Indholdsfortegnelse

Calendar 2.0

"The course has extensively increased my knowledge of the subject. During the course I have received sufficient amount of feedback from my teacher. My overall impression of the course is positive."

Completely disagree? *Neither agree nor disagree*? Eller *completely agree*? Således bliver en studerende på Copenhagen Business School (CBS) bl.a. spurgt om til en evaluering af hvert fag i slutningen af hvert semester. Et tidspunkt, hvor de har eksamenerne hængende over hovedet og måske føler, at det er ligegyldigt at give feedback for, hvad der kunne have været gjort bedre, da det ikke længere vil være aktuelt for de studerende selv. Uanset hvor meget CBS forsøger at pleje dette problem, så vil den feedback studerende giver, aldrig kunne nå at blive effektueret overfor dem. Spørgsmålet er derfor – Hvad kunne CBS implementere for at kunne give de studerende en følelse af, at der var et '*trade-off*' ved at give feedback? Dette er problemet *project_eva* forsøger at løse.

project_eva forsøger at øge funktionaliteten af de studerendes online kalender applikation, ved, udover at vise en oversigt for kommende lektioner, at give muligheden for at komme med feedback til læreren allerede fra den første lektion. Dette foregår således, at det vil være muligt at kunne trykke ind på den enkelte lektion og bl.a. kunne bedømme, kommentere og se andres feedback til lektionen. Naturligt, kommer portalen således også til at give læreren mulighed for at logge ind og se den feedback, der er givet og på den måde kunne lytte til sine studerende og korrigere lektionerne løbende. Således vil kvaliteten af lektionerne kunne øges, og CBS ville opleve en langt mere omfattende og motiveret evaluering af deres kurser. Men er *project_eva* et system, CBS 'bare' kan implementere? Er det opbygget med en professionel tilgang, der tager højde for de problemstillinger, der typisk forekommer ved implementering af distribuerede systemer, såsom skalerbarhed, sikkerhed og andre væsentlige tematikker? Og hvad med fremadrettet, skulle det blive til et reelt projekt, der fik mere tid til at blive udbygget? Hvad skulle der til for at forbedre systemets svagheder? Skulle der tages nogle fundamentale ændringer ift. de valg, der er blevet taget under det tidspres, systemet er udviklet under?

Denne opgave forsøger at besvare disse spørgsmål ved tematisk at gå igennem, hvilke valg der er blevet taget gennem projektet, hvad produktet af disse valg er blevet, og hvilke ændringer og valg man kunne forestille sig at tage fremadrettet.

Netværk & arkitektur

project_eva kunne blive implementeret i CBS' system på to vidt forskellige måder. Det kunne blive fuldt integreret i deres nuværende system, måske endda erstatte *www.calendar.cbs.dk*, eller det kunne blive kørt som et sideløbende projekt. Valget her bestemmer hvordan den egentlige arkitektur kommer til at være for systemet. Systemet er i øjeblikket opbygget med en three-tier arkitektur, hvor vi har klientsiden som *presentation tier*, serveren som *application tier* og en udviklet database i DBMS systemet MySQL som *data tier* til at lagre lektioner, brugere og bedømmelser. Skulle systemet implementeres i CBS' nuværende system, så ville databaseimplementeringen ikke kunne bruges til andet end mulig inspiration til ændringer og tilpasninger i CBS' egne databaser. Derfor vil opgaven fremadrettet betragte en fremtidig implementering af systemet som et sideløbende projekt, som også kommer til at koble en dedikeret database til systemet.

Den fysiske model

Tanker om den fysiske model er i grunden fuldstændig forsømt under udviklingen af *project_eva*. Systemets frontend er udviklet i JavaScript og *markup*-sproget HTML, serveren er udviklet i Java og databasen med DBMS-systemet MySQL. Endvidere er alle tre *tiers* testet og udviklet på samme lokale miljø, hvilket er skylden i de manglende tanker om den fysiske model. De eneste tanker, der er forekommet kan opsummeres til, at man har regnet med, at server og database kommer til at blive kørt gennem CBS' nuværende løsning – om det er ekstern server-hosting, eller om de selv har servere stående. Klienten er her udviklet således, at det kører gennem et systems browser, og derfor kan den fysiske model for klienter forekomme som computere, tablets eller mobiltelefoner. Man kunne forestille sig, at CBS benytter sig af en *load-balancer* udover en *proxy*, da *concurrency* bestemt er noget, de skal tage hånd om med 22.829 studerende, der benytter deres portaler (CBS, 2016). Systemet ville helt klart finde en nytte i dette, da det kommer til at skulle håndtere den samme trafik.

