehk SQS, on võimas idee, mis on pakendatud lihtsasse tootesse. See võimaldab saata sõnumeid rakenduse komponentide vahel, kartmata kõiki asju, mis võivad valesti minna. Iga komponent, mis peab sõnumi saatma teisele komponendile või rakendusele, saadab selle Amazoni majutatud SQSi. Kõik rakendused, mis peavad sõnumit vastu võtma, loevad samast järjekorrast. Kui sõnum töödeldakse edukalt, kustutatakse see järjekorrast, vastasel juhul säilitatakse see, kuni see uuesti töödeldakse. Selline arhitektuur võimaldab luua usaldusväärseid ja skaleeritavaid rakendusi, keerukust arendajatele lisamata [8].

Järjekorranime parameetri väärtus võib olla alfanumbriline kombinatsioon pikkusega kuni 80 tähemärki. See võib sisaldada alakriipse (_) , sidekriipse (-) , tähti ja numbreid. On olemas ka spetsiaalseid järjekordi, mida nimetatakse first-in-first-out (FIFO) järjekordadeks, mille lõpp on ".fifo". Kui järjekorranimes on vigaseid tähemärke või kui see ei lõpe ".fifo" -ga, siis luuakse järjekorra loomisel tõrketeade.

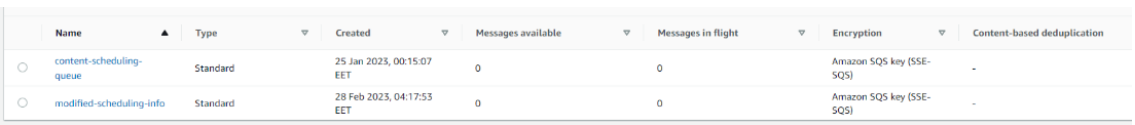
Igal järjekorral võib olla erinevaid eelistusi, mida konfigureerida. Näiteks saab konfigureerida, kui kaua sõnum järjekorras peaks enne kustutamist olema või kontrollida, kui suur võib olla sõnum, mida SQSi poolt tagasi ei lükata. Need konfiguratsioonid saab edastada järjekorra loomise ajal [8].



Joonis 28. SQS järjekorra loomise vaade.



Joonis 29. SQS järjekorra loomise vaade #2. Järjekorda luues on võimalik sätestada, kes pääsevad järjekorrale ligi ja saavad sinna sõnumeid saata.



Joonis 30. Projektis kasutatavate SQS järjekordade nimekiri. Korrektse toimimise puhul peaks sõnumite arv olema mõlemas järjekorras pidevalt 0, sest saadud info toodetakse koheselt vastavalt Lambda või Temporali workloadi poolt.

IAM - AWS Identity and Access Management (IAM) on veebiteenus, mis aitab turvaliselt hallata juurdepääsu AWS ressurssidele. IAM-ga saab keskselt hallata õigusi, mis kontrollivad, millistele AWS ressurssidele kasutajad pääsevad ligi. IAM-i kasutatakse, et kontrollida, kes on sisse logitud ja autoriseeritud kasutama ressursse [9].

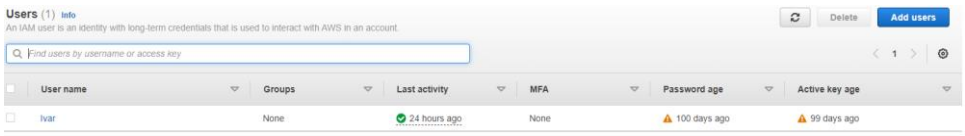
AWS-i konto luues, alustab igaüks ühe sisselogimisidentiteediga, mis omab täielikku juurdepääsu kõigile AWS-i teenustele ja ressurssidele kontol. Seda identiteeti nimetatakse AWS-i konto juurkasutajaks (root user) ja sellele pääsetakse juurde, logides sisse e-posti aadressi ja parooliga, mida kasutati konto loomisel [9].

IAM võimaldab järgmiseid funktsioone:

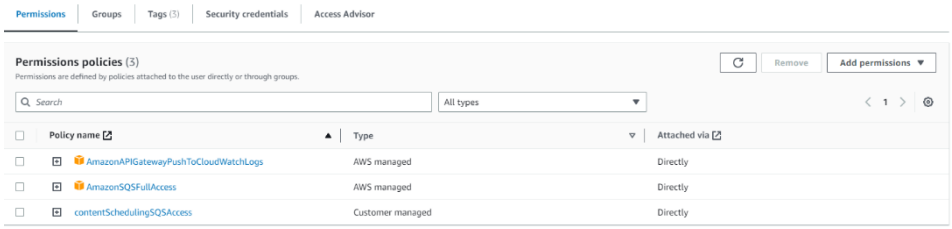
Jagatud juurdepääs AWS kontole
On võimalik anda teistele inimestele õiguse hallata ja kasutada ressursse enda AWS kontol ilma, et peaks jagama parooli või juurdepääsuvõtit [9].

Detailne õiguste juhtimine
Erinevatele inimestele saab anda erinevaid õigusi erinevatele ressurssidele. Näiteks võib lubada mõnel kasutajal täieliku juurdepääsu Amazon Elastic Compute Cloud'ile (Amazon EC2), Amazon Simple Storage Service'ile (Amazon S3), Amazon DynamoDB-le, Amazon Redshiftile ja teistele AWS-i teenustele. Teistele kasutajatele saab lubada vaid lugemisõigusi mõnele S3 ämbrile või õigusi hallata ainult mõningaid EC2 eksemplare [9].

Lisaks nendele võimaldab IAM järgnevat: turvalist juurdepääsu AWS-i ressurssidele Amazon EC2-s töötavate rakenduste jaoks, mitmekordset autentimist (MFA), identiteediföderatsiooni, identiteediinformatsiooni tagatiseks, PCI DSS nõuetele vastavust, tasuta kasutamist. Just detailsemalt kirjeldatud funktsioone ja võimalust IAMi tasuta kasutada tarbiti ka projekti valmistamisel, sest arendamisel oli vaja kõigil arendajatel võimalust ligi pääseda SQS järjekordadele ning nendes olevaid andmeid vaadata ja pärida [9].



Joonis 31. IAM võimaldab luua kasutajaid, et tagada neile kergelt ligipääs vajalikele ressurssidele.



