Figur 1. Datasäkerhet. Från [17].

Kryptering av persondata för ökad säkerhet mot dataintrång

Kandidatarbete för Data- och Informationsteknik

Jonatan Berko
Julius Andersson Hopf

Institutionen för Data och Informationsteknik
CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET
Göteborg, Sverige 2020

Examensarbete

Kryptering av persondata för ökad säkerhet mot dataintrång

Jonatan Berko

Julius Andersson Hopf

CHALMERS
UNIVERSITY OF TECHNOLOGY

Institutionen för Data och Informationsteknik
CHALMERS TEKNISKA HÖGSKOLA
GÖTEBORGS UNIVERSITET
Göteborg, Sverige 2020

Kryptering av persondata för ökad säkerhet mot dataintrång
Jonatan Berko, Julius Andersson Hopf

Handledare: Sakib Sistek, Chalmers Tekniska Högskola
Examinator: Jonas Duregård, Chalmers Tekniska Högskola

Kandidatarbete 2020
Institutionen för Data och Informationsteknik
Chalmers Tekniska Högskola
SE-412 96 Göteborg

Institutionen för Data- och Informationsteknik Göteborg 2020

Abstract

This report describes the development of an application devised to perform dynamic searches on an encrypted database and measuring and analyzing the transaction times of different volumes of data in said searches.

The challenges of dynamic searching on an encrypted database are extensive. There is no way to make comparisons between encrypted fields without decrypting the data first which in turn increases the transaction time of every search.

During the project a full stack application was developed in three parts. A client layer where a user can perform searches on the encrypted database and have them presented as decrypted data, a server layer which handles communication between the client and the database and finally a database layer where the encrypted data is stored and cryptographic operations are performed. To avoid decrypting the entire database at every search, methods were used to only decrypt specific columns and therefore decreasing the transaction time.

The aims of the project were achieved, however, when volume testing, a considerable increase in transaction time was noted when comparing the results of the tests made on an encrypted database against the ones made on a decrypted one. Further steps can be taken to increase the security of the application itself such as securing keys, securing the system against possible attacks, and the usage of certificates issued by a trusted authority.

Keywords: REST API, Cryptography, TLS, PostgreSQL, Spring Boot, AES, Security, GDPR

Sammandrag

Denna rapport beskriver utvecklandet av en applikation vars syfte är att utföra dynamiska sökningar i en krypterad databas samt mätning och undersökning av transaktionstider för olika volymer data vid dessa sökningar.

Problematiken kring dynamisk sökning på krypterad data är omfattande. Om datan är krypterad går det inte att göra jämförelser mellan olika data utan att dekryptera den först, vilket medför ökningar i transaktionstid vid varje sökning.

Under projektets gång utvecklades en fullstack-applikation i tre delar. Ett klientlager där en användare kan utföra sökningar i databasen och få den presenterad okrypterad, ett serverlager som kommunicerar mellan klienten och databasen och slutligen ett databaslager där den krypterade datan lagras och kryptografiska operationer utförs. För att inte behöva dekryptera hela databasen vid varje sökning användes metoder för att endast dekryptera specifika kolumner i databasen vid sökning och därmed minska tidskostnaden.

Projektets alla mål uppfylldes men vid volymtestning antecknades stora ökningar i tidskostnad för sökning i en krypterad databas kontra okrypterad. Det finns även ytterligare steg att ta i applikationens egna säkerhetsaspekter så som skyddande av nycklar, skydd mot eventuella attacker och användande av certifikat utfärdade av en betrodd auktoritet.

Nyckelord: REST API, Kryptografi, TLS, PostgreSQL, Spring Boot, AES, Säkerhet, GDPR

Förord

Först och främst vill vi tacka Alex Crayvenn och Lars-Olof Strid för all hjälp och vägledning med vårt arbete, men även alla goda råd de gett oss för efter examen. Vi vill också tacka vår handledare Sakib Sistek vars positiva anda och goda samtal har varit en ljuspunkt under dessa märkliga tider.

Ett extra tack till Aleksander, Johan, Mellard, Sam och Tomas för ert stöd och kamratskap under dessa tre åren.

Terminologi och förkortningar

API: Application Programming Interface. Ett gränssnitt som utvecklare använder för att upprätta kommunikation mellan två fristående applikationer och skapa ett utbyte av funktioner och tjänster. Kommer vanligtvis i form av bibliotek eller HTTP-tjänst

CA: Certificate Authority. En betrodd tredje part som säkerställer certifikats och nycklars äkthet.

CRM: Customer Relationship Management. Bred term som beskriver arbete med kundrelationer. Exempelvis kundstöd.

CRUD: Create-Read-Update-Delete. Funktioner som kan användas för att skapa, läsa, uppdatera och hämta data.

Endpoint: I ett API representerar en endpoint en ände av kommunikation. Vid anrop mot ett API är det endpoints som används för kontakt. En endpoint kan exempelvis vara en specifik URL.

GDPR: General Data Protection Regulation är en förordning som reglerar behandling och flöde av EU medborgarnas personuppgifter både inom och utanför EU.

GUI: Graphical User Interface. Ett grafiskt verktyg för att underlätta kommunikation mellan användare och bakomliggande tjänster.

HTTP: Hypertext Transfer Protocol. Protokoll för datakommunikation på internet.

IV: initialization Vector. Ett slumpmässigt framtaget tal som används i kombination med en nyckel för att kryptera data.

JDK: Java Development Kit. En Open Source-miljö för utveckling i Java och innehåller kompilator, klassbibliotek etc.

JSON: Ett kompakt, textbaserat format som används för att utbyta data.

MAC: Kryptografisk bit information som används för att autentisera ett meddelande

ORM: Object Relational Mapping. Ett system som konverterar databastabeller till objektorienterade klasser.

OSI: Open Systems Interconnection model. En modell som beskriver de 7 lager som innefattas i datorkommunikation där varje lager tillhandahåller en specifik tjänst.

PKCS 12: Public Key Cryptography Standards nummer 12. Ett filformat för lagring av kryptografiska objekt som nycklar och certifikat.

POJO:	Plain Old Java Object. Ett javaobjekt som implementerar väldigt få interfaces och ärver från väldigt få superklasser.
Proof-of-concept:	En utvecklad prototyp av en idé i liten skala för att bevisa att idén är genomförbar
Query:	En query är en begäran för data från en eller flera tabeller i en databas. Vanligtvis för hämtning men också för andra CRUD-operationer för manipulering av data.
RDBMS:	Relational Database Management System. Programvara för att hantera data i en relationsdatabas.
SQL:	Structured Query Language. Ett språk för att hantera, skicka begäran mot och manipulera data i en relationsdatabas.
X.509:	Standardtyp för certifikat för asymmetriska nycklar. Används vanligen för TLS/SSL vid upprättning av HTTPS.
XML:	Extensible Markup Language. Ett universellt märkspråk som används för att koda dokument som kan läsas både av människa och maskin.

Innehållsförteckning

1 Introduktion

1.1 Bakgrund

Både den privata och offentliga sektorn arbetar kontinuerligt med att utveckla standarder och protokoll med fokus på att skydda individens personuppgifter och integritet, som även är kärnan i dataskyddsförordningen (GDPR). Dock skapar tillväxten av uppkopplade tjänster nya utmaningar och ställer högre krav på säkerhetsdiscipliner som gör det möjligt för rätt personer att komma åt rätt resurser vid rätt tidpunkt och av rätt skäl.

En typisk arkitektur som stödjer kundnära applikationer, med anslutning till identitetshantering och behörighetskontroll samt CRM, byggs ofta på protokoll som behandlar mängder med personliga data, såsom användarnamn, kontaktuppgifter, men även andra känsliga attribut. Ofta ligger denna data okrypterad i databaser på servrar för att inte försvåra när en användare vill ställa en mer komplicerad query till databasen. Detta gör datan sårbar för avlyssning när den skickas mellan databas och API men även om någon skulle angripa servern och komma åt hela databasen. Detta arbetet kommer försöka tackla detta problem genom att skapa en fullstack-applikation där en användare via ett enklare GUI kan utföra CRUD operationer samt skicka dynamiska anrop via ett REST API som är kopplat till en krypterad databas.

1.2 Problemformulering

En viktig egenskap i en relationsdatabas är funktionen att kunna göra dynamiska queries över flera kolumner för att matcha värden på olika rader i en tabells olika kolumner. Exempelvis en databas med en tabell som hanterar personinformation så som förnamn, efternamn och ålder där personens personnummer fungerar som primary key, se figur 2.

Personnummer	Förnamn	Efternamn	Ålder
19911010-1919	Otto	Andersson	28
19900101-9090	Jonathan	Andersson	29
19921111-0101	Tove	Andersson	27
19900202-1111	Julius	Andersson	29

Figur 2. Utdrag ur relationsdatabas där personnummer agerar primary key

En dynamisk query på denna tabell skulle kunna vara en sökning på alla personer som heter Andersson i efternamn och som är 28 år eller äldre. Databasen hade då gjort en sökning över kolumnerna ”Efternamn” och ”Ålder” för att se vilka rader som uppfyller alla sökvillkor och sedan returnerat de raderna.

För att förhindra läckage av känslig information vid ett angrepp mot servern som databasen ligger på är det fördelaktigt om raderna i tabellen inte lagras i klartext. Ett vanligt sätt att lösa problemet med känslig data som man inte vill ha i klartext är att kryptera den. I fallet med databaser skulle man kunna kryptera alla värden i raderna vid insättning, där tabellerna efter kryptering hade kunnat se ut enligt Figur 2.