Heldigvis er bedømmelserne af lektionerne noget der skal forekomme løbende gennem hvert semester, hvilket betyder, at der vil være en stabil samtidighed af brugere gennem halvårene, modsat portaler som eksempelvis *www.eksamen.cbs.dk*, hvis trafik kunne forestilles at være mere situationsbestemt eksempelvis i perioder med eksamener.

Den arkitektoniske model

At systemet skulle blive løst-koblet og modulært har været et centralt drejepunkt under udviklingen. Eksempelvis er klassen `DBWrapper.class` på serveren udviklet til at stå for at generisk kunne opbygge SQL-statements uden at være afhængig af, hvilket DBMS-system *project_eva* skal benytte. Derved kan man hurtigt oprette dedikerede klasser til andre DBMS-systemer end MySQL, som der er i det nuværende og på den måde gøre systemet løst-koblet på dette led. Endvidere har det også været et mål at gøre systemet tilgængeligt til brugere uafhængigt af platformen de befinder sig på. Dette har bl.a. været et af grundene til, at klienten er skrevet i JavaScript, da browsere har det som en norm at have en JavaScript *interpreter* indbygget. På den måde er klientsiden gjort platformuafhængig, da det så er browseren der står for at *interprete* og udføre JavaScript-koden på en brugers enhed.

Interoperabiliteten mellem klienten og serveren er ligeledes udviklet med en løst-koblet tilgang. Forbindelsen imellem forekommer ved at serveren udstiller et REST API service, der gør det muligt at kommunikere med andre systemer gennem simple HTTP metoder, da forbindelsen forekommer over internettet, og JSON er formatet for dataudvekslingen. Da programmet består af en simpel funktionalitet, der egentlig kun har behov for CRUD-operationer på server-siden og klientsiden er ønsket at være tilgået gennem browsere, ville en videreudvikling stadig holde fast i at forsøge at være en RESTful web service. Dette fordi at SOAP, som er det andet typiske alternativ, er et ældre framework, der er mere tungt og måske først kunne have været aktuelt, hvis klienten krævede mere komplekse løsninger på serveren end blot CRUD-operationer.

Afslutningsvis, er det forsøgt at gøre klienten så tynd som muligt ved at forsøge at holde logikken på serverlaget, og kun lade klienten stå for præsentationen af den indkomne data. Dette er gjort med

henblik på at holde klienten så 'let-vægtig' som muligt, da meningen med applikationen er, at studerende hurtigt skal have muligheden for at kunne gå ind og bedømme deres time på stedet og uafhængigt af deres enhed. Selvom der ikke forekommer de største udregninger i systemet, så er et godt eksempel at hive op, hvor beregningerne af de forskellige statistikker for de enkelte lektioner foregår. Disse beregninger kunne have foregået på klienten, når lektionerne blev modtaget, men i stedet er de sat til at blive beregnet på serversiden og blive kaldt fra klienten som simple `Integer`- og `Double`-værdier, når der blev trykket på en lektion. Da det kan formodes, at anvendelsen af klienten vil forekomme hurtigt og kort og med en tilstedeværende internetforbindelse i anvendelsens øjeblik, kunne man dog godt have gjort klienten endnu tyndere ved eksempelvis at have haft implementeret mere *lazy loading*, ved kun at hente lektioner ned for den konkrete uge man stod i, når man loggede ind. På den måde ville det indledende login foregå mere optimeret, da der ikke skulle downloades et halvt semesters lektioner ned hver gang. Lektionerne kunne så blive downloadet løbende, skulle en bruger navigere igennem ugerne i kalenderen. På den anden side kunne der også argumenteres for, at systemet som helhed, faktisk var mere optimalt med en tykkere løsning. Med den computerkraft man kan have helt nede på telefon-niveau, kunne det tænkes, at i stedet for at serveren skulle stå for udregninger for alle samtidige studerende i systemet, så distribuere dele af arbejdsbyrden til klienterne, for at effektuere en implicit *loadbalancing*, der betød en mere troværdig server. Her burde der aktivt blive udarbejdet en analyse for, om et *trade-off* mellem brugernes bekvemmelighed kontra mulig server-stabilitet, for at finde frem til den mest optimale løsning.