Joonis 32. Kasutajatele tuleb määrata vastavad õigused ressursside töötlemiseks.

Role name	Trusted entities	Last activity
AWSServiceRoleForAPIGateway	AWS Service: ops.apigateway (Service-Linked Role)	-
AWSServiceRoleForSupport	AWS Service: support (Service-Linked Role)	22 days ago
AWSServiceRoleForTrustedAdvisor	AWS Service: trustedadvisor (Service-Linked Role)	-
content-scheduling-webhook-role	AWS Service: apigateway	24 hours ago
extract-DSL-and-contentId-role-ztpkxqyy	AWS Service: lambda	15 minutes ago

Joonis 33. Eraldatakse kasutajaid ja rolle. Kasutajatele saab määrata rolle. ’content-scheduling-webhook-role’ on vajalik roll, mis on loodud webhooki töötlemiseks API gateways.

Joonis 34. Õiguste jagamiseks saab luua poliitikaid. ’contentSchedulingSQSAccess’ on poliitika, mis annab root useri SQS järjekordadele haldamisõigused.

3.5 Temporal platvormi arhitektuur

Temporal on avatud lähtekoodiga platvorm, mis võimaldab luua vastupidavaid ja skaleeruvaid hajusrakendusi. Tegemist on suhteliselt uue tehnoloogiaga, mis on loodud 2018 aastal. Temporal on disainitud selliselt, et ta muudaks töö arendaja jaoks

kergemaks, võimaldades kirjutada äriloogikat ja peita hajusrakendustega seotud keerukust.

Temporali platvorm koosneb serverist ja klientidest.

Temporali kliendid on CLI, Web UI ja SDK klient. CLI on käsurea rakendus, millega saab Temporali serveriga suhelda. WEB UI on kasutajaliides, millega saab Temporali serverit hallata. SDK klient on see, mida tavaliselt kasutatakse loodavas rakenduses serveriga suhtlemiseks [10].

Temporali Server koosneb *frontend* ja *backend* teenustest. On üks frontend teenus ja mitu backend teenust, mis töötavad koos ja on horisontaalselt skaleeruvad. Produktsioonis on tavaliselt mitu instantsi igast teenusest jaotatud mitme masina vahel, et tõsta töökindlust ja jõudlust [10].

Temporali Cluster on Temporal serveri töötav osa, mis on jagatud mitme masina vahel, ning mis kasutab lisakomponente. Ainuke kindlasti nõutud komponent on andmebaasihaldur, meie projekti raames on kasutusel PostgreSQL. Temporali Cluster jälgib ja salvestab iga *workflow* täitmist, hetkeolekut ja sündmusi, mis on toimunud, et rikke korral taastada süsteemi hekteolek. Eraldi komponendina on võimalik lisada Elasticsearch, mis annab juurde efektiivsema otsimise ja sorteerimise [18].

Üks asi, mis tuleb üllatusena on, et Temporali Cluster ei käita koodi, vaid lihtsalt tagab selle õigeaegse ja veakindla täitmise läbi orkerstreerimise. Rakenduse kood töötab väljaspool Clusterit, tüüpiliselt eraldi serverites. Koodi täitmist viivad läbi Worker-id, mis on eraldi serverites. Worker suhtleb serveriga, et hallata Workflow’de täitmist. Rakenduse kood sisaldab *worker*i lähtestamist ja *workflow* äriloogikat, mis on kirjas koodina ning vajaduselt ka koodi, millega saab vaadata *workflow* olekut. Käitusajal on vaja ka kõiki teeke, mida see kood kasutab ja mida *worker* käitab.

Temporal Clusteri võib ise hostida või kasutada Temporali pakkumist (Temporal on nii firma nimi kui ka toote nimi). Meie kasutame oma projekti piirides Docker Compose’i ja ise hostimist, mis võimaldab lihtsalt Temporali rakendusi arendada. Ise hostimise alternatiivi nimi on Temporal Cloud, mis annab 99.9% ülalolekuaja garantii. Ükskõik, kas kasutada ise hostitud Temporal Clusterit või Temporal Cloud pakkumist, äriloogika kood jookseb serverites, mida tuleb ise kontrollida. Temporal Clusteril ei ole äriloogika koodile juurdepääsu.

Temporali eelised:

- Abstrahheerib ära madalamatel kihtidel olevad detailid.
- Pakub programmeerimise mudeli ja infrastruktuuri, millega luua hajusrakendusi.
- Töökindlus: tõrgete tekkides ei lähe andmed kaduma. Proovib automaatselt uuesti *workflow* läbi viia.
- Skaleeruvus: toetab paljude *workflow*'de paralleelset töötamist
- Olekuhaldus *workflow*'dele: kasulik eriti pikalt jooksvate töövoogude puhul

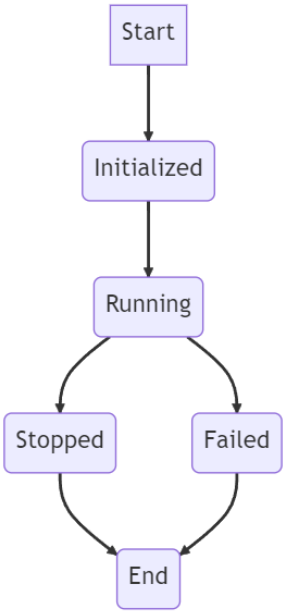
Temporal pakub SDK'sid mitmes programmeerimiskeeles: Go, Java, TypeScript, PHP, Python. Meie kasutame oma projekti piirides TypeScript SDK'd.

Temporali arendustarkvara (SDK) annab töötamiseks järgmised vahendid:

- Kliendi, millega klasteriga suhelda
- API-d *workflow'dega* töötamiseks
- API-d Workerite haldamiseks
- Activity kirjutamise API-d
- Üldkasutatavad teegid [11]

Temporal omab ka CLI-d nagu eespool mainitud. Selle nimi on *tctl* ja sellega saab klasteriga suhelda: alustada *workflow*sid, jälgida töövoogude ajalugu jne. Seal on 10 erinevat käsku, mida saab käitada ja millel on erinevad alamkäsud.
Temporali projekti loomine TS keeles kasutab standardiseeritud kataloogide, alamkataloogida ja failide struktuuri. Seda on ka meie projektis jälgitud. Eraldi koodifailid on *activity*te, *workflow*de, *worker*ite ja klientide jaoks.
Workflow definitsioon on funktsioon, mis peab jälgima nelja sammu:

- Importima Workflow paketi SDK-st
- Importima Activity tüübid vastavast koodifailist
- Defineerima Activity parameetrid. Näiteks aeg, mille jooksul Activity peab algama, enne kui ta aegub
- Sisaldama koodi, mis kutsub välja Activity [11]



Joonis 35. Workeri olekute skeem.

