Thomas Krut
JAVAD22

Teknikfördjupning
2024-01-18

Microservices

med CQRS och Event Sourcing

Inledning

Detta projekt går ut på att studera några designmönster för microservicesarkitektur: *CQRS* (Command Query Responsibility Segregation) och *Event Sourcing*. Jag kunde valt ett av dem, då de inte egentligen är beroende av varandra, men dessa mönster passar bra ihop och förekommer ofta i samma lösningar.

CQRS går ut på att man delar upp en applikation i två delar: en del som hanterar kommandon (ny data och uppdateringar av data) och en som endast läser data. Kriterierna är att kommandon inte returnerar något (utom möjligtvis en statuskod), och att queries inte leder till några mutationer. Det skiljer sig från en vanlig CRUD-lösning där alla requests som regel går mot samma service och datamodell. Bygger man utifrån CQRS är man istället fri att använda helt skilda modeller för läsning respektive skrivning av data. Man kan också använda olika databaser och tekniker, samt skala varje del oberoende av den andra.

Event sourcing går ut på att data lagras som en sekvens av händelser. För att få ”current state” spelar man upp händelserna från början till slut. Händelser får aldrig tas bort eller muteras; för att rulla tillbaka operationer krävs att man registrerar nya kompenserande händelser. Eventloggen är alltid source of truth för applikationen, men olika delar kan välja att läsa olika händelser och ha egna ”states” för sina specifika ändamål.

Dessa två mönster går fint att kombinera. *Command*-delen av applikationen tar då emot kommandon som processas och ger upphov till händelser som sparas i eventloggen. *Query*-delen läser loggen och bygger olika vyer eller *read models* som sedan kan konsumeras av användare eller andra applikationer. Till skillnad från när en relationsdatabas används finns inte något gemensamt schema; varje read model kan konstruera sitt eget schema utifrån eventloggen med de samband som just den modellen behöver för sitt use case. En read model kan ha en egen dedikerad databas som passar dess behov, eller hantera sitt state in-memory. Loggen läses då i sin helhet eller från ett cachat snapshot varje gång applikationen startas; därefter behöver endast nya händelser läsas in för att uppdatera state vid behov.

Bakgrund

Termen CQRS går att spåra till Greg Young i början av 2010-talet[1] och är en vidareutveckling av det äldre begreppet CQS (Command Query Separation). CQS definieras enligt kriterierna för läsning och skrivning ovan. CQRS går steget längre och delar upp arkitekturen i separata objekt som ansvarar för läsning respektive skrivning.

1 https://www.eventstore.com/cqrs-pattern

Event sourcing är egentligen ett koncept med rötter längre tillbaks i tiden. Inom bokföring vill man ofta kunna spåra ändringar och se vilka transaktioner som gett upphov till nuvarande saldo. Redan i forntidens Egypten upprättade man sådana register på stentavlor och i 1500-talets Venedig blev det grunden för den moderna dubbla bokföring[2] som fortfarande är grunden för alla ekonomiska system. Idag används tekniken inom många områden där man inte bara är intresserad av *vad* datan består av utan *hur* den kom till. Den ligger också till grund för många verktyg som utvecklare använder dagligen: ångra-kommandot i IDE:n, versionshantering i Git – även en vanlig SQL-databas använder en transaktionslog under huven.

Syfte

Greg Young är noga med att påpeka att en vanlig, eller som han benämner den, *stereotypisk arkitektur* ofta är att föredra och att den fungerar utmärkt i 80% av fallen. Han menar då en traditionell lagerbaserad struktur där DTO:s (data transfer objects) hämtas och skickas fram och tillbaka mellan klient och applikationsserver, ofta med ORM-verktyg som sköter mappning mellan databas och objektmodell. Som regel är det i stort sett samma objekt som går åt båda håll, kanske med uppdaterade fält. Eventuella nackdelar med denna struktur blir uppenbara först i mer sammansatta domäner där mer affärslogik och beteenden behöver vara en del av själva datamodellen – i en CRUD-lösning handlar det bara om fyra verb: create, read, update och delete. De procedurer som utgör den faktiska affärslogiken låter sig sällan representeras i sådana termer.