Personnummer	Förnamn	Efternamn	Ålder
0cc175b9c0f1b6a83 1c399e269772661	cda9194666616fe3c 93c603818c04806	f7d8c86ca871bbc77 2e8ea17a72ecc63	33e75ff09dd601bbe 69f351039152189
92eb5ffee6ae2fec3a d71c777531578f	4cb7c834a25feb87c b704f98ae092f6f	f7d8c86ca871bbc77 2e8ea17a72ecc63	6ea9ab1baa0efb9e1 9094440c317e21b
4a8a08f09d37b7379 5649038408b5f33	e356c6ffd51fbbf7ba 48709d2a5a7c0c	f7d8c86ca871bbc77 2e8ea17a72ecc63	34173cb38f07f89dd bebc2ac9128303f
8277e0910d750195 b448797616e091ad	9b7add2cee0983bf3 401612983a9da7f	f7d8c86ca871bbc77 2e8ea17a72ecc63	6ea9ab1baa0efb9e1 9094440c317e21b

Figur 2. Krypterad data av Figur 1

Den viktiga egenskapen hos en relationsdatabas att kunna ställa en dynamisk query till databasen blir nu oanvändbar. Denna problematik gör att många databaser med viktig information idag lagras oskyddat.

1.3 Syfte

Syftet med projektet är att utveckla en applikation som ökar säkerheten i en relationsdatabas ur en konfidentiell synpunkt. Applikationen ska skicka anrop till en krypterad relationsdatabas och presentera resultatet okrypterat i ett användargränssnitt. Applikationen kommer fungera som ett proof-of-concept för en lösning på problembilden. Den kommer även att kunna användas för att göra jämförelser på transaktionstiden för en krypterad gentemot okrypterad databas för olika stora datavolymer.

1.4 Mål

Detta projekt avser att:

- Upprätta en relationsdatabas med krypterad data
- Utveckla ett REST API kopplat till relationsdatabasen som sköter anrop samt kryptering till databasen.
- Utveckla ett enklare användargränssnitt för presentation av okrypterad data
- Konstruera ett verktyg som kan användas för att beräkna transaktionstider från en databas med eller utan kryptering
- Implementera en krypterad kommunikationskanal mellan klient och REST API
- Utvärdera huruvida sökningarna i den krypterade databasen utförs kostnadseffektivt ur ett prestandaperspektiv

1.5 Omfång och avgränsningar

All kryptering som sker i projektet kommer utföras med hjälp av redan befintliga standarder, alltså kommer ingen egen algoritm för kryptering att implementeras. Olika krypteringsstandarder kommer inte att prövas i applikationen utan en lämpligt utvald standard kommer att användas. Kryptografiska system kommer inte implementeras på nytt, istället kommer proprietära implementationer utvecklat av PostgreSQL att användas. Exempeldatabasen som används kommer att vara lokal och all data kommer att vara mockad.

2 Teknisk bakgrund

2.1 RESTful tjänster

REST eller REpresentational State Transfer är en typ av arkitektur som används för att upprätta kommunikation mellan system på internet. REST bygger på användning av funktioner i HTTP för att skapa en tillståndslös, kompatibel och skalbar kommunikation mellan klient och server [1].

Eftersom så många programmeringsspråk har stöd för HTTP är det enkelt att, mha REST, bygga en generell webbtjänst som inte är bunden till ett specifikt språk eller system. Det krävs heller inte proprietära implementationer för kommunikationssäkerhet mellan klient och server eftersom HTTP kan användas över välkända säkerhetsmekanismer som SSL och TLS [1].

Tack vare att REST bygger på HTTP kan man bygga klient- och serverapplikationer helt oberoende av varandra. De enda kraven är att båda sidor kan hantera HTTP och att den typ av data som ska hanteras specificeras, därefter sköter ett REST API kommunikationen mellan de två. REST kan hantera godtycklig typ av data men i de flesta fall används JSON eller XML [1].

En RESTful tjänst är ett API som är baserat på REST-teknologi och hanterar anrop från en klient med funktioner som GET, POST, PUT och DELETE för att hämta, lägga till, ändra eller ta bort data på en serverbaserad applikation [2].

Tjänsten använder endpoints, unika sökvägar för att specificera vilken typ av anrop som ska exekveras eller vilken data som ska hämtas. För att hämta data skickas ett anrop från en klient till en endpoint för valt anrop, med eller utan meddelande, varpå REST API hanterar givna parametrar, hämtar data från servern och skickar tillbaka datan till klienten [2].

Vid användning av REST API:t “Where is the ISS at?” som hanterar uppdaterad information för ISS (International Space Station) kan ett GET-anrop skickas till endpoint https://api.wheretheiss.at/v1/satellites/25544, där 2544 specificerar ett unikt ID för ISS, varpå information om ISS nuvarande koordinater och annan information skickas tillbaka till klienten i JSON-format.

Att REST är tillståndslöst innebär att server och klient inte har någon information om varandras tillstånd. Alltså skickas varje anrop från klient till server isolerat utan att servern behöver ha någon vetskap om tidigare händelser. Utan krav på lagring av klienters session-data på servern blir uppskalning enklare eftersom servern kan hantera många klienters anrop samtidigt. Dessutom behöver servern aldrig hålla reda på vilken klient som gör vad eftersom varje klient alltid skickar all nödvändig information med varje meddelande.

2.2 Relationsdatabas

En relationsdatabas är en samling av data där relationerna mellan olika data är fördefinierad på ett visst sätt. Relationen mellan data förklaras genom tabeller med kolumner och rader. Varje rad består av data relaterade till varandra och kolumnerna beskriver de olika typer av attribut för raden. En rad representerar ett objekt som förklarar hur de olika datan i varje kolumn förhåller sig till varandra och identifieras genom något som kallas en primärnyckel, en eller flera kolumner som innehåller ett unikt identifierande fält för raden.

För att kommunicera med en databas används språket SQL för att lägga till, ändra, ta bort och hämta data ur databasen, en eller flera av dessa operationer kan grupperas ihop till något som kallas en transaktion. Transaktioner för en relationsdatabas måste vara vad man kallar ACID (översatt till Atomisk, konsistent, isolerad och varaktig). Atomicitet innebär att för en transaktion ska vara giltig behöver hela transaktionen genomföras, konsistent betyder att den följer alla regler och begränsningar som satts upp för databasen, isolerad innebar att varje transaktion är självständig för att kunnat uppnå jämlöpande exekvering och varaktig innebär att när transaktionen är färdig så är förändringarna permanenta.

2.3 Kryptografi

Kryptografi är en bred term som beskriver de matematiska tekniker som används för att skydda meddelanden från att tolkas eller avlyssnas av en obehörig part [3]. De parter som har behörighet till ett meddelande delar en hemlig nyckel som används vid kryptering och dekryptering och utan nyckeln är datan oläslig för någon som uppfångar meddelandet på vägen.

2.3.1 Symmetrisk kryptering

I symmetrisk kryptering används en hemlig nyckel för både kryptering och dekryptering av meddelanden. Krypteringen fungerar genom att ett meddelande, tillsammans med en hemlig nyckel körs igenom en krypteringsalgoritm som resulterar i en oläslig chiffertext. För dekryptering körs samma algoritm i omvänd ordning för att erhålla meddelandet.

Eftersom endast en nyckel används är det av stor vikt att utväxling av nyckeln sker genom säkra kanaler. Skulle nyckeln falla i fel händer är kommunikationen komprometterad och inte längre säker.

2.3.2 Asymmetrisk kryptering

I asymmetrisk kryptering används två nycklar, en hemlig och en publik där den ena används för kryptering och den andra för dekryptering. Den hemliga nyckeln är känd endast för parten som genererat nyckelparet och den publika nyckeln är känd för alla. För att kryptera ett meddelande används den publika nyckeln tillhörande parten som meddelandet ska skickas till för att kryptera meddelandet. Mottagaren dekrypterar sedan meddelandet med sin privata nyckel.

Utbytet av ett krypterat meddelande sker helt utan att någon av parterna är medvetna om varandras privata nyckel, alltså är det, till skillnad från symmetrisk kryptering, enklare att utväxla nycklar men däremot är operationerna mer kostsamma beräkningsmässigt.

2.4 Transport Layer Security

Transport Layer Security eller TLS, efterträdaren till Secure Sockets Layer, är ett kryptografiskt kommunikationsprotokoll och används för att upprätta privat och säker kommunikation mellan applikationer [4]. TLS ligger över transportprotokollet och under applikationslagret i OSI-hierarkin.

I början av varje anslutning sker en handskakning där nycklar för kryptering av data upprättas. Därefter autentiserar en eller flera parter sig mha digitala certifikat som resterande parter sedan validerar. Till slut skapas unika nycklar för sessionen och anslutning upprättas.

Autentiseringen och utväxling av nycklar använder asymmetrisk kryptografi som RSA och krypteringen av skickad data använder symmetrisk kryptografi.

För att säkerställa den skickade datans integritet appliceras en MAC till varje meddelande som sedan kontrolleras för att se att ingenting i meddelandet har manipulerats från att det skickades tills att det togs emot.

2.5 Digitala Certifikat

Ett certifikat är ett digitalt dokument som används för att validera ägandeskap av en publik kryptografisk nyckel. Dokumentet innehåller information om nyckeln, namnet på ägaren, namnet på utfärdaren av certifikatet och giltighetstid för certifikatet [5].

Certifikat utfärdas av en CA (Certificate Authority), en betrodd tredje part som fastställer validiteten hos publika nycklar. Certifikatet signeras av samma CA som bevis på att certifikatet inte manipulerats och att nyckeln är pålitlig [5].

Det är också möjligt att skapa ett självsignerat certifikat, men eftersom det saknar signatur från en CA anses det inte vara pålitligt. Hos tjänster som använder självsignerade certifikat vid exempelvis upprättande av HTTPS möts användare av en varning att kommunikationen inte är säker.

3 Systemimplementation

Följande kapitel beskriver det praktiska genomförandet av projektet. Här detaljeras de verktyg som använts under projektets gång och en full redogörelse av hur själva systemet, i dess tre delar, implementerats.

3.1 Verktyg

3.1.1 Spring Boot

Spring är ett flexibelt ramverk som används främst för att bygga webb- och företagsapplikationer [7] i Java. Spring förser utvecklare med en mängd färdiga funktioner och möjligheter till konfiguration, vilket gör det enkelt att skräddarsy behoven för specifika applikationer utan att behöva skriva all kod från grunden.