Initialized – algne olek peale Worker.create() kutset ja serveriga ühenduse loomise õnnestumist.
Running – peale Worker.run() kutset, *worker* pollib ootejärjekorda.
Failed – *worker* sattus taastamatusse olukorda ja Worker.run() peaks failima

Workflow definitsioon on asünkroonne funktsioon. Et Temporal saaks salvestada *workflow*-de sisendit ja väljundit, siis peavad need andmed olema serialiseeritavad, mis tähendab, et ei saa kasutada Date ja BigInt tüüpe. Sisend- ja väljundparameetrid on siiski salvestatud ka Temporal klastris, vajadusel on võimalik luua enda andmekonverter, mis krüpteerib andmed klastrisse sisenemisel ja dekrüpteerib väljudes [11]. Meil seda nõuet ei ole ja jätame võimaluse kasutamata.
Nagu mainitud, siis Temporali Event History sisaldab funktsioonide sisendeid ja väljundeid, mis saadetakse üle võrgu klastrisse, seetõttu on mõistlik hoida need andmete kogused võimalikult väikesed. Server annab ka koguste suurenedes hoiatusi, olenevalt tõsidusest.

```
function getDSLExpression(message:any)
{
    const body = JSON.parse(message.Body);
    const expression = body.DSLString;
    //const body = "*/30 */3 * * *_4 22 4 29_21";
    return expression;
}
function getContentId(message:any)
{
   const body = JSON.parse(message.Body);
   const id = body.uid;
   return id;
}

const app = Consumer.create({
   //queueUrl: 'https://sqs.eu-north-1.amazonaws.com/124799232608/content-scheduling-queue',
   queueUrl: 'https://sqs.eu-north-1.amazonaws.com/124799232608/modified-scheduling-info',
   handleMessage: async (message:any) => {
      const expression = getDSLExpression(message);
      const c_id = getContentId(message)
      console.log(c_id + " " + expression)
      await startExtractDSLWorkflow(expression, c_id);
      console.log('Workflow complete----------------------------------')
      //app.stop();
   },
   sqs: new SQSClient({
     region: 'eu-north-1',
     credentials: {
       accessKeyId: [redacted],
       secretAccessKey: [redacted]
     }
   })
});
```

Joonis 36. SQS consumeri loogika, mis võtab järjekorrast payloadi, käivitab workflow ja lisab need PostgreSQL andmebaasi

```
export async function addScheduleWorkflow(startDate:string, endDate:string, contentId:string):Promise<ResultIterator | null>
{
   return await AddSchedule(startDate, endDate, contentId);
}

export async function deleteScheduleWorkflow(scheduleId:string):Promise<ResultIterator | null>
{
   return await DeleteSchedule(scheduleId);
}

export async function updateScheduleWorkflow(scheduleId:string, startDate:string, endDate:string, contentId:string):Promise<ResultIterator
{
   return await UpdateSchedule(scheduleId, startDate, endDate, contentId);
}

export async function GetSchedulesBetweenByContentIdWorkflow(contentId:string, startDate:string, endDate:string):Promise<ResultIterator |
{
   return await GetSchedulesBetweenByContentId(contentId, startDate, endDate);
}
```

Joonis 37. Koodinäide mõnest Temporali workflow’st.

```
export async function extractDSL(expression:string, contentId:string): Promise<boolean>{
  const matrix:Array<any>=  extractDateTimesArray(expression);// "*/30 */3 * * *_4 22 4 29_21"
  matrixToISOStrings(matrix);
  try{
      await DeleteSchedulesWithContentId(contentId)
      await DeleteContent(contentId)
      await AddContent(contentId, "Test name")
      //calls addSchedule multiple times
      await saveSchedules(matrix, contentId);
  }
  catch(error)
  {
    console.log(error)
  }

  console.log('successfully saved -------------------------')
  return true;
}

export async function ContentIdExists(contentId:string):Promise<boolean>
{
  const client: DBClient = CreateClient();
  const result = await client.ContentIdExists(contentId);
  return result;
}
```

Joonis 38. Koodinäide mõnest Temporali activity'st. 'extractDSL' activity.

```
export async function DeleteContent(contentId:string)
{
  const client: DBClient = CreateClient();
  const result = await client.DeleteContent(contentId);
  return result;
}
export async function AddContent(contentId:string, name:string):Promise<ResultIterator| null>
{
  const client: DBClient = CreateClient();
  const result = await client.AddContent(contentId, name);
  return result;
}

export async function getAllSchedules(): Promise<ResultIterator |null> {
  const client:DBClient = CreateClient();
  const result = await client.GetAllSchedules();
  return result;
}

export async function AddSchedule(startDate:string, endDate:string, contentId:string): Promise<ResultIterator |null>{
  const client:DBClient = CreateClient();
  const result = await client.AddSchedule(startDate, endDate, contentId);
  return result;
}
```

Joonis 39. Koodinäide mõnest Temporali activity'st #2.

3.6 gRPC

GRPC loodi Google poolt hästi skaleeruvaks kommunikatsioonikanaliks. GRPC toetab koormuse tasakaalustamist, kahesuunalist voogedastust, autentimist ja erinevaid diagnostika utiliite. GRPC kasutab andmete pakkimiseks binaarset serialiseerimist, mis

muudab ta väga kiireks ja efektiivseks. GRPC klient saab kutsuda välja funktsioone eemalolevas serveris just nagu see oleks lokaalne funktsioon ning toetab paljusid erinevaid keeli ja raamistikke [12].

Kahesuunaline voogedastus on voogedastuse moodus, kus mõlemad pooled saavad voost lugeda ja kirjutada misiganes järjekorras. Server võib oodata kõiki kliendi teateid enne vastuteate saatmist, võib ka lugeda teate ja kohe saata vastuse ja võib ka kasutada mingit muud järjekorda [13].