Den fundamentale model

project_eva er et distribueret system, der anvender den asynkrone interaktionsmodel i form af AJAX-forespørgsler til kommunikationen mellem klient og server. Dette er fordi, at klienten er udviklet som en hjemmeside, hvor en *non-blocking* interaktionsmodel er at foretrække ved det meste af funktionaliteten, da man som bruger skal have muligheden for at agere som ønsket. Endvidere ville det ikke være intuitivt som bruger at skulle vente på et svar fra serveren ved hvert forekommende kald til den. Det er aktivt valgt også at gøre kaldet til serveren ved log ind asynkront, da et formål var, at brugeren ikke skulle opleve nogen form for *blocking* i vedkommendes oplevelse på siden. Dog er det implementeret således, at den asynkrone funktionalitet ikke får systemet til at logge ind først, før

loginet overhoved er blevet verificeret, men at det er muligt at navigere videre rundt på hjemmesiden i stedet for. Løsningen forekommer dog således, at klienten blot skifter HTML-side, så snart et bekræftet *callback* kommer tilbage. Dette betyder, at man som bruger kan gå op og skifte URLen manuelt fra `../index.html` til `../user.html` og komme ind i viewet for en bruger der er logget ind, uden at systemet ved hvem du er, eller hvilke lektioner den skal hente ned for dig. Logikken bag verificeringen og skiftningen af viewet, skal derfor omstruktureres og udbygges således, at dette ikke ville være muligt, skulle *project_eva* realiseres.

At systemet skal kunne håndtere fejl, var et af kravene i kravspecifikationen. Det var tiltænkt for at implementere en fejlmodel der havde en mere korrekt og omfattende tilgang til fejlhåndtering, end blot at lave `try/catch`-statements, de steder en fejl kunne opstå i koden. Dette endte dog alligevel med at ske, og en stillingstagen til en fejlmodel blev negligeret. Systemet implementerer kun fejlhåndtering ved processer, og det er egentlig primært *fail-stop*- og *crash*-fejlhåndtering, hvor en `exception` bliver smidt, og fejlen bliver logget. Systemet tager slet ikke hånd om fejl der kan have noget at gøre med forbindelsen, hvad der kan ske med data undervejs i forbindelsen eller andre aspekter af fejlmodellen. Ligeledes bliver en bruger på klientsiden ikke altid informeret, hvis en fejl opstår, da det flere steder blot er implementeret til at smide en `exception` i konsollen, skulle der opstå noget. Derved skulle en dybere implementering af fejlhåndtering bestemt forekomme i det videre forløb, da det i det nuværende blot er meget overfladisk.

Sikkerhed har været et vitalt emne at have i bagtankerne under forløbet, og derfor vil sikkerhedsmodellen blive uddybet i næste kapitel. For at opsummere, så er det tydeligt, at der mangler mange overvejelser ift. systemets fysiske og fundamentale model, modsat arkitekturen, hvor der i udviklingsforløbet har været mange overvejelser om. Dog er det også tydeligt, at disse overvejelser mangler en omstrukturering, pudsning eller videreudvikling, hvis systemet bliver udviklet videre.

Kryptografi

project_eva arbejder med sensitiv data. Måske er selve fag-oplysningerne ikke sensitive (hvilket også kan forklare, hvorfor CBS' egen API står til rådighed uden kryptering), men en studerendes adgangskode er, og ligeledes er de personlige kommentarer og bedømmelser, de hver især kan skrive, da systemet lover anonymitet. Derfor har det været vigtigt at tage nogle foranstaltninger under udviklingen af systemer, der adresserede dette.

Hashing

Hashing er den mest typiske foranstaltning for at øge sikkerheden af et password. Når man hasher et password, så forvandles det, ved hjælp af nogle algoritmer, til en vilkårlig streng man ikke kan vende tilbage fra. Her kan der bruges forskellige algoritmer såsom MD5 eller de forskellige SHA-versioner, hvor MD5 blev valgt i dette system. Efter implementeringen af MD5 og *code-freezen* i forløbet var nået, fandt man ud af, at det ikke var det bedste valg at anvende MD5, da algoritmen i dag (siden 2005) egentlig ikke betragtes særlig sikker (Sotirov, A. et al, 2008). Ritter, T (2010) opsummerer de adskillige angreb man kan anvende mod MD5. Da algoritmen er gammel, findes der eksempelvis massevis af *rainbow-tables*, ubudne gæster kan anvende til såkaldte *dictionary attacks*. Dernæst er længden af hash-værdien for MD5 *kun* 128 bits, hvor der findes SHA-algoritmer der har helt op til 512 bits og endda giver muligheden for en vilkårlig længde. Endvidere er det også vist, at MD5 er sårbar og ikke længere *'collision resistant'*, som betyder, at det har lykkedes at finde to vilkårlige meddelelser(strenge), som bliver hashet til den samme værdi. (Selinger, P. 2006)