Spring Boot är en förlängning av Spring-ramverket och utvecklades som lösning till problematiken kring konfigurering av Spring-applikationer. Ju mer funktionalitet som lades till i Spring desto mer komplex blev konfigureringen och därmed benägenheten att ge upphov till fel. Spring Boot kommer med en rad färdiga typ-konfigurationer för applikationer som t.ex. Webb eller kommunikation med databas. Med hjälp av Java-annotationer kan utvecklare konfigurera dependencies och dependency injection direkt i källkoden istället för att behöva göra specifikationer i en XML-fil bunden till projektet [7]. Stöd för snabb lansering av applikationer kommer också inbyggt, allt för att utvecklare ska kunna fokusera på själva applikationen.

3.1.2 Eclipse IDE

Eclipse är en IDE (Integrated Development Environment) som används för mjukvaruutveckling. Majoriteten av Eclipse är skriven i Java och dess primära användning är för att utveckla Java- och Webbapplikationer [8]. Eclipse har flera nyttiga funktioner som underlättar arbetet för utvecklaren, några av dessa är code completion, debugger och möjligheten att installera plug-ins för att skräddarsy utvecklarens behov.

3.1.3 PostgreSQL

PostgreSQL är ett open-source ORDBMS. PostgreSQL har sitt ursprung i POSTGRES-projektet som var ett projekt finansierat av det amerikanska försvaret och grundades på University of Berkeley i Kalifornien 1986.

År 1994 skapades Postgres95 av Andrew Yu och Jolly Chen som la till funktionen att tolka SQL till POSTGRES kodbasen. I en senare version döptes Postgres95 om till PostgreSQL som idag används inom den privata, kommersiella och akademiska sektorn.

PostgreSQL är idag en av de vanligaste relationsdatabaserna med fördelarna att den kan användas gratis samtidigt som den har moderna funktioner som komplexa queries och

ytterligare inbyggda moduler som pgcrypto som förser användaren med kryptografiska funktioner för PostgreSQL [9].

3.1.4 pgAdmin

PgAdmin är ett verktyg för att hantera databaser i PostgreSQL. Verktyget erbjuder bla. möjligheten att skapa nya databaser, radera befintliga databaser, verktyg för att skicka queries mot databaser och ett gränssnitt för att presentera data.

PgAdmin finns tillgängligt både som desktop- och webbapplikation och den senaste versionen är 4.

3.1.5 WindowBuilder

WindowBuilder är en inbyggd plug-in till Eclipse som gör det enkelt att skapa GUI-applikationer i Java. Genom att använda Swing-biblioteket och verktyg för WYSIWYG-layout (What You See Is What You Get) underlättar det för användaren att generera kod för att ta fram grafiska komponenter. Funktionen “visual designer” tillåter användaren att dra och släppa komponenter till fönstret utan att behöva uppskatta koordinater, initiera variabler eller instansiera objekt.

Användaren kan skapa enkla användargränssnitt utan att skriva någon kod som istället genereras av WindowBuilder. Det är sedan möjligt att redigera fritt i den genererade koden och det krävs inga andra bibliotek för att kompilera och köra den.

3.1.6 Hibernate

Hibernate är ett ORM-ramverk som används för att behålla persistence i applikationer. Persistence innebär att man vill spara ett objekts tillstånd även efter man avslutat applikationen. Hibernate erbjuder stöd för det genom att omvandla objektinstanserna i en java-applikation till en rad i en relationsdatabas och föra in den i relationsdatabasen. Hibernate tar även hand om arbetet med att konvertera mellan de olika datatyperna i Java till datatyperna i relationsdatabasen [10].

3.1.7 Maven

Maven är ett open source build tool främst för Java-utveckling. Maven tillhandahåller smidig och kraftfull hantering av dependencies [11] vilket innebär att eventuella behov av externa projekt eller open source ramverk laddas ned av Maven automatiskt genom enkla deklarationer i en XML-fil med namnet POM (Project Object Model) istället för att behöva ladda ner var och en separat.

Mavenprojekt kommer utformade med en standardiserad katalogstruktur [11]. Alltså finns klara rekommendationer om vart diverse filer av olika typ ska placeras, vilket ger utvecklare

en tydlig bild om hur ett projekt skall organiseras och även en tydligare generell överblick av projektet.

3.1.8 Java 8

Java är ett klassbaserat och objektorienterat programmeringsspråk utformat för att applikationerna som skapas med det ska ha så få dependencies som möjligt. Efter att java kod är kompilerad kan den köras på alla plattformar som har en JVM (Java Virtual Machine) installerad. Javas syntax liknar C och 2019 var Java ett av de mest använda programmeringsspråken [12].

3.1.9 Pgcrypto

Pgcrypto är en modul som tillhandahåller ett flertal funktioner för att kryptera data i en databas i PostgreSQL [13].

3.1.10 Java Keytool

Java Keytool är ett verktyg för skapande och hantering av kryptografiska nycklar och certifikat och kommer inkluderat med Oracle JDK [6]. Användare kan skapa nycklar, både symmetriska och asymmetriska, och tillhörande certifikat för att autentisera sig mot kommunicerande parter.

Nycklarna och certifikaten lagras i en keystore som är en specifik databas för kryptografiska nycklar och certifikat.

3.2 Krypteringsalgoritmer

I detta kapitel beskrivs de olika krypteringsalgoritmer som använts.

3.2.1 Advanced Encryption Standard

Advanced Encryption Standard eller AES är en krypteringsstandard och är ett av de mest använda symmetriska krypteringssystemen som finns tillgängliga för att kryptera digital data [14]. AES togs fram som ersättare år 2000 för den föråldrade standarden Data Encryption Standard eller DES av Vincent Rijmen och John Daemen.

AES krypterar data genom en algoritm som delar upp binärdata i block av bestämd storlek som krypteras med en hemlig nyckel, sedan substitueras och permuteras datan i varje enskilt block ett specifikt antal ronder för att göra den oläslig för en obehörig part [14]. Datan dekrypteras genom att köra chiffertexten genom samma algoritm och nyckel som den krypterades med i omvänd ordning.

AES har stöd för nycklar av storlek 128, 192 och 256 bitar och kör substitution och permutation 10, 12 och 14 ronder beroende på nyckelns storlek.

3.2.2 RSA

Rivest–Shamir–Adleman eller RSA är ett asymmetriskt krypteringssystem och är, sen dess kreation år 1977, det mest använda och implementerade systemet i världen [15]. Likt andra asymmetriska kryptografiska system använder RSA två nycklar, en publik för kryptering och en privat för dekryptering.

Generering av nycklar med RSA sker enligt följande:

1. Två stora primtal p och q av samma storlek väljs slumpmässigt.
2. p och q multipliceras för att få produkten N.
3. e väljs slumpmässigt så att $1<e<(p-1)(q-1)$ där e och $(p-1)(q-1)$ är relativt prima.
4. d beräknas sådan att $de \equiv 1 \bmod (p-1)(q-1)$

Den publika nyckeln består sedan av p och N och den hemliga nyckeln av d och N.

För kryptera ett meddelande omvandlas först texten till ett tal $M < N$. Sedan beräknas chiffertexten enligt:

$$C = M_e \bmod N.$$

För att dekryptera ett meddelande beräknas sedan:

$$M = C_d \bmod N = M = M_{ed} \bmod N.$$

RSA anses vara säkert eftersom systemet förlitar sig på svårigheten att primtalsfaktorisera stora tal. Eftersom endast e och N är kända, krävs det att en illvillig part kan primtalsfaktorisera N för att beräkna e och i sin tur d. Eftersom N är ett väldigt stort tal kan inte beräkningen med dagens datorer utföras inom rimlig tid.

3.2.3 pgp_sym_encrypt/decrypt

Pgp_sym_encrypt är en funktion för symmetrisk kryptering av meddelanden och är proprietär för pgcrypto-modulen i PostgreSQL [13]. Funktionen tar två strängar som inparametrar, ett meddelande och ett lösenord och returnerar ett krypterat binärt meddelande. Det krypterade meddelandet består av två paket, ett paket för en nyckel och ett paket för krypterad data.

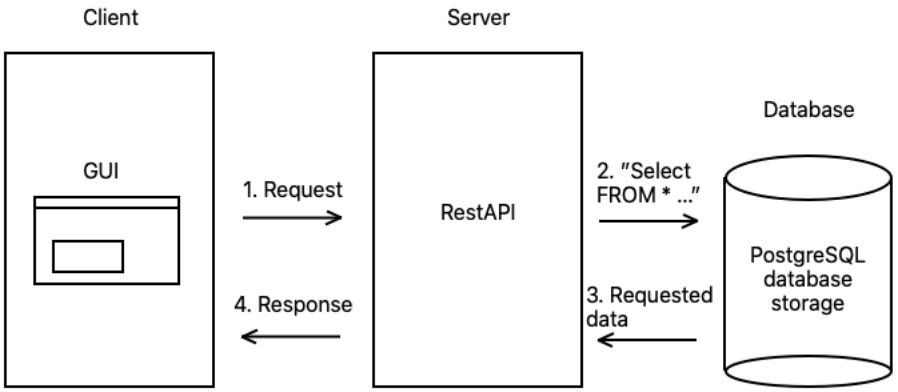
Vid exekvering körs först det angivna lösenordet genom en hash-algoritm som omvandlar strängen till en binär nyckel som används för krypteringen. Därefter sker, om specificerat, omvandling och komprimering innan en slumpmässig IV adderas till början av meddelandet. Sedan bifogas en hash av samma slumpmässiga IV i slutet av meddelandet. Slutligen krypteras meddelandet med den binära nyckeln enligt AES-128 algoritmen, om ingen annan algoritm specificerats, och placeras i data-paketet [13].

För att dekryptera meddelandet används funktionen pgp_sym_decrypt som tar två inparametrar, en sträng för lösenord och ett krypterat meddelande i binär form och returnerar ett okrypterat meddelande [13].

3.3 Systemuppbyggnad

Applikationen är uppdelad i tre lager. Ett klientlager med ett grafiskt användargränssnitt som används för att utföra dynamiska sökningar i databasen, göra insättningar och borttagningar av användare i databasen samt presentera sökresultat. Klientlagret kommunicerar med serverlagret genom REST-anrop som ibland innehåller JSON-data.