3.7 Temporal parimad praktikad

- Temporali *workflow-s* peab sisend olema määratud klassina, kuna see parandab oluliselt koodi muutustele vastupidamist
- *Task Queue* nimi peab olema määratud konstandis, sest vale nime kasutamise korral võib juhtuda, et seda viga ei leita piisavalt varakult
- Käivitada produktioonis vähemalt kaks *worker-i* protsessi ühe Task Queue kohta, sest see parandab oluliselt vastupidamist erinevatele tõrgetele
- *Workflow* käivitamisel tuleb talle panna unikaalne *workflow* Id, mis on antud kontekstis loogiline

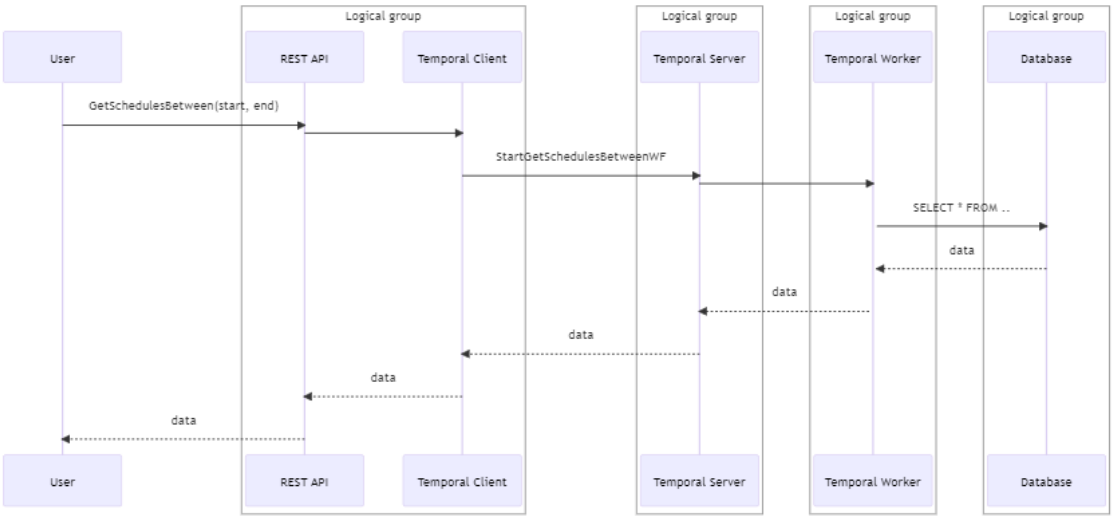
Kuigi eelnevad soovitused on triviaalsed, tuleb neid jälgida ka juhul, kus see ei paista suure probleemina [11].

3.8 REST API

Eelpool mainitud *feed* on implementeeritud REST API-na, mis näitab, kas kanne peab teatud ajavahemikus olema nähtaval või mitte. Võimalik on teha erinevaid päringuid:

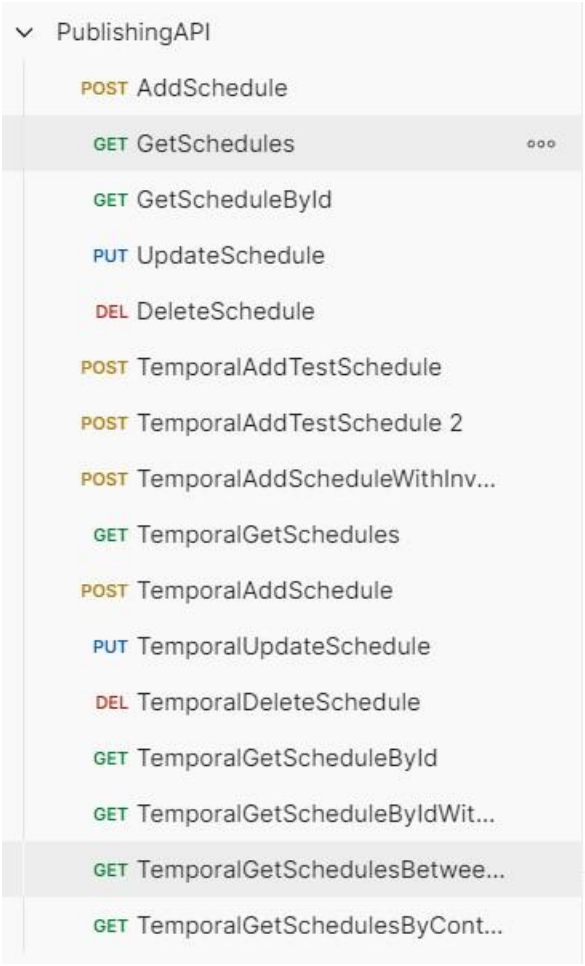
- Näha avaldamisi, mis jäävad kahe kuupäeva vahele
- Näha avaldamisi, mis on kindla *entry* Id-ga

- Teha CRUD funktsioone

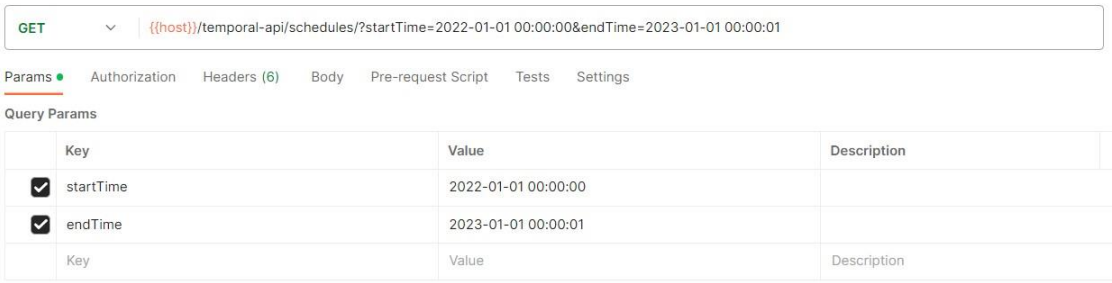


Joonis 40. REST API andmevooskeem.

REST API integratsiooniteste tegime kasutades Postmani, kuna sellega olime juba varemalt tuttavad.