Det blev dog besluttet at beholde MD5-algoritmen, da sikkerheden sås tilstrækkelig ift. hvor sensitivt og interessant data, systemet arbejder med, er for eksterne personer. Algoritmen er implementeret således, at der både forekommer en hashing af passwordet på serversiden såvel som på klient-siden. Klient-siden implementerer det for, at en såkaldt *man-in-the-middle* (MitM) ikke kan "lytte" på en forbindelse og opsnappe et password i *plain-text*, da han her i stedet ville få den hashede værdi. Serversiden gen-hasher så dette password, før det bliver sendt til databasen. Grundlaget for dette er, at serveren har afgrænset sig fra omverdenen og ekspliciteret indgangene en klient kan have til

den og på den måde afgrænset resten af systemet fra omverden. Dette betyder, at ved at hashe "bag lukkede døre", tilføjer det et ukendt element for MitM og ubudne gæster, der ikke har mulighed for at vide, hvad der sker bag endpointsne. Endvidere bliver der tilføjet et ekstra lag på begge sider, ved at passwordet bliver sammenkædet med et *salt*, der tilføjer en tilfældig streng til passwordet før det bliver hashet – både på klient- og serversiden. *Saltets* formål er at gøre det langt mere besværligt at kunne gætte sig til passwordet. En MitM kan prøve at køre flere *rainbow-tables* igennem for at finde frem til en persons password, men hvis vedkommende ikke ved, at der er en vilkårlig længde streng adderet til passwordet, eller godt ved det, men ikke ved hvordan *saltet* bliver genereret, bliver det straks langt svære at gætte korrekt.

Ovenstående er de tanker, der er gjort ved systemets implementering af MD5 *hashing* med et *salt*. Men udover at have haft brugt en anden mere sikker algoritme som eksempelvis SHA-2, hvad kunne der så have været udviklet anderledes? Og hvor kan der opstå sikkerhedsbrister?

Ingen effekt ved saltet

Formålet med at modarbejde mulighederne for *rainbow-tabellerne* og længden af den tid det ville tage at finde frem til det rigtige password, har været en god tanke. Det er dog blevet implementeret på sådan en måde, der ophæver denne selvsamme ønskede effekt. Da *saltet* er *hardcodet* på en klient-side, der står åbent til rådighed for hele internettet, kan en MitM selv åbne hjemmesiden og kigge på de filer, der bliver downloadet ned i browseren, når websiden forsøges åbnet. Her ville han kunne kigge script-filen `MD5Hasher.js` (Coyier, C. 2011) igennem og finde *salt*-værdien, der bliver tilføjet på passwords. Derved kan han blot addere *saltet* til hans *rainbow-tabeller*, hvilket derved netop ugyldiggør *saltets* formål. Vedkommende behøves heller ikke at have kendskab til, hvordan det bliver hashet på serveren, og om det sker med et *salt* eller ej, da han blot igen står over for 'bare' at skulle ramme det rigtige password.

Krypteringen af den data (hermed også det hashede password) bliver naturligvis krypteret, så det ikke bliver læsbart for en eventuel MitM. Dog er implementeringen her ligeledes heller ikke særlig sikker, hvilket bliver uddybet i det følgende afsnit. Men forestiller vi os, at en MitM får dekrypteret

den data der bliver sendt til serveren, (hvilket bestemt er muligt) så kan han egentlig blot *replaye* forespørgslen med vedkommendes brugernavn og hashede password til et endpointet, der autoriserer en bruger, og få adgang på et hvilket som helst tidspunkt.

Ser vi bort fra, at krypteringen kan sikre dette, så kan det bl.a. løses ved at gøre hashingen unik for hver enkelte person. Eksempelvis kunne man implementere det således, at systemet tager vedkommendes mail og addere det sammen med passwordet. Men dette ville kun have en ekstra effekt indtil man fandt frem til, hvad det var der blot skete ved hashingen. Derfor kunne man tildele et tilfældigt-genereret salt på serversiden ved registrering af en ny bruger, og så gemme saltet i databasen sammen med brugeren, og på den måde randomisere hashingen på individets niveau, men samtidig kunne huske saltet. Dette udsætter dog systemet til en ny trussel – da det kan være, at det er databasen, nogen fik adgang til, da saltet så ville stå hardcodet sammen med det hashede password, hvilket igen ugyldiggør saltets formål. Derfor kunne det sidste niveau man tog være, at man udover tildele unikke salt, også dele saltene op i to og addere dem på hver sin side af det hashede password, sådan så det så ud som følgende: *(1/2 salt) + (hash(password + salt)) + (2/2 salt)*. Således betyder det, at hvis en hacker ikke har adgang til logikken på serveren, så er chancerne for at kunne regne ud, for det første hvad det er der sker, minimale, da det umiddelbart for ham, blot er et hashed password. For det andet skulle han efterfølgende regne ud, hvordan saltet er blevet tilføjet til hashværdien, og for det tredje, hvor langt saltet er og hvor meget af værdien er saltet på begge side (hvis han skulle have regnet ud, at det adderes på begge sider). Dette er hvert fald en løsningsovervejelse man kunne implementere for at sikre en mere sikker tilgang til hashing af de studerende passwords, hvilket derved leder os videre til kryptering. Udover *replaying*, kan systemet stå overfor andre typer af angreb, som det skal kunne beskytte sig i mod. Det kunne være alt fra *tampering* og *masquerading* til *vandalism*. Hvordan beskytter systemet sig mod sådanne slags angreb, og hvordan kunne det udbygges?