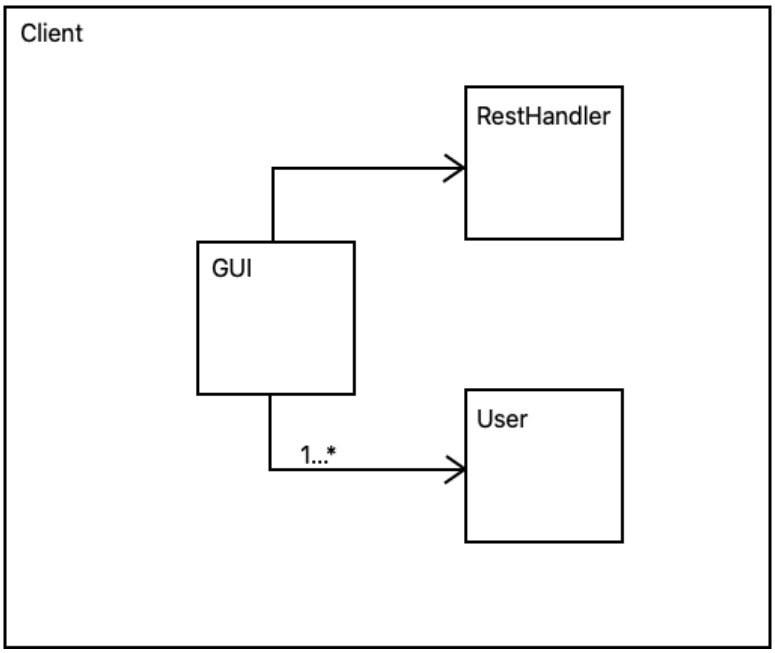
Serverns uppgift är att tolka användarens anrop och förvandla den eventuella datan till SQL för att senare skicka den vidare till databasen. I figuren nedan visas en övergriplig bild över hur dataflödet ser ut när användaren gör en sökning mot databasen.



Figur 3. Applikationens dataflöde

3.3.1 Klientlager

Klienten består till huvuddel av tre olika klasser; User, RestHandler och GUI.
User-klassen är en POJO klass som modellerar en användare från databasen till en objektorienterad instans. RestHandler-klassen skickar den tolkade inputen från användaren till REST API:t och hanterar konvertering från JSON-objekt till User-instanser. GUI-klassen målar upp de grafiska komponenterna och presenterar resultat och tar emot input från användaren. Klassen har en instans av RestHandler, som den anropar när förändringar skett i användargränssnittet, och en ArrayList av User-objekt, relationen mellan klasserna kan man se i Figur 4.



Figur 4. UML-diagram för Klientlagret

3.3.1.1 User

User är en POJO-klass som också finns i servern. Klassen har fyra klassvariabler varav tre strängar för en användarens id, förnamn och efternamn och en heltalsvariabel för användarens ålder. Klassen har tillhörande get- och set-metoder för variablerna.
Id-variabeln är unik och genereras av servern när en användare skapas och motsvarar en primary key i databasen. Klassen har tre konstruktorer; en tom, en som tar in parametrar för alla klassvariabler och en som tar parametrar för alla klassvariabler utom för id.

3.3.1.2 RestHandler

Klassen RestHandler består av fem olika metoder som används för att skicka olika anrop till servern. Den innehåller en instans av typen ObjectMapper som används för att göra om JSON-data till en ArrayList av User, en URL-variabel vars värde kommer vara beroende på vilken typ av anrop som kommer att göras och en OutputStream som är en ström som kommer öppnas med den URL som är angiven.

Metoden postRequest används för att lägga till en ny eller ändra en befintlig användare i databasen. Den tar emot ett User-objekt som inparameter där User-objektet är den nya användaren eller den befintliga användare som läggas till eller ändras. Om det är en ny användare är id fältet i User-objektet tomt och annars är det användaren med givet id som ska ändras.

En ström öppnas på angivet URL (i detta fall http://localhost:8081/users/), sedan används ObjectMappers metod writeValue som gör om User-objektet till JSON-format och skickar ett POST-anrop på strömmen, postRequest väntar sedan på att få en svarskod från REST API:t som berättar huruvida operation var lyckad eller inte.

Metoden deleteRequest tar emot en sträng som inparameter, skickar sedan ett DELETE-anrop till servern på URL “http://localhost:8081/users/<strängens värde>” där den begär om att ta bort användaren med id angivet i strängen. Metoden väntar sedan på en svarskod från REST API:t som svarar hur operation gick.

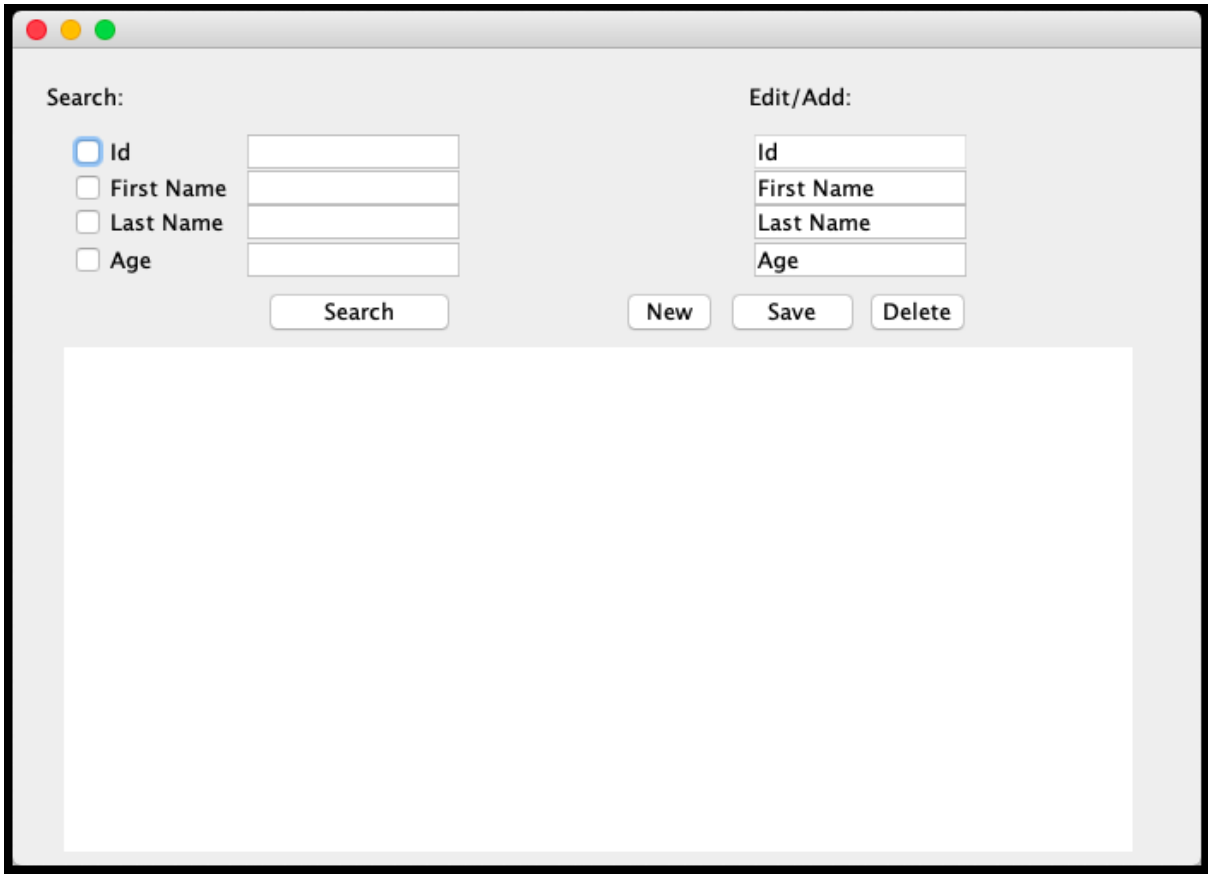
Metoden readJSON används för att hämta alla användare i databasen, den skickar ett GET-anrop till URL “http://localhost:8081/users”. Den tar sedan emot ett JSON-objekt med användarna och gör om de till en ArrayList av User-objekt med hjälp av ObjectMappers readValue-metod. Metoden returnerar sedan denna ArrayList av User-objekt.

Den andra metoden heter readJSON och tar en sträng sominparameter och används för att göra olika GET-anrop till servern. Vilken typ av anrop som kommer göras bestäms av värdet på strängen, exempelvis “http://localhost:8081/firstName/Jonatan” där man vill ha alla användare med förnamnet “Jonatan”. Metoden tar sedan emot ett JSON-objekt med alla användare som heter Jonatan och gör mha ObjectMapper readValue om dessa till en ArrayList av User-objekt som returneras av metoden.

Den sista metoden är readIDJSON som tar en sträng som inparameter som specificerar en användares id. Den skickar sedan en GET-anrop till servern på URL “http://localhost:8081/users/<strängens värde>” där den begär användaren med givet id. Metoden tar sedan emot ett JSON-objekt med användaren som den omvandlar till en ArrayList av User-objekt med bara ett User-objekt som sedan returneras.

3.3.1.3 GUI

GUI-klassen har tagits fram med hjälp av WindowBuilder för att generera ett lämpligt gränssnitt för CRUD. Gränssnittet är uppdelat i tre delar; en del för att välja vilka parametrar man vill söka på i databasen och vilken data du vill ange somsökparameter, en som presenterar resultatet i en tabell och en del för att lägga till, ta bort eller ändra på användare i databasen. När GUI-klassen körs målas fyra stycken kryssrutor, fyra stycken knappar, en tom tabell och åtta stycken textfält upp enligt Figur 5. Alla textfält utom det översta textfältet under “Edit/Add:” är editerbara.