Joonis 41. Postmani vaade REST API-le.



Joonis 42. Postmani vaade REST API-le #2.

3.9 PostgreSQL andmebaas ja funktsioonid

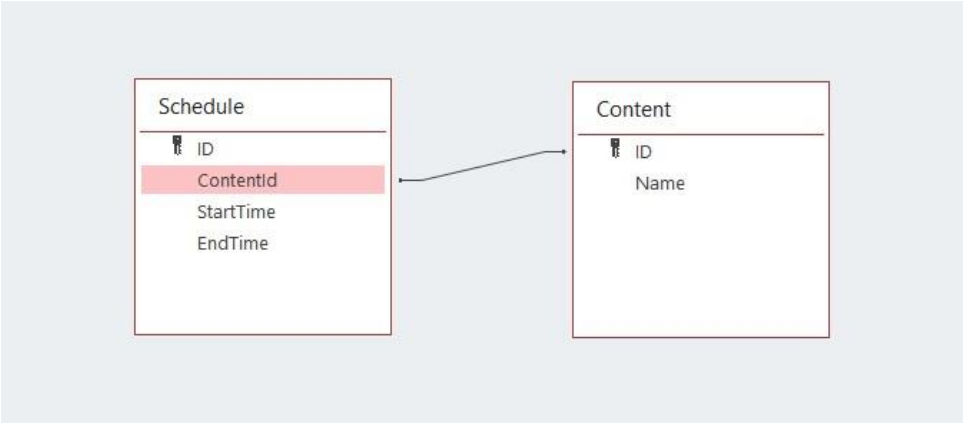
Mainitud REST API jaoks on vaja luua andmebaas ja teha päringuid. Andmebaasis on tabel Content, kus on ContentId ja tabel Schedule, mis koosneb ContentId-st ning avaldamise algus- ja lõpu ajast, mis on koos ajavööndiga salvestatud.

PostgreSQL funktsioonid on kirjutatud talletatud protseduuridena, mida saavad päringutes kõik kliendid kasutada. Kasutada PostgreSQL protseduure oli üks tehnilistest nõuetest. Olemas on järgmised protseduurid:

- Päring kõigi avaldamiste kohta, mis on kindla ContentId-ga ja jäävad kindla ajavahemiku vahele
- Päring kõigi avaldamiste kohta, mis jäävad kindla ajavahemiku vahele
- CRUD funktsioonid avaldamiste kohta

Lisaks on olemas skriptid, mis võimaldavad need tabelid ja päringud andmebaasis üles seada. Üles seadmise skript rakendub automaatselt kasutades meie Docker Compose faili andmebaasi loomiseks.

Me leiame, et talletatud protseduurid on päringute jaoks efektiivsed, sest näiteks kustutamise operatsiooni puhul on võimalik tagastada ka andmed selle kohta, kas midagi ka päriselt kustutati ja seda ilma lisaprotseduuride kirjutamiseta.



Joonis 43. Andmebaasi skeem.

3.10 Dokumentatsioon

Dokumentatsiooni loomiseks kasutasime Mermaid js abi, mis on võimas süsteem skeemide ja visualisatsioonide kirjeldamiseks. Mermaidi süntaks võimaldab luua andmevoo skeeme, andmebaaside skeeme, töövoo skeeme jne. Mermaid skeemid luuakse teksti põhjal ja on seega lihtsasti integreeritavad erinevatesse süsteemidesse. Kuna skeemid luuakse märgistuskeeles jääb ära vajadus salvestada suuri pilte pildiformaadis.

Lihtne on jagada skeeme arendajate vahel ja luua erinevaid malle, mida igaüks saab kasutada. Samuti lihtsustab see skeemide lisamist versioonikontrolli süsteemidesse. Peamine eelis seisneb ikkagi selles, et me ei vaja näiteks Enterprise Architect-i litsentse ja installeerimist, ehk me ei sõltu mingist kindlast tarkvarast. Samuti on võimalik kirjutada koodi, mis genereerib märgistuskeelt.

```
sequenceDiagram
    participant U as User
    box Logical group
    participant CS as Content Stack
    end
    box Logical group
    participant SQS as AWS Simple Queue Service
    end
    box Logical group
    participant SQSC as SQS Consumer
    participant TC as Temporal Client
    end
    box Logical group
    participant TS as Temporal Server
    end
    box Logical group
    participant TW as Temporal Worker
    end
    box Logical group
    participant DB as Database
```

Joonis 44. Mermaid märgistuskeele näidis. Skeemis osavõtjate defineerimine.

```
U->>CS: Schedule publishing
CS->>SQS: send Schedule as DSL
SQSC->>SQS: take Schedule as DSL
SQSC->>TC: uses Temporal Client
TC->>TS: starts DSL extraction workflow
TS->>TW: extracts start and end times
TW->>DB: saves start and end times
DB->>TW: ok
TW->>TS: ok
TS->>TC: ok
TC->>SQSC: WF complete
SQSC->> SQS: dequeue()
```

Joonis 45. Skeemis osavõtjatevaheliste seoste kirjeldamine.

3.11 Docker

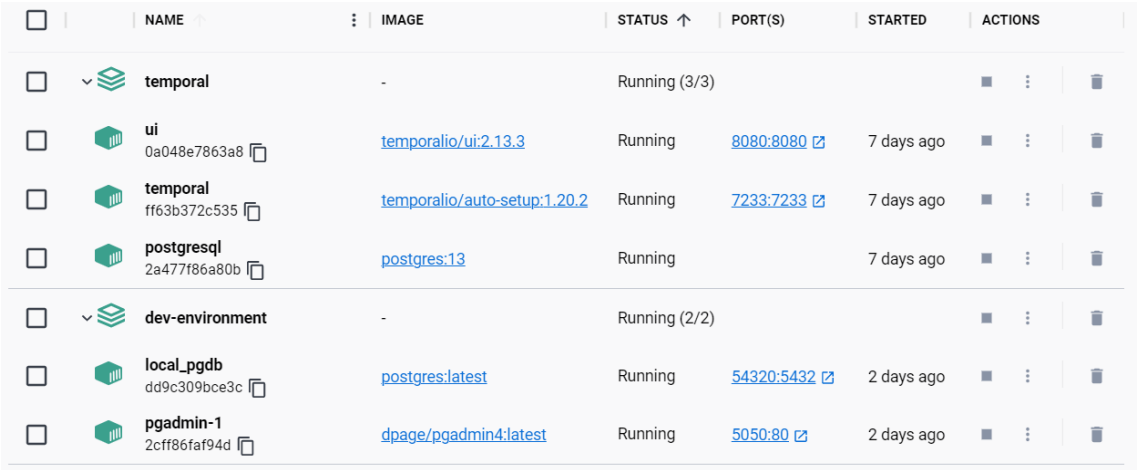
Docker on avatud lähtekoodiga platvorm, mis käivitab rakendusi ja muudab protsessi arendamise lihtsamaks. Dockeris ehitatud rakendused pakendatakse kõigi nendega kaasnevate sõltuvustega standardvormi, mida nimetatakse konteineriks. Need konteinerid töötavad isoleeritult operatsioonisüsteemi tuumal põhineval eksekutsioonikeskkonnas. Lisakihi abstraktsioon võib mõjutada tulemust jõudluse seisukohast [14].