Kryptering

Den data der bliver sendt mellem klient og server i *project_eva* er krypteret med krypteringsalgoritmen XOR, som serveren implementerer gennem Javas eget Security-bibliotek og klienten gennem

JavaScript filen `XORCipher.js` stillet til rådighed af Weaver, D. (2014). En streng der skal agere som en nøgle til at kryptere data står skrevet i `config.json` filen på serveren, mens samme nøgle står hardcodet i XORCipher scriptet på klientsiden, som netop er blevet konfigureret så den kunne læse en hardcodet nøgle. Systemets to sider kryptere og dekryptere meget simpelt den data der kommer ud og ind på begge sider, og så bliver der egentlig ikke foretaget flere foranstaltninger for systemets sikkerhed.

Dette er naturligvis ikke den mest sikre løsning. Som Weaver advarer i hans script, så er XOR en algoritme der er nem at knække. Endvidere ville en implementering af SSL helt klart være at foretrække, hvilket CBS også gøre brug af når deres Single Sign On service dukker op, når man prøver at tilgå et af deres portaler, der kræver autentifikation.

Som det kan ses i bilag A, så anvender CBS en anden krypterings algoritme ved navn AES_256_GCM og RSA som den asymmetriske nøgle-udvekslingsmetode. SSL (herunder RSA og TLS) fungerer således, at når en bruger tilgår en HTTPS-forbindelse, eller en klient der har HTTPS implementeret, så sendes der et certifikat til klienten fra serveren, der verificerer serverens integritet, og samtidig fortæller klienten, hvilken krypteringsalgoritme der skal anvendes sammen med den 'offentlige nøgle' (også kaldet *public key*), en klient skal kryptere sin data med. Det sikre ved den asymmetriske nøgle-udvekslingsmetode er, at dét der krypteres med en *public key* kan kun dekrypteres med en *private key*, som en server holder skjult hos sig selv. Derved bliver det ligegyldigt om en MitM får opsnappet en data-forsendelse eller selv får adgang til en *public key*. Han vil ikke kunne dekryptere uden den tilhørende *private key*. Når det så er, at forbindelsen bliver etableret mellem en klient og en server, så genererer klienten en symmetrisk nøgle, som egentlig kaldes et *session-ID*, som bruges til at identificere den specifikke klient (Bickel, J., 2014). Dette session-ID krypteres med den tildelte public key, hvilket netop sikrer fortroligheden af den indtil den når serveren. Efterfølgende bruges dette session-ID som den symmetriske nøgle for begge parters krypterings- og dekrypteringsnøgle for efterfølgende datapakker.

Ved at implementere SSL som autentikationssystem og krypteringsalgoritme, ville systemet kunne gardere sig langt bedre mod sikkerhedsbrister som *masquerading* og *replaying*, specielt når der også inkluderes et session-ID, der udløber efter en session er overstået, derfor gør det umuligt at *replaye* forbindelsen. Det løser dog også et andet sikkerhedsproblem, der ikke er blevet adresseret endnu og som er markant tilstedeværende i *project_eva*. Åbenheden af APIet og implementeringen af JavaScript med manglen af session-IDs som verificering, gør tampering og vandalism en aktualitet der under alle omstændigheder bør tages hånd om.

Lad os eksempelvis tage et kig på endpointet `/api/review/delete` for at slette et review. Som klient trykker du ind på en lektion og kan, udover dit eget, se andre folks kommentarer, hvor du har mulighed for at slette dit eget, jf. bilag B. Har en person dog lidt styr på JavaScript og får lyst til at lege med scriptet bag klienten, så kan han i realiteten gå ind og ændre *value*-værdien for slette-knappen som kan ses på det første billede. Det er udviklet sådan, at knappen får hardcodet reviewets ID, for 'nemt' at få fat i det specifikke ID ,der er nødvendigt at sende til serveren. Da der så mangler en ekstra form for verificering, et session-ID eksempelvis, på serverens side til at verificere en forespørgsel på sletning af et review, kan enhver bruger gå ind og slette alle kommentar og bedømmelser der overhoved er oprettet i systemet, hvilket klart er en sikkerhedsbrist. Ved at have implementeret et session-ID, som sendes med i forespørgslen, og som serveren først skal verificere, kan serveren hurtigt finde ud af, om klienten med det specifikke session-ID har autoritet til at slette det ønskede review.