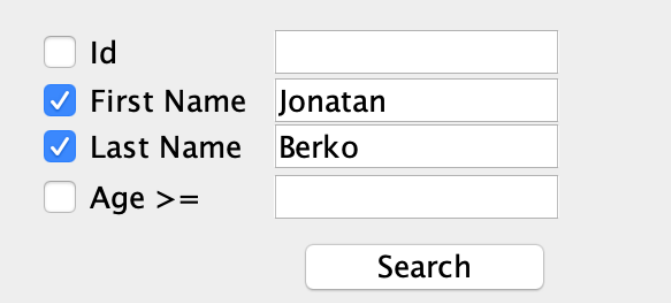


Figur 5. Överblick över GUI

De fyra stycken kryssrutorna kopplas till varsin ItemListener som lyssnar om kryssrutan blir avklickad och om så är fallet ska textfältet bredvid kryssrutan tömmas. De fyra knapparna kopplas till varsin ActionListener som på olika sätt kommer att kalla på RestHandler-objektet för att skicka önskat anrop till server.

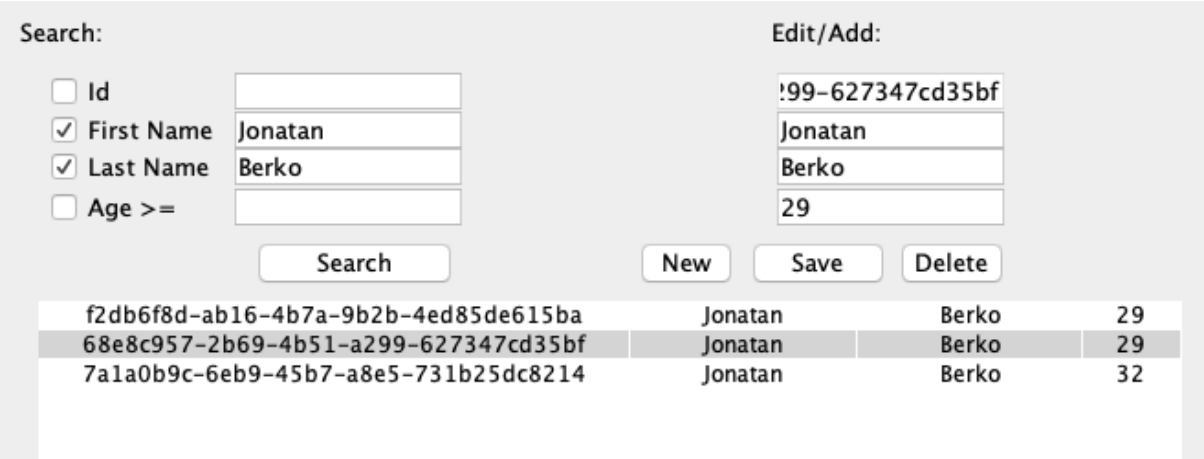
Knappen “Search” används när man vill göra en sökning i databasen, när en ActionListener märker av att knappen har tryckts på kallar den på metoden readFilter.

Metoden readFilter läser först av input i textfälten bredvid kryssrutorna som sparas i variabler, metoden går sedan igenom en switch-case där den kollar vilken av kryssrutorna som är klickade. Sedan kallar den på RestHandler-instansen och använder sig av dess readJSON-metod med en sträng sominparameter. Strängen är utformad efter vilka kryssrutor som är iklickade och vad värdet i textfälten jämte dess kryssruta är. Ett tryck på “Search” enligt Figur 6 hade genererat strängen “http://localhost:8081/fullName/?firstName="+”Jonatan”+"&lastName="+”Berko”” som RestHandler sedan använder för att skicka ett anrop till servern. RestHandler kommer sedan att returnera en ArrayList av User-objekttill GUI-klassen som kommer att ges sominparameter till metoden createTable.



Figur 6. - Sökning på för- och efternamn

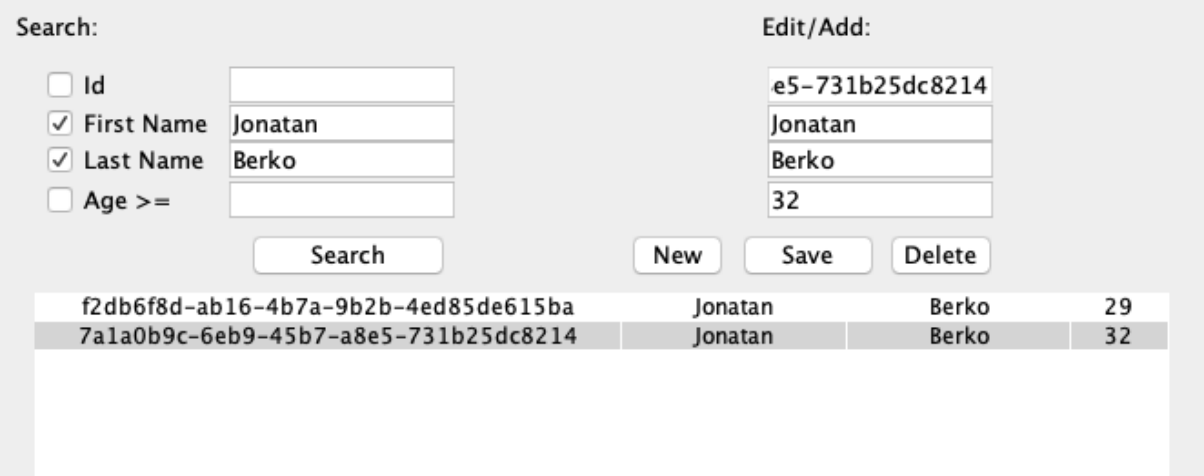
Metoden createTable tar emot en ArrayList av User-objekt och skapar kolumner i tabellen för id, förnamn, efternamn och ålder. Den skapar sedan rader i tabellen för varje User-objekt i specifik ArrayList och applicerar en ListSelectionListener till tabellen som gör att den känner av när en av raderna i tabellen markeras. När ett klick sker på en av raderna laddas värdena i tillhörande textfält under “Edit/Add:” där de tre understa fortfarande är editerbara, se figur 7.1.



Figur 7.1. - Data laddas till “Edit/Add” med klick på en rad.

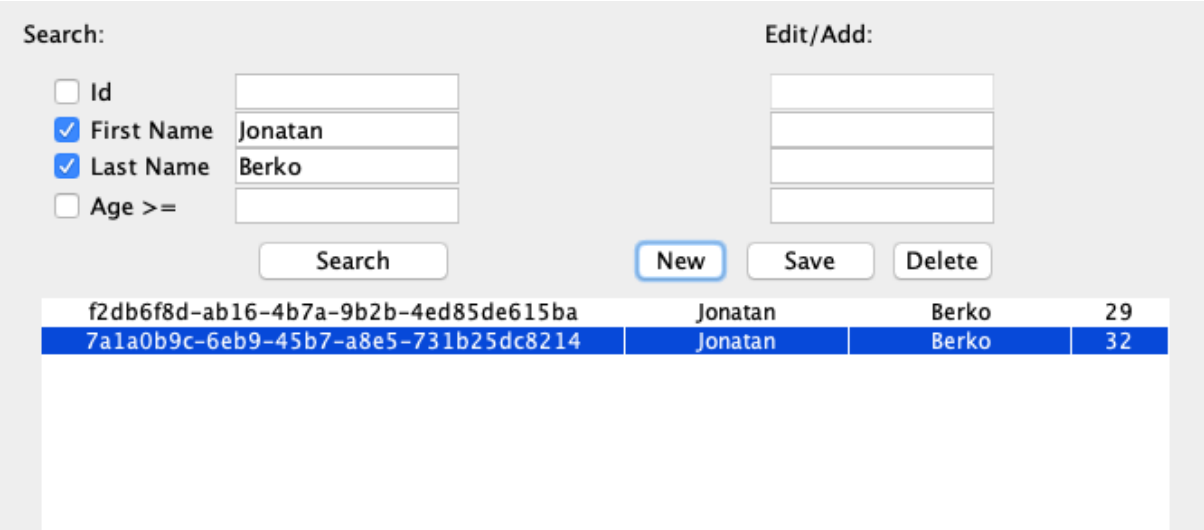
Om man sedan ändrar värdet i ett eller flera av fälten och vill spara ändringar till databasen trycker man på knappen “Save”. Den ActionListener som är kopplad till knappen “Save” läser värdena från textfälten och skapar ett nytt User-objekt. Den kallar sedan på restHandler och metoden postRequest med User-objektet som inparameter för att skicka ett POST-anrop till servern. Efter att anropet skickats till servern körs metoderna readFilter och createTable för att ladda om tabellen.

Om en användare är laddad till fälten och man trycker på “Delete”-knappen anropar den ActionListener, som är kopplad till knappen, på deleteRequest-metoden i RestHandler med id-värdet i det översta fältet som inparameter för att skicka ett DELETE-anrop till servern som tar bort användaren med givet id från databasen. Sedan körs metoderna readFilter och createTable igen för att ladda om aktuella data till tabellen.



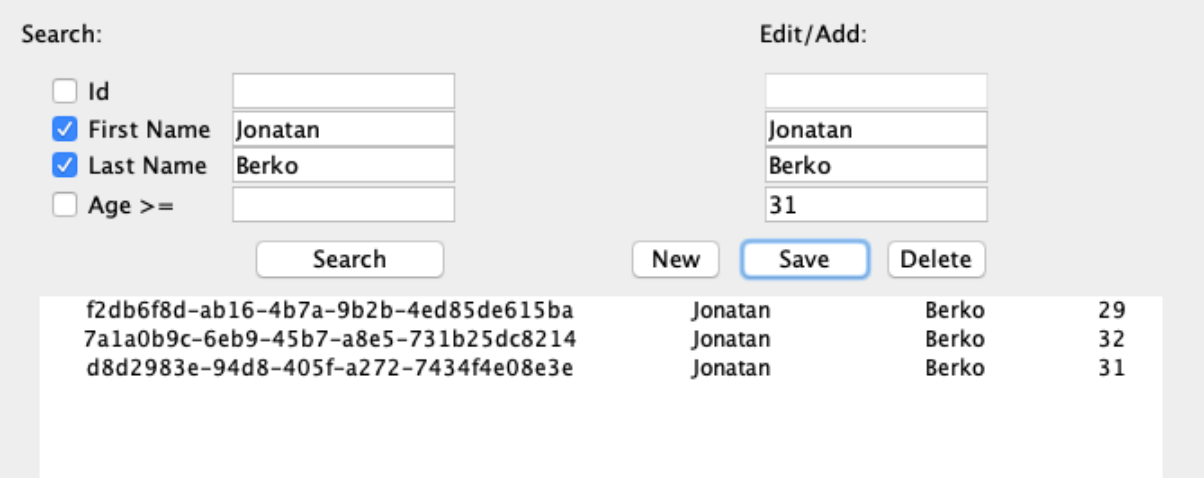
Figur 7.2. Användare har tagits bort.