Docker pakub võimalust automatiseerida rakenduste paigaldamist konteineritesse. Konteinerites virtualiseeritud ja käitatud rakenduste keskkonnas lisab Docker neile juurde paigaldusmootori lisakihi. Docker on loodud selliselt, et pakub kiiret ja kergekaalulist keskkonda, kus koodi saab tõhusalt käivitada ning samuti annab see täiendava võimaluse koodi võtmiseks arvutist testimiseks enne tootmisesse viimist [16]. Russell (2015) kinnitab, et Docker võimaldab testida oma koodi ja viia seda tootmiskeskkonda nii kiiresti kui võimalik [15]. Turnbull (2014) järeldab, et Docker on hämmastavalt lihtne [16].

Projektis kasutame Dockerit esmalt Temporali imagete loomiseks tööarvutisse, et seadistada lokaalne arenduskeskkond. Vastavad arenduseks mõeldud konteinerid on kättesaadavad Temporali kodulehelt või Visual Studio terminalis kasutades ‘npm install

temporal’. Selle rea abil paigaldatakse ja käivitatakse imaged Temporali serveri ja Temporali veebi UI jaoks.

Dockeri abil on projekti raames loodud ka arenduskeskkond, mille abil saab kergelt paigaldada ühe rea abil ettevõtte repositooriumist endale PostgreSQL andmebaasi, PGAdmini, projektiga seotud andmebaasid ja andmebaasifuntsioonid ning vastavad PGAdmini ja andmebaasivahelised serveriühendused, et projekti taasloomine ja katsetamine isiklikus arenduskeskkonnas oleks lihtsam ja ühtlane.



Joonis 46. Projekti toimimiseks vajalikud Dockeris töötavad imaged.

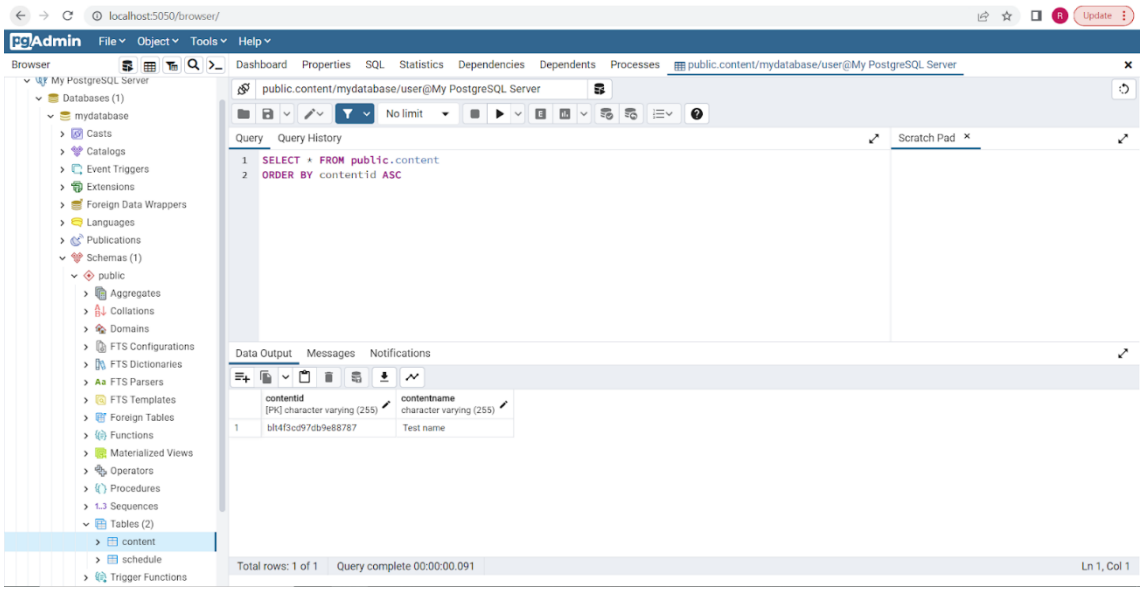
```yaml
version: "3.8"

services:
  db:
    image: postgres
    container_name: local_pgdb
    restart: always
    ports:
      - "54320:5432"
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
      POSTGRES_DB: mydatabase
    volumes:
      - ./tables-setup.sql:/docker-entrypoint-initdb.d/tables-setup.sql

  pgadmin:
    image: dpage/pgadmin4
    restart: always
    environment:
      PGADMIN_DEFAULT_EMAIL: email@email.com
      PGADMIN_DEFAULT_PASSWORD: password
      PGADMIN_CONFIG_SERVER_MODE: 'False'
      PGADMIN_CONFIG_MASTER_PASSWORD_REQUIRED: 'False'
      PGADMIN_SERVER_JSON_FILE: '/pgadmin4/servers.json'
    ports:
      - "5050:80"
    volumes:
      - './servers.json:/pgadmin4/servers.json'
      - local_pgdata:/var/lib/postgresql/data
    #TODO custom image for Node application using Dockerfile

volumes:
  local_pgdata:
  pgadmin-data:
```

Joonis 47. Kood Dockeri abil ühtlase arenduskeskkonna loomiseks.