Som exempel ger Young att ändra adressen på en kund – betyder *UpdateAddress* att adressen var fel och korrigerades, eller att kunden flyttat? Får svaret på den frågan konsekvenser i andra delar av systemet? Vilka steg måste i så fall utföras? Ofta finns dessa regler någon annanstans – på papper, eller i värsta fall i huvudet på de som använder systemet. Det kanske fungerar ändå, och kostnaden det skulle innebära att byta arkitekturmönster kanske inte väger upp nyttan.[3] Det kan finnas fördelar med att hålla isär datamodell och affärslogik, men motivationen att regler ofta ändras och att man vill slippa ändra i mjukvaran varje gång. Den mänskliga faktorn spelar då större roll, och frågan blir hur mycket det kostar när det blir fel.

I en CQRS-baserad arkitektur fångar man upp användarens intentioner genom att definiera kommandon som upprätthåller affärslogiken. I exemplet ovan hade man kanske haft kommandon som *CorrectAddress* och *RelocateCustomer*. Greg Young menar att om man inte reflexmässigt faller tillbaka på standardverben uppstår en diskussion om de faktiska use cases man behöver hantera vilken i sin tur leder till en djupare förståelse för hur domänen och dess premisser kan representeras i koden.

Den stereotypa arkitekturen har många fördelar. Det går snabbt för nya teammedlemmar att sätta sig in i projektet. Alla känner till den och har jobbat med den och den är sannolikt rätt val i många fall – faktum är att den är så generisk att den går att applicera var om helst. Därför blir

2 https://dev.to/dealeron/event-sourcing-and-the-history-of-accounting-1aah
3 https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf

den lätt ett automatisk förstahandsval. Varken CQRS eller event sourcing är lämpligt att använda i alla situationer. Noggrann analys av tradeoffs och ROI är nödvändig där de övervägs.

Några problem som som teknikerna hjälper till att lösa kan vara:

CQRS:

- Tillämpningar där infrastruktur för läsning och skrivning behöver kunna skala oberoende av varandra.

- Komplexa domäner, där man vill modellera objekt för skrivning och läsning på olika sätt och inte låsa sig till en gemensam relationell modell. Ofta eftersträvar man efter tredje normalformen när data ska skrivas och något som ligger närmare första normalformen när data ska läsas. Med CQRS kan man ha båda samtidigt, eller valfritt antal read models med olika struktur och underliggande teknik för lagring.

- När man har nytta av en gemensam vokabulär för de operationer som ska kunna utföras på systemet, och vill hålla dessa åtskilda från hur datan konsumeras.

Event sourcing:

- Tillämpningar där man vill kunna spåra ändringar och se hur applikationens tillstånd såg ut vid en given tidpunkt. Detta kan ha flera användningsområden[4]:

> - Felsökning, spårning av anomalier. Eventloggen kan spelas upp i en testmiljö med avsikten att härleda felaktig data till en viss kedja av händelser. Att debugga med hjälp av loggar är inget nytt men fördelen med en eventlogg är att händelser kan processas på nytt för att återskapa applikationens tillstånd i ett visst skede.
>
> - Analys av beteenden, mönster, samband. Att exponera vilka händelser som ägt rum i systemet för en viss användare eller entitet kan vara ovärderligt i exempelvis en supportsituation, eller där man försöker analysera hur användare navigerar genom systemet. Det finns naturligtvis andra tekniker för detta men med en eventlogg får man det mer eller mindre på köpet.
>
> - Möjligheten att i framtiden kunna plocka ut data för att tillgodose nya behov som är svåra att förutspå när systemet designas. Lägger man till ny funktionalitet processar man helt enkelt befintliga events med den nya koden.

4 https://martinfowler.com/eaaDev/EventSourcing.html

- När prestanda och skalbarhet är viktigt. En applikation vars enda lagringsmekanism är att skriva till slutet av en logg är lättare att skala horisontellt i form av ett distribuerat system, jämfört med en relationell databas.