ActionListener kopplad till knappen “New” tar bort värdena i fälten under “Edit/Add:”.



Figur 7.3. Rensning av fält vid klick på “New”.

Användaren kan nu fylla i värden i de tre understa fälten men inte id-fältet. När användaren sedan trycker på Save-knappen kommer dess ActionListener skapa ett User-objekt med tre parametrar till konstruktorn vilket gör att det skapas ett User-objekt utan något värde i id-variabeln. RestHandlers postRequest-metod anropas sedan med User-objektet som inparameter och skickar ett POST-anrop till servern där ett id kommer genereras innan objektet läggs in i databasen. Sedan körs metoderna readFilter och createTable igen för att ladda om aktuell data till tabellen.

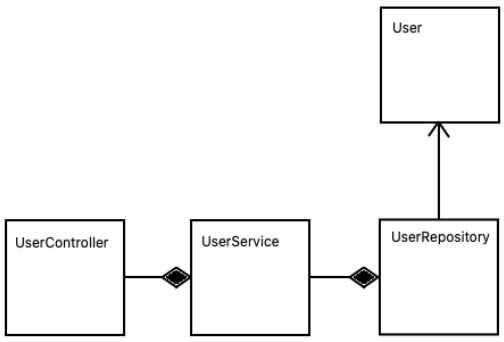


Figur 7.4. Ny användare har lagts till efter klick på “Save”

3.3.2 Serverlager

Serverlagret är ett REST API konstruerat enligt Spring MVC modellen och byggt med hjälp av Spring Boot. Serverlagret består främst av fyra klasser. UserController som är gränssnittet som klientlagret kommunicerar med, tar emot HTTP-anropen och sedan returnerar JSON-data till klienten som mottagits från UserService-klassen.

Klassen UserService sköter serverns logik och fungerar som mellanhand mellan UserController och UserRepository som hämtar data från databasen. Klassen User används som modell för ORM för att veta hur den ska modellera användare från rader i relationsdatabasen till POJO-objekt.



Figur 8. Klassrelationer i serverlager

3.3.2.1 UserController

UserController-klassen tolkar HTTPS-anrop som kommer från klienten, den dirigerar sedan UserService-klassen till vilken operation som ska utföras på databasen. Om HTTPS-anropet är av typen GET väntar den på att UserService ska returnera en lista av User-objekt som den sedan omvandlar till JSON-format för att skicka tillbaka till klienten. Om anropet är av annan typ skickar den passande svarskod beroende på hur operationen gick.

Eftersom klassen använder sig av Spring Boots @RestController-annotation är mycket av logiken inbyggd, t.ex. omvandlingen mellan POJO till JSON. För att lägga till ett nytt anrop att ta emot i klassen används @Mapping-annotationer.

```
//http://localhost:8081/fullName/?firstName={firstName}&lastName={lastName}
@GetMapping("/fullName/")
public List<User> findUsersByFullName(@RequestParam String firstName, @RequestParam String lastName){
    return userService.findUsersByFullName(firstName, lastName);
}
```

Figur 9. Metod för hämtning av alla användare i databasen

Den kommenterade raden i Figur 9 beskriver till vilken endpoint som anropet skickas till från klienten, kommentaren fyller ingen funktion mer än att underlätta för en eventuell läsare och användare av applikationen. @GetMapping-annotationen specificerar att metoden är ett GET-anrop som kommer tas emot på domänen /fullName/. Klassens @RestController-annotation initierar ett anrop på den beskrivna metoden, den tar emot parametrar för för- och efternamn och ber sedan UserService använda dess metoder för att hitta en List av User-objekt som uppfyller sökparametrarna. Den inbyggda logiken sköter sedan omvandlingen till JSON-format och skickas tillbaka till klienten.

I UserService-klassen finns nio stycken @GetMapping-annotationer för att ta hand om alla de olika fall som sökfiltret i GUI kräver. Dessa fall är följande:

- Hämta alla användare
- Hämta en specifik användare med angivet id
- Hämta alla användare med ett specifikt förnamn
- Hämta alla användare med ett specifikt efternamn
- Hämta alla användare med en specifik kombination av förnamn och efternamn
- Hämta alla användare med en specifik kombination av förnamn och ålder
- Hämta alla användare med en specifik kombination av ålder och efternamn
- Hämta alla användare med en specifik kombination av förnamn, efternamn och ålder

Det finns en @DeleteMapping-annotation som tar id som inparameter för att ta bort en användare med angivet id. Sedan finns det en @PutMapping-annotation som tar emot ett id och ett User-objekt som inparameter och används för att uppdatera användaren med angivet id och ersätter den med det uppdaterade User-objektet. Sist finns @PostMapping-annotationen som tar emot ett User-objekt som inparameter och lägger till användaren i databasen.

3.3.2.2 UserService

I klassen UserService tillämpas serverns logik, den har ett UserRepository som den använder för att dirigera CRUD-direktiven den fått från UserController. Den har metoder som anropas av UserController och det är i vissa av dessa metoder som man definierar vilka undantag som kan ges om man till exempel försöker uppdatera en användare som inte finns. Klassen genererar lämpliga svarskoder till UserController att skicka tillbaka till klienten för sina annoterade metoder. För @GetMapping anropas bara UserRepository för att hämta användare.

3.3.2.3 UserRepository

UserRepository har Spring Boots inbyggda @Repository-annotation och kommunicerar direkt med databasen. UserRepository ärvs från JPA Repository och är en ORM som hjälper till att göra om POJO User-objekt till rader i relationsdatabasen, den sköter även typomvandlingar för olika datatyper. Från JPA Repository kommer ett flertal inbyggda metoder såsom save, findAll, delete och findById som används för att spara, hämta alla användare, ta bort respektive hitta användare med ett specifikt id.

Ytterligare finns det sju stycken @Query-annotationer för att skicka queries som kan begäras av klienten i sökfiltret men som inte täcks av JPA Repository, se Figur 10.

```
@Query("SELECT u From User u WHERE u.lastName = ?1")
public List<User> findUsersByLastName(String lastName);
```

Figur 10. Specificerad query som inte täcks av JPA.

@Query-annotationen beskriver hur en query som ska sändas till databasen ska se ut när metoden som beskrivs under anropas. När metoden anropas görs lämpliga typomvandlingar och skickar en query till databasen som sedan returnerar ett svar. @Repository-annotationens inbyggda logik omvandlar sedan databasens SQL till POJO User-objekt som returneras i en List av User.

3.3.2.4 User

User-klassen används som modell av UserRepository för att omvandla en tabellrad i databasen till ett POJO, och av UserController för att omvandla ett POJO till JSON-format som sedan skickas till klienten. Ytterligare använder sig klassen av Hibernate och dess annotation @ColumnTransformer som tillåter anpassningsbar läsning och skrivning av ett POJO eller en tabellrad i databasen.

```
@ColumnTransformer(forColumn = "first_name",
        read = "pgp_sym_decrypt(first_name::bytea, 'AC10G31VX9330XP0')",
        write = "pgp_sym_encrypt(?, 'AC10G31VX9330XP0')")
@Column(name = "first_name", nullable = false)
private String firstName;
```

Figur 11. Användning av @ColumnTransformer.

I Figur 11 används @ColumnTransformer för att anpassa SQL-uttrycket som skriver java-strängen firstName i User till kolumnen first_name i databasen. Istället för vanlig insättning har uttrycket anpassats så att värdet på strängen krypteras. Vid läsning från kolumnen first_name i databasen till java-strängen firstName så anpassas uttrycket istället för att dekryptera värdena i databasen.

Klassen har fyra variabler. De tre variablerna som representerar förnamn, efternamn och ålder använder sig av @ColumnTranfsformer. Utöver dom finns en id-variabel som fungerar

som primary key i databasen. Den genereras av Hibernates @GenericGenerator för att se till att den är unik.

3.3.2.5 Krypterad kommunikation

Eftersom sökning i databasen sker via ett REST API krävs att kommunikationskanalen är säker. Utan konfigureringar sker kommunikation mot ett REST API via HTTP som inte anses säkert eftersom all data som skickas från, eller tas emot på klientsidan är okrypterad[16]. Därför krävs en implementation av HTTPS.

För att upprätta HTTPS implementeras TLS över transportlagret så att servern kan köra HTTPS över TLS i applikationslagret.