content_scheduling_poc > temporal-hello > dev-environment > tables-setup.sql

```sql
CREATE TABLE Content(
  ContentID VARCHAR(255) NOT NULL,
  ContentName VARCHAR(255),
  PRIMARY KEY (ContentID)
);
CREATE TABLE Schedule(
    ScheduleID SERIAL NOT NULL,
    ContentID VARCHAR(255),
    StartTime timestamptz,
    EndTime timestamptz,
    PRIMARY KEY (ScheduleID),
    FOREIGN KEY (ContentID) REFERENCES Content(ContentID) ON UPDATE CASCADE
);

CREATE PROCEDURE AddContent(c_ID INOUT VARCHAR(255), c_Name VARCHAR(255))
LANGUAGE plpgsql AS
$$ BEGIN
INSERT INTO Content(ContentID, ContentName)
VALUES (c_ID, c_Name) RETURNING ContentID INTO c_ID;
END $$;

--CALL AddContent(NULL, 'Test Content');

CREATE OR REPLACE PROCEDURE AddSchedule(s_ID INOUT INT, c_ID VARCHAR(255),  s_StartTime TimeStamptz,  s_EndTime TimeStamptz)
LANGUAGE plpgsql AS

```

Joonis 48. Tabelite seadistamise fail, mis defineerib PostgreSQL tabelid ja funktsioonid.



Joonis 49. PGAdmin vaade, kuhu on Docker image'te abil automaatselt tehtud serveri seadistus ja loodud vastavad tabelid.

4 Tulemused

Meie arhitektuur hõlmab mitut erinevat süsteemi, kuna ettevõte kasutab siseselt traditsioonilist mikroteenuste arhitektuuri. Peamine kasutusjuht on Contentstack-is ajastatud sisu loomine. Arendasime CMS-ile kasutajaliidese, millega ajastatud avaldamise kohta andmeid sisestada. Avaldamise info saab kätte läbi meie loodud REST API. Lisaks sai loodud DSL avaldis, mis on CRON-i edasiarendus. Kasutasime *workflow*-de käitamiseks Temporali.

Loodud süsteem on POC ja võib öelda, et projekt õnnestus. Kui ettevõte saab edukalt hakkama veel mõne POC-iga, mis kasutab Temporali, siis on võimalik, et nad otsustavad Temporali lisada oma Kubernetese klasterisse. Kuna organisatsioonil on palju pikalt jooksvaid hajutatud töövooge, uurivad nad edasi Temporali integreerimist oma hajutatud süsteemidesse.

Meie töövoogude kood on töötav kood, aga suuremate projektide jaoks on vaja kasutada mikroteenuste disainimustreid. Järgmiseks etapiks ongi uurida *saga* mustri implementeerimist kasutades Temporali. *Saga* muster võimaldab rikke korral *activity*-le

teha *undo* ja koosneb koodist, mis viib läbi mingi transaktsiooni ja koodist, mis muudab selle transaktsiooni olematuks. See töötab ainult transaktsiooniliste operatsioonide korral ja ei lahenda kõiki probleeme, sest on olemas osa koodi, mida ei saa olematuks teha. Igaljuhul on see meie järgmine loogiline samm Temporali uurimiseks [17].

5 Järeldused

Kuigi Temporali kasutamine läbi mingisuguse API, meie juhul siis REST API, on turvalisuse koha pealt soovitatav, sest otse kliendi juurest ühenduse loomine Temporal serverisse ei ole üldjuhul turvaline ja vajab lisa meetmeid, siis on meie REST API puhul otseselt ebavajalik kasutada Temporali, kuna tegemist ei ole pikalt jooksva protsessiga. Ja me otsustasime et selle API puhul võib pöörduda otse andmebaasi poole, ilma Temporali vahekihti kasutamata.
Temporali kasutamise kohta meie peamise kasutusjuhu puhul, milleks on DSL ekstraktimine ja andmebaasi salvestamine, leidsime, et see tasub ennast ära, kuna tegemist on pikalt jooksva protsessiga.
Nagu eelmises peatükis mainitud, siis järgmise sammuna uurime me saga mustrit hajusrakendustes, kasutades Temporali.

6 Kokkuvõte

Projekti eesmärk on lisada Contentstack CMS-ile ajastatud avaldamise võimalus. Lisaks proovime seda probleemi lahendada kasutades Temporali. Temporal on platvorm, mis võimaldab luua vastupidavaid ja skaleeruvaid hajusrakendusi ja ei ole veel ettevõttes kasutusel.

DSL on loodud eesmärgiga maksimeerimaks funktsionaalsust, mida kasutajal võib vaja minna. Ehkki tihti võib minna vaja kõigest ajastatud avaldamist, milles on defineeritud täistundidega avaldamine, on meie lahendus veidi abstraktsem ja kasutatav rohkemates kohtades kui ainult selles projektis spetsiifiliselt.

Lahendus hõlmab CMSi kasutajaliidest ajastamise info sisestamiseks. Seejärel liigub see info ootejärjekorda (queue). Eraldi teenus võtab ajastamise info ja dokumendi tuvastmise koodi sealt ootejärjekorrast ning algatab workflow. Workflow sisestab avaldamise alguse ja lõpu ajad koos dokumendi koodiga andmebaasi. Andmebaasis olevat avaldamise infot on võimalik eraldi teenuse kaudu pärida.

Loodud süsteem on POC ja võib öelda, et projekt õnnestus. Kui ettevõte saab edukalt hakkama veel mõne POC-iga, mis kasutab Temporali, siis on võimalik, et nad otsustavad Temporali lisada oma Kubernetese klasterisse.

Temporali kasutamise kohta meie peamise kasutusjuhu puhul, milleks on DSL ekstraktimine ja andmebaasi salvestamine, leidsime, et see tasub ennast ära, kuna tegemist on pikalt jooksva protsessiga.

Kasutatud kirjandus

[1] B. Hoffmann, N. Urquhart, K. Chalmers, and M. Guckert, “An empirical evaluation of a novel domain-specific language – modelling vehicle routing problems with Athos - Empirical Software Engineering,” SpringerLink, https://link.springer.com/article/10.1007/s10664-022-10210-w (17.05.2023).

[2] M. Fowler and R. Parsons, *Domain-Specific Languages*. Upper Saddle River etc.: Addison-Wesley, 2011.