- En komplex domän kan bli lättare att förstå och hitta ett gemensamt språk för om den översätts till beteenden och händelser. En händelse som ett obestridligt faktum att en viss sak inträffade är mindre öppet för tolkning än att prata om mer abstrakta fenomen som kan ha olika innebörd i olika områden av affärslogiken. Man slipper också ta hänsyn till det faktum att en relationell modell och en objektorienterad modell bygger på skilda paradigm och att en utvecklare som jobbar i systemet behöver vara mycket väl förtrogen med båda modellerna för att kunna fatta rätt designbeslut i olika situationer.[5]

Frågeställning

Jag undersöker i detta projekt hur en enkel applikation kan designas enligt mönster för CQRS och event sourcing. Detta för att förstå vilka konsekvenser det får i kod och vilken teknisk infrastruktur som behövs. Syftet är inte att applicera mönstret på ett problemområde där införandet är förenligt med en faktiskt affärsnytta – det är out of scope och onödigt komplicerat. Jag tar istället en enkel domän och introducerar en del komplexitet där den strikt talat inte behövs, för att på så sätt få en praktisk förtrogenhet med teknikerna och därmed en bättre grund att stå på inför framtida designbeslut. Min utgångspunkt är att det blir lättare att känna igen de fall där teknikerna är motiverade om jag faktiskt själv implementerat dem i en egen lösning någon gång. Det blir också en lägre tröskel om man någon gång ska jobba i ett projekt där dessa mönster förekommer.

Beskrivning av tekniker

Jag har valt att blanda tekniker något mer än vad som kanske är nödvändigt, för att få insyn i lite olika tillvägagångssätt. Min mikroservicesarkitektur bygger på REST-anrop från klienten, sedan en message broker (NATS[6]) som mina olika command-services prenumererar på. Dessa anropar i sin tur en event store-service över gRPC – detta för att få en gemensam uppsättning typer för events som flera services kan använda, och för att eventdatan ändå behöver serialiseras i någon form när den lagras. gRPC[7] är ett protokoll för att serialisera och överföra data i binär form (som ett mer effektivt alternativ till JSON) och möjliggör användandet av en plattformsoberoende syntax (*protobuf*) för definitioner av typer och metoder. Verktyget genererar kod för server och klient i olika programmeringsspråk som kan användas för tätare kontrakt och kommunikation (*remote procedure calls)* mellan tjänster.

5 https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf
6 https://nats.io/
7 https://grpc.io/

Min event store-service använder en instans av Eventstore DB[8] för att lagra events. Eventstore DB är en dedikerad databas för event sourcing utvecklad av bland andra Greg Young, mannen bakom begreppet CQRS. Det hade också gått utmärkt att använda en enkel SQL-databas med en tabellrad per event, och en kolumn för sekvensordningen. Eventstore DB tillhandahåller dock ett smidigt API för att läsa events från exempelvis ett visst sekvensnummer.

Alla services på command-sidan skriver jag i Go. Query-delen av min applikation kommer utgöras av ett API skrivet i Java och Spring boot med Thymeleaf-templates för representationer av ett par olika read models. Denna service kommer kommunicera direkt med Eventstore DB för att läsa eventloggen vid uppstart och sedan hålla aktuellt state in-memory. Vid anrop kommer den kontrollera om nya events kommit in och i så fall endast processa dessa.

Alla services kommer köra som docker-containers med en volume för databasen och definieras i en docker-composefil.

Metod

Som redan konstaterats är det inte rimligt att hitta ett use case för detta projekt där CQRS faktiskt vore det rätta valet av designmönster, då det framförallt är motiverat i fall med komplexa applikationer med krav på prestanda och skalbarhet. Event sourcing går att se nyttan med i ett bredare spektrum av tillämpningar, men även där skulle troligtvis den applikation jag väljer att bygga här ha blivit mindre och enklare om jag valt en ORM-lösning eller använd någon SQL-databas.

För att besvara min frågeställning har jag valt en simpel programidé i form av ett bokningssystem för någon form av turistanläggning. Systemet ska hantera beläggningen på anläggningen över tid, man ska kunna skapa, ändra och ta bort bokningar, checka in och ut gäster samt kunna få en grafisk representation i ett webbgränssnitt. Utmaningen blir att separera ut logik för läsning och skrivning av denna data, att lagra den i form av en eventlogg, och se vilka konsekvenser detta får när funktionaliteten ska implementeras.