Det er også klienten, der verificerer om et af de indkomne reviews til en lektion tilhører den bruger, der er logget ind eller ej. Er der det, så bliver review-feltet *disabled*, så en person ikke kan skrive flere kommentarer. Man kan så på samme måde gå ind og *enable* review-feltet og skrive så mange kommentarer, som man har lyst til. Endvidere har serveren heller ikke nogen form sikkerhedskontrol, skulle dette *breach* bruges til vandalisme. Her kunne der eksempelvis være blevet lavet en sikkerhedskontrol, der holdte et session-ID op mod en begrænsning af forespørgsler for at kunne 'slukke' for et session og herunder en klient, der opførte sig unormalt. Eller i værste fald, nu når sessions ikke er tilstedeværende i systemet, kunne blokere for en klient, der kommer fra en IP adresse, der

handlede ligeledes. I øvrigt kan en klient også starte fejl på serveren ved blot at lave en apostrof i kommentarfeltet, da dette forårsager fejl i SQL-statementet, når det forsøges registreret i databasen. Derfor er systemet højst sandsynlig også tilbøjelig for SQL-injections.

Det er således tydeligt, at *project_eva* har haft forsøgt at implementere nogle sikkerhedsforanstaltninger, der har skulle forestille at gøre systemet sikkert. Men en dybere analyse viser, at dette slet ikke er tilfældet, skulle systemet ligge sig ud med en person, der havde den mindste forståelse for, hvordan distribuerede systemer fungerede.

Skalerbarhed

Skalerbarhed er et af de andre vitale aspekter, der definerer et distribueret system. Et distribueret system skal kunne skalere og vokse. Dog er dette kun muligt, hvis man aktivt har banet vej for det. Derfor er skalerbarhed naturligvis været med i ens overvejelser under udviklingen af *project_eva*, dog til den grad, som det har været muligt på lokalmiljøet.

Windows, macOS, iOS og Android. Disse er fire af de mest anvendte operativsystemer der finder sted hos forbrugere. I kombination med forskellige programmeringssproge og implementeringsmuligheder på disse platforme, skabes der en heterogenitet, der er svær at imødekomme bredt som udvikler. Som nævnt tidligere, er dette et af grundene til, at klienten er udviklet i JavaScript. Endvidere implementeres der en responsiv løsning af klienten, hvilket fjerner de fleste barriere, der forekommer ved denne heterogenitet.

Som der også er blevet adresseret tidligere, så er den store fokus på at gøre systemet løst-koblet fra hinanden en sikring mod fremtidig skalerbarhed. Eksempelvis skal der ikke meget til at gøre systemet brugbart i kombination med andre DBMS-systemer der anvender SQL, sådan som DBWrapper klassen er udviklet. Endvidere, er alle hardcodede variable for database-tabeller og -kolonner placeret i config.json filen, for at gøre det muligt ikke kun at ændre DBMS-systemet, men også tilpasse systemet til personlige database-implementeringer, der er anderledes end den der medfølger. Dette

gør altså hele database-aspektet fleksibelt og ombytteligt. Derudover er der også implementeret *internationalization (i18n),* som gør det muligt at lave en oversættelse af serveren til et nyt sprog, skulle systemet også skaleres internationalt.

Serveren er udviklet ved at gøre brug af Jersey biblioteket, der har gjort det muligt at implementere et RESTful API, hvor Jersey selv sørger for at *spawne* threads fra thread-pool'en til hver forespørgsel der indsendes til serveren. Dette betyder, at håndteringen af samtidigheden af brugere på systemet er overladt til en ukendt, men professionel tilgang. Herved kan det på det lokale miljø fremstå som en god løsning, men dog slet ikke fungere som ønsket, når systemet skulle skaleres til CBS' studerende, da der er ingen kendskab til hvor fleksibelt og skalerbart, Jersey er. Det har generelt været svært at teste skalerbarheden af *project_eva*, da det hele er udviklet på et lokalt miljø. Dog har der været situationer, hvor det har været synligt, at selv lokalt var der nogle fundamentale fejl, der blokerede for samtidige brugere. Eksempelvis var `MYSQLDriver.class` klassen bygget op på sådan en måde, hvor det var samme instans af `Connection.class` klassen der blev brugt på alle threads, da den var instantieret som en global og statisk variabel. Dette blev løst ved at instantiere en 'connection' i begge metoder og derved gøre dem lokale, så de blev instantieret ved hvert thread, der skulle interagere med databasen.