[3] “What is a headless CMS?,” contentstack.com, https://www.contentstack.com/cms-guides/headless-cms/ (17.05.2023).

[4] “Key concepts and hierarchy of Contentstack elements,” contentstack.com, https://www.contentstack.com/docs/overview/key-concepts-and-hierarchy-of-contentstack-elements/ (17.05.2023).

[5] R. Kolodiy, “How AWS pricing works: AWS pricing model & principles,” TechMagic, https://www.techmagic.co/blog/aws-pricing-model-overview/ (17.05.2023).

[6] “Whitepapers,” Amazon, https://docs.aws.amazon.com/whitepapers/latest/awsoverview/introduction.html (17.05.2023).

[7] S. Patterson, *Learn AWS Serverless Computing: A Beginner’s Guide to Using AWS Lambda, Amazon Api Gateway, and Services from Amazon Web Services*. Birmingham: Packt Publishing Ltd., 2019.

[8] J. P. Buddha and R. Beesetty, *The Definitive Guide to AWS Application Integration with Amazon SQD, SNS, SWF and Step Functions*. Berkeley, CA: Apress, 2019.

[9] I. H. V. Whitehouse-Grant-Christ, “IAM,” Amazon, https://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html (17.05.2023).

[10] M. Herrera, “Easily manage workflows at scale with Temporal.io and Astra DB,” DataStax, https://www.datastax.com/blog/easily-manage-workflows-at-scale-with-temporal-io-and-astra-db (17.05.2023).

[11] “Temporal courses,” temporal, https://temporal.talentlms.com/unit/view/id:2170 (17.05.2023).

[12] V. Khononov and J. Lerman, *Learning Domain-Driven Design: Aligning Software Architecture and Business Strategy*. Sebastopol, CA: O’Reilly Media, Inc, 2022.

[13] “Core Concepts, architecture and Lifecycle,” gRPC, https://grpc.io/docs/what-is-grpc/core-concepts/ (17.05.2023).

[14] C. Boettiger, “An introduction to docker for reproducible research,” *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 71–79, 2015. doi:10.1145/2723872.2723882

[15] B. Russell, “Passive Benchmarking with docker LXC, KVM & OpenStack”, 2015

[16] J. Turnbull, “The Docker Book: Containerization is the new virtualization“, 2014

[17] “Microservices pattern: Sagas,” microservices.io, https://microservices.io/patterns/data/saga.html (17.05.2023).

[18] “Temporal Cluster deployment guide,” https://docs.temporal.io/cluster-deployment-guide

Lisa 1 – Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks[1]

Mina, Ivar Osila ja Raoul Rocco Riigov

1. Annan Tallinna Tehnikaülikoolile tasuta loa (lihtlitsentsi) enda loodud teose „Ajastatud avaldamise funktsionaalsuse lisamine Contentstackile kasutades Temporali“, mille juhendajad on Viljam Puusep ja Jeno Laszlo
   1.1. reprodutseerimiseks lõputöö säilitamise ja elektroonse avaldamise eesmärgil, sh Tallinna Tehnikaülikooli raamatukogu digikogusse lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;
   1.2. üldsusele kättesaadavaks tegemiseks Tallinna Tehnikaülikooli veebikeskkonna kaudu, sealhulgas Tallinna Tehnikaülikooli raamatukogu digikogu kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.
2. Olen teadlik, et käesoleva lihtlitsentsi punktis 1 nimetatud õigused jäävad alles ka autorile.
3. Kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest ning muudest õigusaktidest tulenevaid õigusi.

17.05.2023

1 Lihtlitsents ei kehti juurdepääsupiirangu kehtivuse ajal vastavalt üliõpilase taotlusele lõputööle juurdepääsupiirangu kehtestamiseks, mis on allkirjastatud teaduskonna dekaani poolt, välja arvatud ülikooli õigus lõputööd reprodutseerida üksnes säilitamise eesmärgil. Kui lõputöö on loonud kaks või enam isikut oma ühise loomingulise tegevusega ning lõputöö kaas- või ühisautor(id) ei ole andnud lõputööd kaitsvale üliõpilasele kindlaksmääratud tähtajaks nõusolekut lõputöö reprodutseerimiseks ja avalikustamiseks vastavalt lihtlitsentsi punktidele 1.1. ja 1.2, siis lihtlitsents nimetatud tähtaja jooksul ei kehti.

Lisa 2 – Eneseanalüüs

1 Ivar Osila

Tehtud töö ülevaade: *backend* arendamine sh päringute ja tabelite loomine PostgreSQLis, *workflow*'de ja *activity*'te kood kasutades Temporali SDK-d, REST API kood TypeScriptis, dokumentatsioon kasutades Mermaidi. Lisaks eraldi teenuse loomine nö *consumer*ina, mis pollib SQSi, samuti TypeScriptis ja kasutades Node'i. Igapäevane aruandlus oma töö progressist ja osalemine koosolekutel.

Eneseanalüüs: õppisin TypeScripti, PostgreSQL päringuid ja funktsioone, Temporali, Mermaidi. Sain väikese ülevaate, kuidas reaalselt hajusrakendusi arendatakse ja kuidas näiteks SQSi kasutades saab sellise rakenduse arendust oluliselt lihtsustada. Sain osa ka koosolekust Temporali loojatega ja muust huvitavast.

2 Raoul Rocco Riigov

Tehtud töö ülevaade: DSLi ja selle parisimiseks vajaliku loogika välja töötamine. Contentstacki UI elemendi loomine. AWS teenustega töötamine (Lambda, SQS, IAM, API Gateway) ja nende integreerimine Contentstackiga. Dockerisse andmebaasiga seotud konteineritele täienduste tegemine. Samuti regulaarne koosolekutel osalemine.

Eneseanalüüs: Projekti käigus täienesid oluliselt teadmised hajusrakenduste ja Dockeri kasutamise kohta. Ühtlasi sai kätt proovitud mitmete AWS teenustega, millest võib tulevikus kasu olla. Tegin tutvust Temporaliga ja Contentstackiga, mis pruugivad samuti mängida lähitulevikus arendamisel suuremat rolli. Üldiselt oli tegemist esimese IT-alase suurema projekti loomise kogemusega ning huvitav oli osa saada arendusprotsessist. Täiendasin oskusi erinevate probleemide lahendamisel.