Marknadsanalys

Enligt JetBrains *State of Developer Ecosystem*[9] rapporterar 34% av tillfrågade utvecklare att de jobbar med microservices. Av dessa uppger 12% att de använder CQRS som designmönster. Detta är en ökning med 3% under de senaste tre åren enligt samma undersökning. Vid en snabb genomsökning av tjugotalet jobbannonser inom utveckling på LinkedIn nämns microservices i runt hälften, och CQRS i två.

8 https://www.eventstore.com/
9 https://www.jetbrains.com/lp/devecosystem-2023/

CQRS används ofta för mindre delar av system och inte för övergripande arkitektur. Det är definitivt inte ett mönster som går att tillämpa på alla problemområden; tvärtom bör det noga övervägas och bara implementeras där det inte introducerar onödig komplexitet. Jag har inte lyckats hitta några specifika exempel på stora företag eller organisationer som använder just CQRS men då microserviceslösningar är såpass vanliga förekommer det rimligtvis med en viss frekvens. Enligt Greg Young, mannen bakom begreppen CQRS, är tekniken vanligt förekommande inom finans och gambling[10]. Även event sourcing används ofta i sammanhang där pengar och kassaflöden hanteras, då man enkelt kan spåra orsaken till ändringar (*audit trail)*

Microservices har varit på uppgång ett tag men också kommit att ifrågasättas som universallösning. En del större aktörer som Amazon Prime fått stor uppmärksamhet när de gått tillbaka till en motolitlösning och sparat miljardbelopp på sänkta driftskostnader[11]. Kurvan för trenden har nog planat ut. Dock är det ett faktum att microtjänster löser många problem med skalbarhet, tillgänglighet och flexibilitet och att den i många fall utgör rätt val av teknik. Val av designmönster är nästa steg och även där måste det handla om ett noga övervägt beslut. CQRS kommer behövas så länge det finns komplexa datamodeller som kan vinna på en uppdelning mellan läsning och skrivning.

Resultat

Jag har valt att bygga ett enkelt bokningssystem för exempelvis ett hotell eller en camping. En enkel kravspecifikation för en MVP ser ut som följer, med uppdelning enligt commands och queries:

Commands	Queries
- boka en enhet för en viss tidsperiod - ändra en bokning - ta bort en bokning - checka in eller ut en gäst	- se alla bokningar i en kalendervy - se aktuella in- och utcheckningar för dagens datum

För att tydliggöra skillnaden mellan *commands* och *events* är det viktigt hur vi uttrycker oss. Ett kommando är en uppmaning eller en önskan om att något ska hända – entiteten som tar emot kommandot kan utföra åtgärden, eller svara att det inte är möjligt. Ett kommando kan med andra ord misslyckas. Ett event å andra sidan är ett konstaterande att något har inträffat. Det går inte att göra ogjort utan en tidsmaskin[12]. Det går inte att ändra eller radera i efterhand.

10 https://www.youtube.com/watch?v=Q97BWNCkBt8
11 https://www.infoq.com/news/2023/05/prime-ec2-ecs-saves-costs/
12 https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf

Denna distinktion hjälper oss att förstå hur applikationen behöver byggas. För att hålla det tydligt är det enligt Greg Young extremt viktigt hur commands och events formuleras rent språkligt: *commands* i verbets *imperativ*, och *events* i *imperfekt/dåtid*. Använder man samma verbform i båda fallen – man kallar exempelvis både command och event för *CreateCustomer* – behöver den som arbetar i systemet hålla distinktionen i minnet och risken för fel ökar. Att vara explicit gör det hela mycket tydligare: kommandot heter CreateCustomer och eventet heter CustomerCreated.