Skulle systemet implementere de førnævnte sessions og altså en session-baseret autentikation, ville det betyde, at disse skulle lagres i databasen i et bestemt stykke tid, før de så skulle destrueres. Disse session-IDs kunne eksempelvis forestilles at være strenge på 20-30 tegn. Forestiller man sig så en andel af de 22.829 studerende, der samtidigt krævede en session, betyder det, at der skulle genereres og destrueres en masse fra databasen, hvilket kun ville blive mere og mere omfattende, som systemet skalerede. Derfor ville det være at foretrække at benytte et token-baseret autentikationssystem, hvor en bruger blot får genereret et token ud fra en fast algoritme, der gør brug af brugerens informationer, som serveren så selv kunne validere op i mod før nogen form for funktionalitet blev udført fra kaldet. På denne måde skal der ikke lagres ekstra data på databasen, og derfor ville serveren forekomme langt mere skalerbar og fleksibel, da dette token kunne bruges i hvilken som helst sammenhæng og tidspunkt, når denne bruger skulle verificeres (Hoax, 2015).

Afslutningsvis ville spørgsmålet være om CBS skulle foretage en horisontal og/eller vertikal skalering, hvis *project_eva* blev idriftsat som et sideløbende projekt. Dette er svært at tage stilling til, da der ikke er noget kendskab til CBS' servere og implementering. Dog kunne man forestille sig, at de har fremtidssikret sig ved at have nok servere til at opfylde behovet for en horisontal skalering, og derved, hvis der skulle forekomme nye og større workloads, at opgradere og skalere deres servere vertikalt.

Konklusion

project_eva implementerer et system, der giver CBS' studerende mulighed for at give lærere feedback via bedømmelser og kommentarer helt nede på hver enkelte lektions niveau. Dette gøres ved at have udviklet en webapplikation, de studerende kan få adgang til gennem deres enheders browsere. Her kommer de ind på en portal, hvor de kan logge ind og se deres skema og endvidere give denne feedback. Systemet er forsøgt at blive udviklet med en løst-koblet tilgang, der gør dets komponenter udskiftelige og derved fleksible ift. at adaptere sig til andre systemer – eksempelvis hvis det skulle arbejde sammen med CBS' nuværende IT-systemer. Under udviklingen er det forsøgt at gøre systemet sikkert med implementering af både kryptering og hashing. Dog er der analyseret frem til, at løsningen langt fra er den ideelle. Den ideelle løsning ville være at iværksætte SSL protokollen mellem server og klient for at maksimere sikkerheden og distancere en mulig MitM. I kombination med dette, forekommer der mange sikkerhedsbrister i den manglende verificering i hver forespørgsel i serverens API. Dette betyder, at systemets integritet let kan ryge i vejret, hvis folk med IT-kendskaber forsøgte at lege med koden på klient-siden. Udviklingen af *project_eva* har været med formål at få et dybere indblik i distribuerede systemer, og hvilke aspekter og problemstillinger disse indebærer. Nogen aspekter er mere gennemtænkte end andre, og nogen er måske helt negligeret – specielt de aspekter der fokusere på hardware-delen af distribuerede systemer. Derfor skal en fremadrettet udvikling tage hånd om den fysiske model og tilhørende netværk, såvel som hvordan transaktioner har en rolle i et uddannelsessystem, der kommer til at interagere på dagligt niveau med omtrent 23.000 mulige brugere.

Litteraturliste

Hjemmesider

- Sotirov, A., Stevens, M., Appelbaum, J., Lenstra, A., Molnar, D., Osvik, D.A. & de Weger, B., 2008. *MD5 considered harmful today* [Online]. [Besøgt d. 12. dec. 2016]. Findes på: http://www.win.tue.nl/hashclash/rogue-ca/
- Ritter, T., 2010. *Reasons why SHA512 is superior to MD5* [Online]. [Besøgt d. 12. dec. 2016]. Findes på: http://stackoverflow.com/questions/2117732/reasons-why-sha512-is-superior-to-md5
- Copenhagen Business School, 2016. *Nøgletal, rapporter og regler* [Online]. [Besøgt d. 14. dec. 2016]. Findes på: http://www.cbs.dk/cbs/noegletal-rapporter-regler
- Selinger, P., 2011. *MD5 Collision Demo* [Online]. [Besøgt d. 15. dec. 2016]. Findes på: http://www.mscs.dal.ca/~selinger/md5collision/
- Bickel, J., 2014. *SSL TLS HTTPS process explained in 7 minutes* [Online]. Besøgt d. 15. dec. 2016]. Findes på https://www.youtube.com/watch?v=4nGrOpo0Cuc
- Hoax, 2015. *Session Authentication vs Token Authentication* [Online]. Besøgt d. 15. dec. 2016]. Findes på http://security.stackexchange.com/questions/81756/session-authentication-vs-token-authentication