För att konfigurera TLS krävs ett certifikat på servern. För att uträtta RSA-nycklar och tillhörande självsignerat certifikat av typ X.509 används Java Keytool. Certifikatet skapas med ett kommando i terminalen, se Figur 12, och erhåller följande egenskaper:

1. Certifikatet identifierar en nyckel skapad enligt RSA-algoritmen
2. Certifikatets alias är localhost
3. Certifikatet och nyckeln lagras i en keystore-fil som heter localhost.p12 med lösenordet password
4. Keystore-formatet är PKCS #12
5. Nyckelstorleken är 4096 bitar
6. Certifikatet är giltigt i ett år

```
keytool -genkey -keyalg RSA -alias localhost -keystore localhost.p12
-storepass password -validity 365 -keysize 4096 -storetype pkcs12
```

Figur 12. Kommando för generering av självsignerat certifikat

Utöver skapandet av certifikat konfigureras servern för att upprätta TLS. I projekthierarkin under sökvägen src/main/resources återfinns filen properties.xml där eventuella serverspecifika konfigureringar kan appliceras. För att upprätta TLS görs konfigureringar enligt Figur 13.

```
13  server.ssl.key-store=classpath:localhost.p12
14  server.ssl.key-store-type=pkcs12
15  server.ssl.key-store-password=password
16  server.ssl.key-password=password
17  server.ssl.key-alias=localhost
18  server.port=8443
19  server.ssl.enabled=true
```

Figur 13. TLS-konfiguration

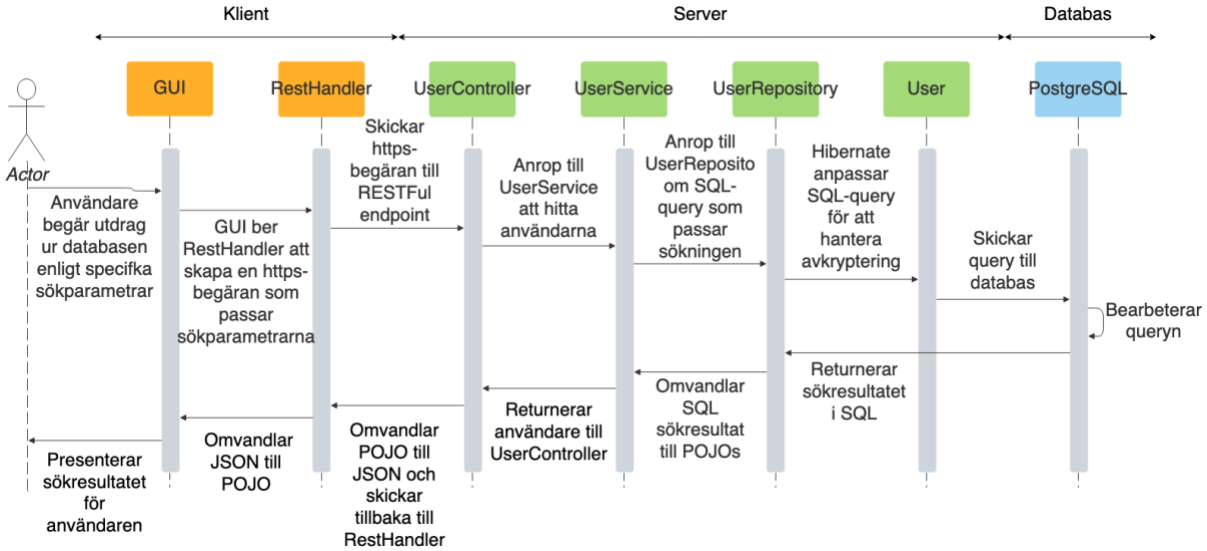
Konfigureringarna i properties.xml specificerar att servern skall använda det skapade certifikatet för att upprätta TLS, köras över port 8443 enligt konvention för kommunikation över HTTPS och till slut att TLS ska aktiveras.

3.3.3 Databaslager

Databaslagret består av en PostgreSQL-databas med en tabell, tabellen har fyra kolumner; user_id som är primary key, first_name som lagrar förnamn, last_name som lagrar efternamn och age som lagrar ålder. Alla variabler i databasen, förutom id, sparas i typen bytea, detta för att kunna krypteras med pgcrypto. Varje rad i tabellen representerar en unik användare.

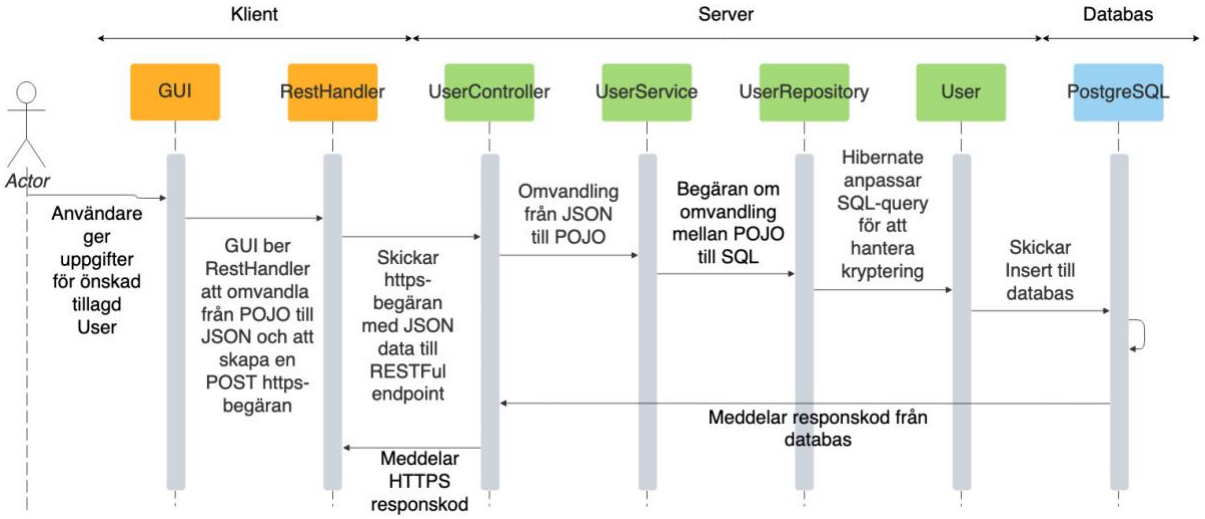
3.3.4 Dataflöde

Dataflödesdiagram för ett GET-anrop:



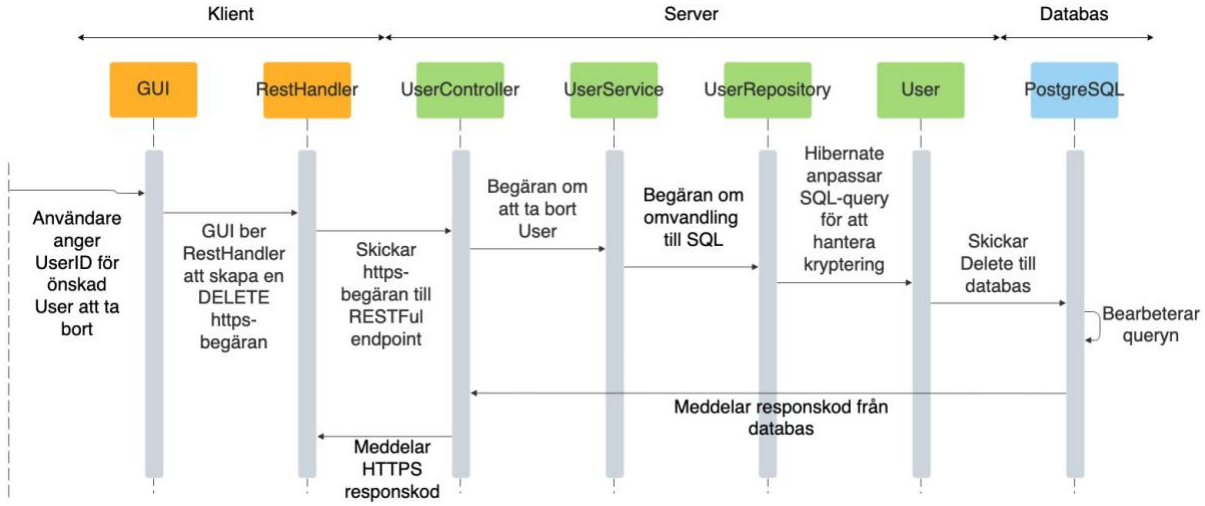
Figur 14.1. GET-anrop

Dataflödesdiagram för ett POST/UPDATE-anrop:



Figur 14.2. POST/UPDATE-anrop

Dataflödesdiagram för ett DELETE-anrop:



Figur 14.2. DELETE-anrop

3.4 Testning

Testning utförs genom att generera en SQL-fil med ett bestämt antal insättningar i Users-tabellen. Ett exempel på en insättning är “INSERT INTO Users VALUES ('"00001"','John', 'Doe', '22')”. Först laddas filen in i databasen med hjälp av pgAdmins Query Tool, sedan görs en sökning i GUI, en tidsstämpel registreras när sökningen skickas från RestHandler och en andra registreras när resultatet mottagits is i RestHandler, starttiden subtraheras sedan från stopptiden för att få den totala transaktionstiden. När sökningen görs på okrypterad data behöver User-klassen på servern modifieras så att den inte använder sig av Hibernates

@ColumnTransformer eftersom data nu inte behöver dekrypteras. Sökningar görs först över kolumnen first_name, sedan first_name och last_name och sist first_name, last_name och age för att undersöka om tiden ökar om man har fler sökparametrar och transaktionstiden för de olika sökningarna noteras.

Sedan genereras en SQL-fil med samma antal insättningar fast på formen “INSERT INTO Users (age, first_name, last_name, id) VALUES (pgp_sym_encrypt('Johan', 'AC10G31VX9330XP0'), pgp_sym_encrypt('Blom', 'AC10G31VX9330XP0'), pgp_sym_encrypt('20', 'AC10G31VX9330XP0'), "00001")”. Databasen töms på de okrypterade användarna och de krypterade insättningarna laddas sedan in i databasen med hjälp av pgAdmins Query Tool. Sedan görs samma sökningar över kolumnerna som tidigare och transaktionstiden noteras och jämförs med sökningarna på den okrypterade datan för att se om de skiljer sig.

Sökningarna görs sedan om för krypterade och okrypterade filer med olika stora antal insättningar för att se om tiderna ökar på olika sätt när antalet rader i tabellen ökar.

4 Resultat

Under projektets gång har en full stack-applikation utvecklats. Applikationen är uppdelad i tre lager; en klient, en server och en databas som tillsammans uppfyller alla de mål som sattes upp i början av arbetet.

Klienten har ett lättförståeligt gränssnitt som kan skicka CRUD förfrågningar mot databasen och sedan presentera ett okrypterat resultat av förfrågan. Klienten sköter även tidmätningen för förfrågan vilket gör att man kan koppla på en okrypterad databas och sedan göra mätningar för jämförelse för med eller utan krypterad data. Klienten och servern kommunicerar genom en krypterad kommunikationskanal uppsatt med HTTPS över TLS-protokollet.

Servern är ett REST API som sköter mottagande, vidarebefordran och returnerande av HTTPS-förfrågningar från klienten mot databasen. Den hanterar även konvertering mellan POJO och SQL och typomvandling för variablerna mellan dessa. Valet av PostgreSQL som relationsdatabas tillsammans med @ColumnTransformer-annotationenen i ramverket Hibernate gör att raderna i databasen kan vara krypterade men att det fortfarande går att göra dynamiska queries mot databasen.