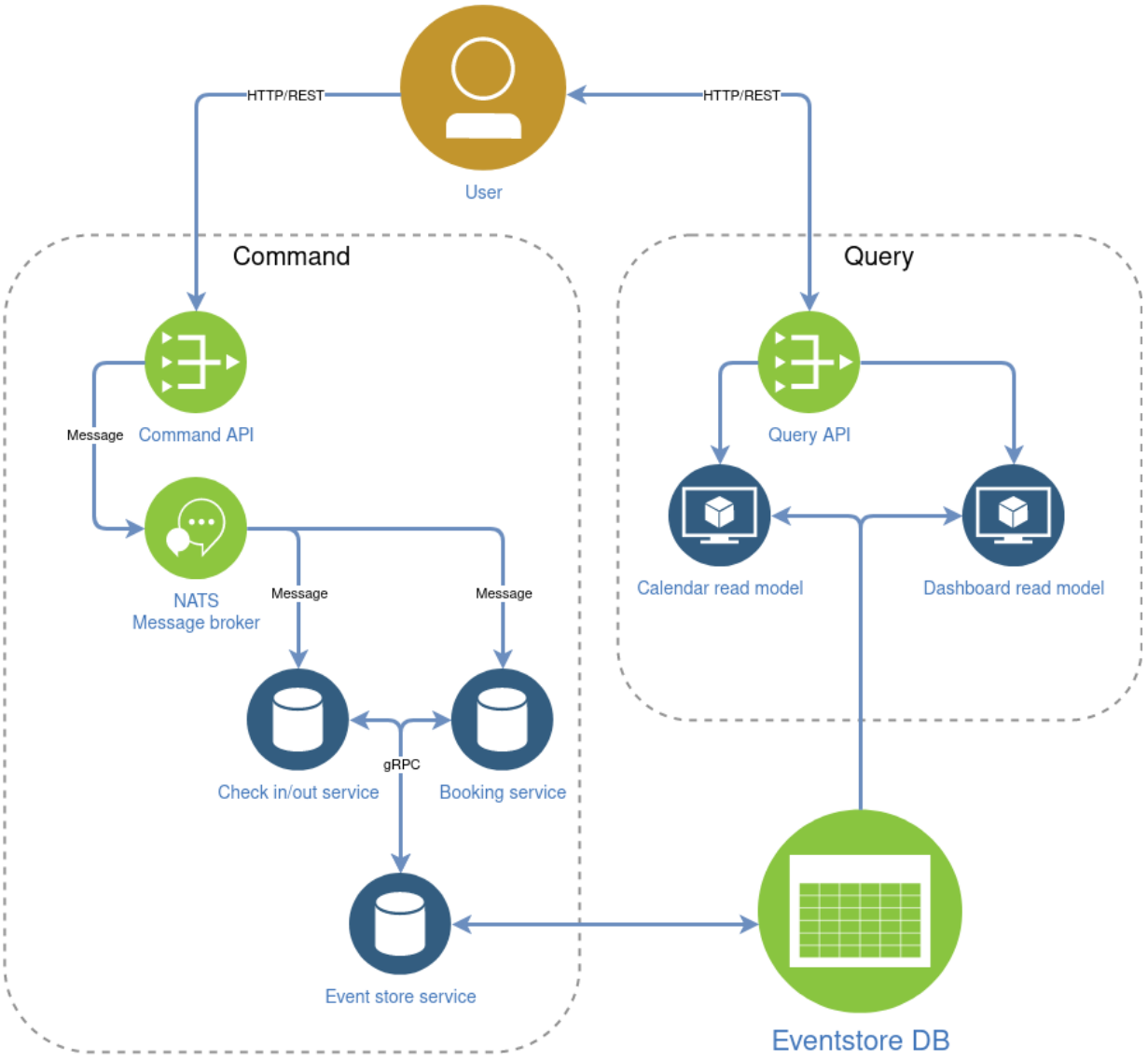
Min applikation har nedanstående endpoints på command-sidan av API:et, som ger upphov till motsvarande commands i form av meddelanden till min message broker, och som sedan – om kommandot gick att utföra – lagras som events i loggen:

API endpoint	Command	Event
POST /booking	command.booking.create	BOOKING_CREATED
UPDATE /booking?id	command.booking.update	BOOKING_UPDATED
DELETE /booking?id	command.booking.delete	BOOKING_DELETED
POST /booking/checkin?id	command.booking.checkin	BOOKING_CHECKED_IN
POST /booking/checkout?id	command.booking.checkout	BOOKING_CHECKED_OUT

En lösning utan CQRS hade kanske nöjt sig med de tre första metoderna, och lagt checkedIn och checkedOut som ett fält på själva bokningen att ändra med en UPDATE. Genom att specificera hur systemet faktiskt används och definiera kommandon på detta sätt får vi ett naturligt ställe att hantera logiken för in- och utcheckningar, nämligen i den microservice som processar dessa kommandon.