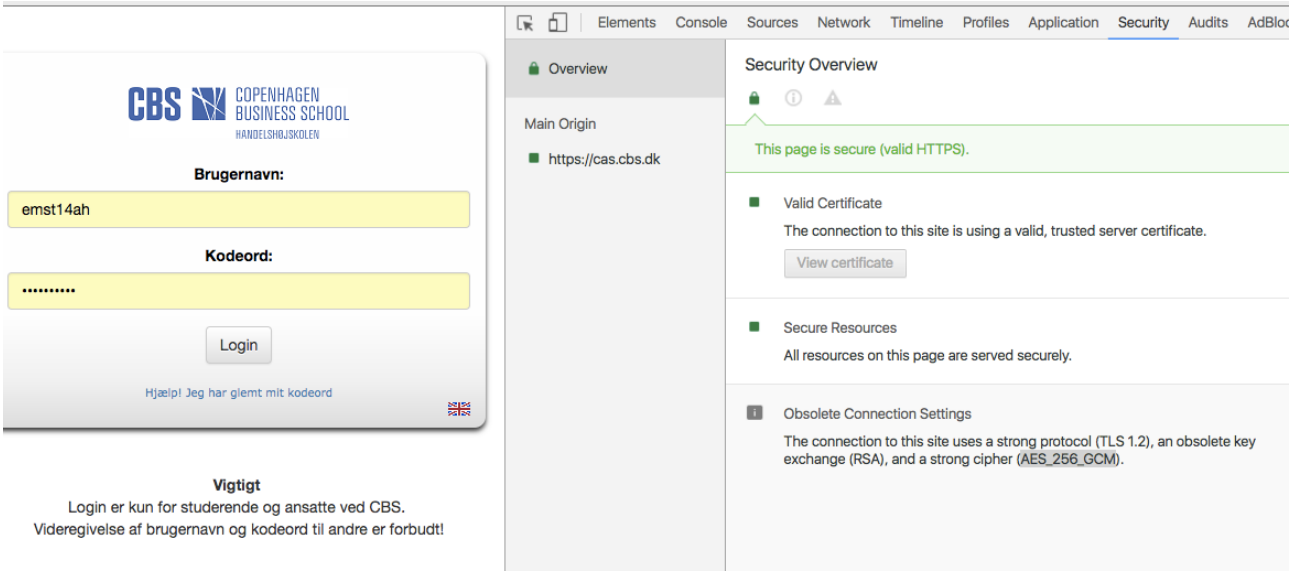
Kode-kilder

- Coyier, C., 2011. *JavaScript MD5* [Online]. Anvendt i fil MD5Hasher.js. [Besøgt d. 4. dec. 2016] Findes på: https://css-tricks.com/snippets/javascript/javascript-md5/
- Weaver, D., 2014. *XORCipher.js* [Online]. Anvendt i fil XORCipher.js. [Besøgt d. 4. dec. 2016] Findes på: https://gist.github.com/sukima/5613286
- W3Schools, *Bootstrap Modal Plugin* [Online]. Anvendt i fil user.html. [Besøgt d. 1. dec. 2016] Findes på: http://www.w3schools.com/bootstrap/bootstrap_modal.asp
- Suthar, L., 2014, *Simple Comment Box using Bootstrap 3* [Online]. Anvendt i fil user.html. [Besøgt d. 1. dec. 2016] Findes på: http://codepen.io/magnus16/pen/buGiB

- Shaw, A., 2016, *fullcalendar.css & …print.css* [Online]. Anvendt i filerne customfullcalendar.css & customfullcalendar.print.css. [Besøgt d. 28. nov. 2016] Findes på: https://fullcalendar.io/
- Start Bootstrap., 2016. *Grayscale v3.3.7+1* [Online]. Anvendt i filerne index.html, user.html, grayscale.js, grayscale.css & grayscaleUser.css. [Besøgt d. 22. nov. 2016] Findes på: http://startbootstrap.com/template-overviews/grayscale
- Visweswaran, K., 2016. *Bootstrap-Star-Rating* [Online]. Anvendt i filen customstar-rating.css. [Besøgt d. 1. dec. 2016] Findes på: http://plugins.krajee.com/star-rating
- Hansen, J.B., 2016. *Javascript-client* [Online]. Anvendt i filen sdk.js. [Besøgt d. 22. nov. 2016] Findes på: https://github.com/Distribuerede-Systemer-2016/javascript-client

Bilag

Bilag A – CBS' Single Sign On



Bilag B – Tampering/vandalism