Request, command och event har som payload den data som behövs för att beskriva eventet. Det rör sig om boknings-ID och i de fall där det behövs även start- och slutdatum för bokningen, rumsnummer, antal gäster och ett namn. Bokningen tilldelas ett unikt ID, och vid skapande av event skrivs metadata till ett speciellt fält med ett timestamp. Där hade man även kunna lägga information om vem som skapat bokningen. Sekvensnummer behöver vi inte bry oss om då detta sköts automatiskt av Eventstore DB. Eventtypen (kolumnen längst till höger ovan) är en enum som definieras i min protobuf-fil.

På nästa sida finns ett diagram över systemets komponenter.



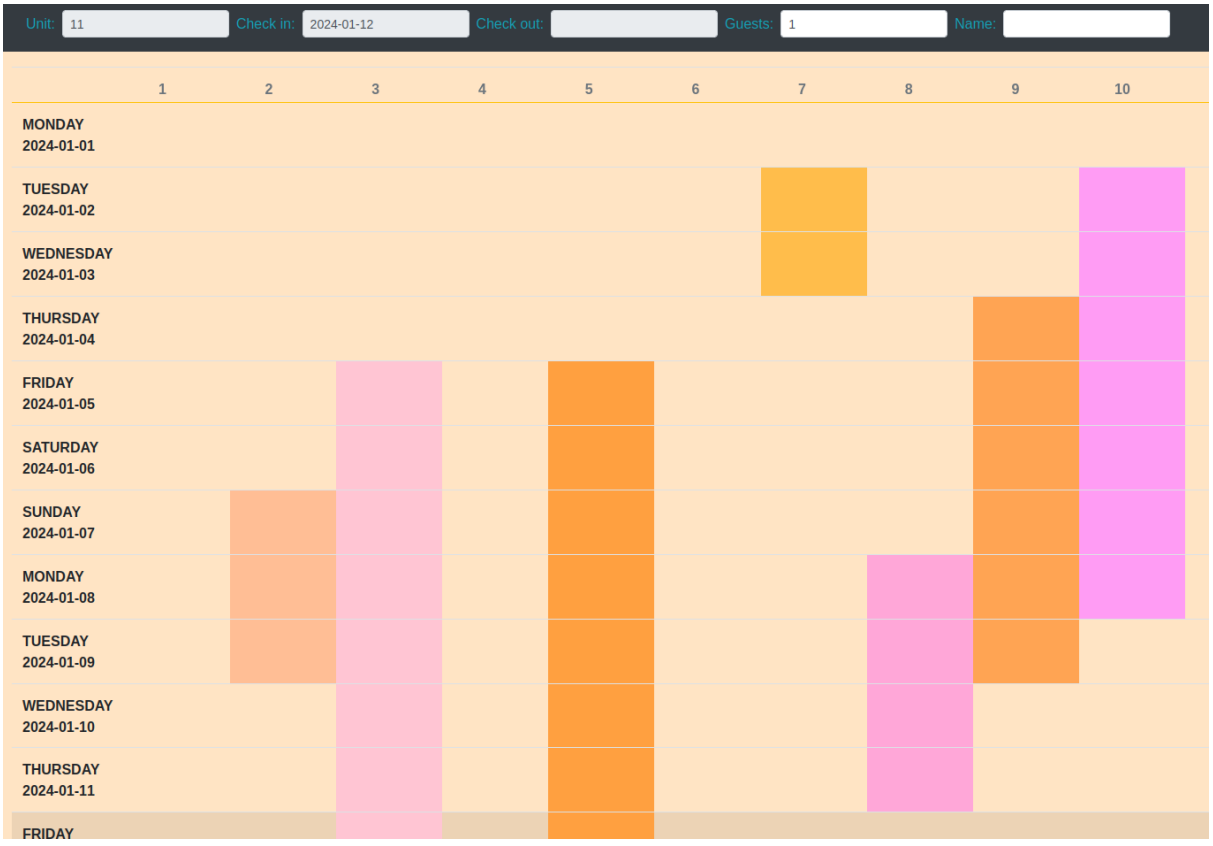
Det finns två tjänster som processar kommandon – check in/out-service och booking-service. Dessa behöver hålla reda på applikationens *current state*: en bokning kan inte skapas om det redan finns en bokning som överlappar valda datum, en gäst kan inte check in om bokningen inte finns, och så vidare. Varje service läser därför in event-loggen när den startar, och skapar sedan listor och hashmaps med objekt för att hålla reda på den data den behöver. Strukturen på dessa optimeras för den typ av valideringar servicen behöver utföra – exempelvis är tjänsten för in- och utcheckning endast intresserad av de bokningar som finns för innevarande datum och håller därmed enbart dessa i minnet. Booking-service behöver å andra sidan inte veta något om vilka gäster som checkat in eller ut.

För att current state ska vara konsekvent med loggen är det nödvändigt att events alltid appliceras på samma sätt, oavsett om de kommer från eventloggens historik eller från ett nytt kommando som processats. Applikationen själv vet inte skillnaden. Events är det enda som

kan ändra state hos en service. Om instansen går ner och startar upp igen kommer loggen läsas i sin helhet och återskapa exakt samma datastruktur som fanns tidigare. Om ett kommando processas men av något anledning inte kan skrivas till loggen kommer inte heller current state att kunna ändras, då det skulle leda till inkonsekvens och dataförlust. Vi returnerar status 503: Service Unavailable i dessa fall och klienten får försöka igen senare.

Även gällande mina read models behöver state hållas up-to-date, och principen är densamma. Här läser jag nya händelser från eventloggen varje gång en ny query kommer in. API:et skickar helt enkelt senast kända sekvensnummer (*revision* på event store-språk) till Event Store DB, får tillbaka eventuella events med större sekvensnummer, och uppdaterar senast kända index till det större talet.

Min read model för kalendervyn ligger till grund för en Thymeleaf-template som renderar en kalender med ifyllda rutor för bokade datum:

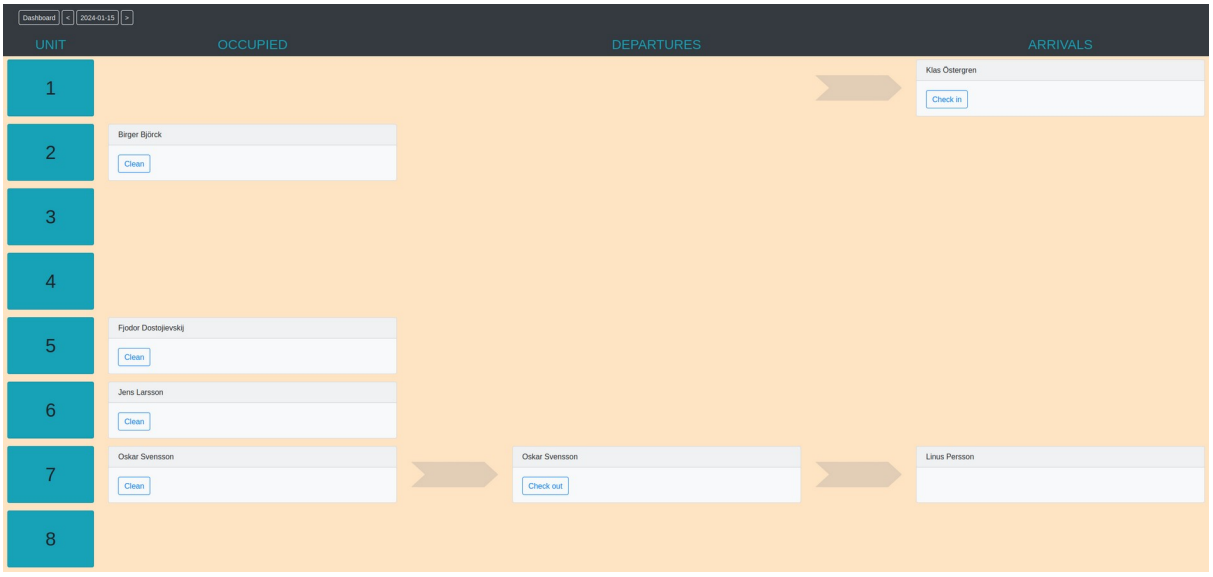


Man kan även välja rader i en kolumn för att skapa en ny bokning, eller klicka på en befintlig bokning för att se bokningsinformation eller ta bort den.

I fallet med kalendervyn är det smidigt att kunna utgå från eventloggen och sedan skapa en icke normaliserad datamodell som passar min template. Där sparar jag samma bokningsobjekt på varje dag som bokningen sträcker sig över. Dubbellagring av data är inte ett bekymmer då datan endast läses och aldrig muteras på query-sidan av ett segregerat

system. Jag drog även nytta av en read models förmåga att tillhandahålla rätt data för ett use case genom att på servern spara namn på veckodagar och månadens första dag på varje bokning, vilket sedan representeras grafiskt i vyn. Tack vare det slapp jag hantera datum i Javascript på klienten, och kom undan med lite AlpineJS[13] för göra min DOM interaktiv. Viktigt att poängtera att eventuella interaktioner mot systemet här, som till exempel nya bokningar, går mot command-API:et och inte mot samma endpoint som tillhandahåller vyn.

Min andra read model är en simpel dashboard för att hålla reda på dagens städ, in- och utcheckningar (kommandon för att städa rum hanteras dock inte av systemet i övrigt):



Precis som i fallet med check in/out-services är det endast dagens datum som är intressant, så modellen hämtar bara den data som behövs. Dubbellagring är inte heller ett problem; state består av hashmaps med boknings-ID som key för bokningar som ska checkas in, checkas ut och städas och samma bokning kan förekomma på flera ställen då inga mutationer sker här.

Slutsatser

På det hela taget var det trivsamt att jobba med en event store och CQRS. Ingen schema för entiteterna behövdes för att komma igång, endast en idé om vilka händelser som ska förekomma i systemet och några enkla domänobjekt. Att slippa skriva queries för joins, uppdateringar och valideringar mot befintlig data i en relationell databas var också behagligt. Ett ORM-verktyg skulle tagit hand om mycket av det men det hade blivit ytterligare en komponent i ekosystemet att förstå och förhålla sig till. En eventlogg är en synnerligen enkel datastruktur att ha i bakgrunden.

Att i varje service sedan läsa loggen på det sätt som möjliggjorde aktuell funktionalitet var ganska trivialt. Ett ordentligt batteri med enhetstester var dock nödvändigt för att säkerställa

13 https://alpinejs.dev/

att eventen påverkade state på rätt sätt. Mönstret med *table driven testing*[14] i Go var mycket användbart för att täcka in alla tänkbara varianter och jag ligger på ungfär 80% coverage i dessa services. Att varje microtjänst har sin egen datamodell innebär en del kognitivt overhead och context switching och där sparade enhetstesterna mycket tid och tankemöda när det var dags att refaktorera eller göra ändringar.

Användandet av en message broker i applikationen visade sig inte ha särskilt stor inverkan på implementationen av de aktuella designmönstren. Jag skulle lika gärna kunnat göra http-anrop mellan tjänsterna, men NATS var ett lättviktigt verktyg som knappt krävde något setup alls och jag slapp en del boilerplate för att skicka och ta emot anrop. Jag kunde fokusera på logiken och flödet, istället för på hur tjänsterna kommunicerade.

En intressant aspekt som jag fått tillfälle att fundera över i samband med projektet är hur viktigt språkbruket är när man definierar ett problemområde. Våra fyra CRUD-verb ger en enkel inkörsport i vilket sammanhang som helst men duger ofta inte som kompetent modell för ett riktigt business case. Det blir mest en samling getters och setters och mer komplexa händelseförlopp låter sig inte modelleras. Områden att läsa vidare inom är Task Based User Interfaces, Domain Driven Design och Behavior-driven Developement. Här har jag verkligen bara skrapat på ytan.

github.com/thomaskrut/tekf

14 https://dave.cheney.net/2019/05/07/prefer-table-driven-tests