Även om applikationen inte hade kunnat lanseras som en färdig produkt så fungerar den som en proof-of-concept för hur man kan lösa problematiken med att kryptera databaser. Lösningen presenterar tidsskillnaderna mellan sökningar mot okrypterade och krypterade databaser av samma storlek.

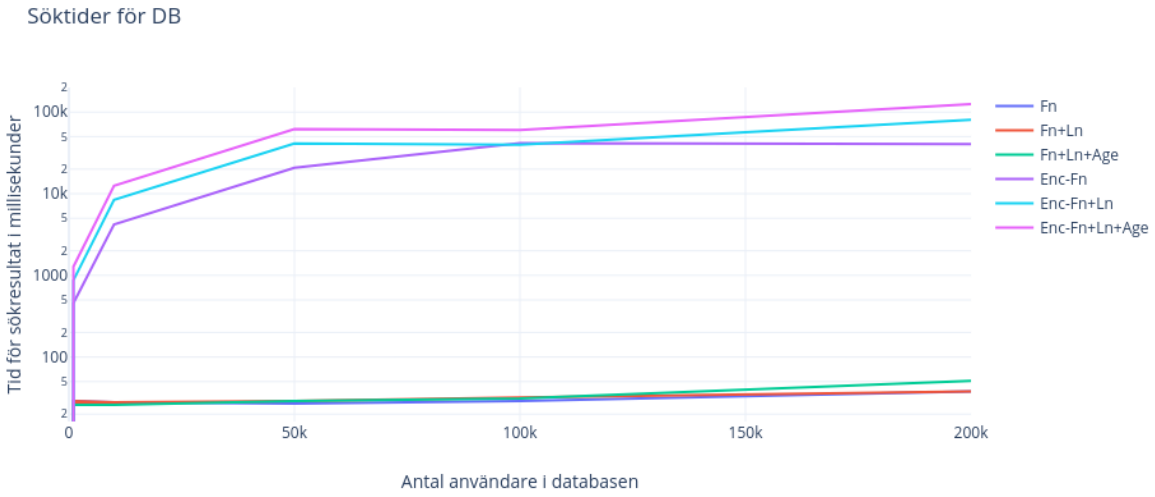
4.1 Volymtester

I Figur 15.1 presenteras resultaten för volymtesterna. Den översta raden anger hur många rader det finns i databasen när sökningen görs. Den första kolumnen representerar vilken typ av sökning som utförs. Fn står för sökning okrypterad databas på förnamn, Fn+Ln för förnamn och efternamn och Fn+Ln+Age för förnamn, efternamn och ålder. Enc-Fn, Enc-Fn+Ln, Enc-Fn+Ln+Age står för sökning i krypterad databas i samma respektive ordning. Varje sökning gjordes tre gånger för att märka anomalier och tiderna i för varje sökning, i millisekunder, kan ses i varje cell. Ett medelvärde för varje sökning togs sedan fram för att ta fram diagrammet i Figur 15.1.

	1000	10000	50000	100000	200000
Fn	37+36+31	29+29+26	29+26+27	30+28+29	40+37+38
Fn+Ln	27+32+27	28+27+28	31+29+28	33+31+32	39+36+39
Fn+Ln+Age	25+26+26	25+26+26	28+29+30	30+31+33	51+50+51
Enc-Fn	450+459+458	4182+4224+4195	20829+20699+20731	41765+41397+41465	37873+39929+43769
Enc-Fn+Ln	879+870+867	8374+8383+8515	41542+40992+41541	40030+39120+40588	79712+80919+80123
Enc-Fn+Ln+Age	1284+1278+1284	12523+12474+12435	62549+61253+62256	56458+66137+57701	123487+128677+122707

Figur 15.1 Tabell över testresultat

Första kolumnen i Figur 15.1 beskriver de sökningar som utförts enligt följande:
Fn = Förnamn, icke krypterad databas
Fn+Ln = Förnamn och efternamn, krypterad databas
Fn+Ln+Age = Förnamn, efternamn och ålder, krypterad databas
Enc-Fn = Förnamn, krypterad databas
Enc-Fn+Ln = Förnamn och efternamn, krypterad databas
Fn+Ln+Age = Förnamn, efternamn och ålder, krypterad databas
Första raden beskriver antalet rader i databasen. Resultaten i cellerna representerar tre olika sökningar och presenteras i millisekunder.



Figur 15.2 Graf över testresultat.

Transaktionstider för sökningar på en okrypterad databas är nästintill oföränderliga för databasstorlekar mellan 1000 och 200 000 användare, på en, två och tre kolumner. En liten ökning kan synas på sökningarna för 200 000 användare över tre kolumner.

För sökningarna på krypterad databas skiljer transaktionstiderna avsevärt. Sökning över en kolumn på en databas med 1000 användare tar cirka 450 millisekunder. I extremfallet när sökningen görs över tre kolumner på en databas med 200 000 användare tar sökningen cirka 120 sekunder. Söktiden dubblas för varje kolumn som tas med i sökningen för alla databasstorlekar förutom 100 000 användare där den beter sig lite oförväntat. Söktiden verkar dessutom öka linjärt med antal användare i databasen förutom även där på databasstorlek 100 000.

5 Diskussion

5.1 Hinder för problembild

Under den undersökande fasen av arbetet möttes vi av många hinder när vi försökte analysera olika potentiella lösningar för problembilden.

Vår initiala idé till att lösa problemet var att kryptera vid insättning och vid sökning kryptera sökparametrar och jämföra dom direkt i databasen. Så om man till exempel vill hämta alla som har förnamn “Jonatan” och efternamn “Berko” kan man kryptera det med samma nyckel och algoritm i ett lager utanför databasen för att kunna jämföra det med raderna i databasen. Det hade krävt att vi använde oss av deterministisk kryptering vilket inte hade varit optimalt eftersom varje data med samma innehåll hade krypterats till samma chiffertext, exempelvis hade alla chiffertexter som erhållits från kryptering av ”Jonatan” sett likadana ut. Trots att det då finns stora sårbarheter för uttömmande attacker är det fortfarande ett snäpp säkrare än spara datan i klartext. Problem uppstår dock när man vill göra sökningar där informationen i värdets innehåll är av betydelse, till exempel om man vill ta alla som är äldre än en viss ålder. Det krypterade värdet kan då inte ge någon information kring vilken ålder de olika raderna har eftersom det presenteras som krypterad text och man kan därför inte veta om raden ska tas med i resultatet eller inte.

Innan vi hittade lösningen med att använda oss av Hibernates @ColumnTransformer tillsammans med PostgreSQL pgcrypto bibliotek kollade vi på ytterligare två möjligheter. Den första var att kryptera data vid insättning, och vid sökning dekryptera hela databasen i ett lager utanför databasen. Alla rader skulle sedan läggas in okrypterade i en ny temporär databas. Därefter skulle sökningen utföras på den temporära databasen med okrypterad data och returnera resultatet till användaren för att sedan slänga den temporära databasen. Det ansågs dock vara en onödigt resurskrävande och komplicerad lösning.

Den andra eventuella lösningen var att kryptera och jämföra data i ett lager utanför databasen samtidigt som man tolkar en query i datalagret. Om den skulle vara av sådan karaktär att betydelsen av värdena i vissa kolumner är viktiga så kan man dekryptera alla värden i dessa kolumner för att kunna göra jämförelser och sedan returnera resultatet.

5.2 Val av verktyg

Java valdes som språk av två anledningar, den ena för att det är språket som vi båda är mest bekväma med och det andra för att vi i tidigare projekt använt java för att hämta och presentera data från en databas och vi kände att det gav oss ett litet försprång.

Med tanke på att den viktigaste biten av arbetet är krypteringen av data i databasen, samt att kunna göra dynamiska queries mot databasen valde vi att inte lägga fokus på att lära oss något mer anpassat ramverk och språk för att ta fram användargränssnittet. Vi hade tidigare

jobbat med Swift för Java och kände att funktionaliteten och utseendet vi skulle få ut komponera ett användargränssnitt med hjälp av Eclipse WindowBuilder skulle vara tillräcklig.

Vi valde Spring Boot för att det är industristandard för att skapa REST API med Java och vi tyckte att det skulle vara ett nyttigt och värdefullt verktyg att ha jobbat med i framtiden. Mycket av Spring Boots färdiga lösningar och bibliotek underlättade arbetet samtidigt som mycket av tiden har gått åt till att förstå sig på hur de olika delarna i ramverket hänger ihop och hur vi skulle anpassa dessa färdiga lösningar till vårt problem.

Valen att använda sig av Hibernate som ORM och PostgreSQL som RDBMS för hur dessa tillsammans skulle hjälpa till med implementationen för lösningen. PostgreSQL var dessutom den RDBMS som vi använt oss av mest tidigare.

5.3 Diskussion relaterat till resultat

5.3.1 Klient

Eftersom huvudsyftet med arbetet var att tackla problemet ur en säkerhetssynpunkt var det aldrig viktigt för oss att skapa ett avancerat användargränssnitt. Vi bestämde oss tidigt nöja oss med att skapa ett för användaren intuitivt gränssnitt som utför jobbet att ta emot input och presentera resultatet. Vi är därför nöjda med klientapplikationen även om det finns mycket man skulle kunnat göra bättre, exempelvis implementera en filterfunktion i tabellen som visar sökresultatet för att lättare navigera bland resultaten. Det finns också utrymme för att presentera svarskoden från REST API:t, alternativt söktiden någonstans i GUI:t.

5.3.2 Server

Servern är ett fungerade REST API och svarar på HTTPS-anrop och sköter flödet mellan anropande klienter och databasen. Servern är inte på något sätt beroende av klienten och vilken applikation som helst som kan skicka HTTPS-anrop kan koppla upp sig mot servern. Eftersom servern är en Spring Boot applikation så är det mycket som är standardiserat och sker bakom huven och det är därför inte mycket som hade kunnat implementeras på ett annat sätt.

5.3.3 Databas

Databasen fyller funktionen av att representera användare med ett unikt id, namn, efternamn och ålder och är en förenklad version av en databas. Den består av endast en tabell med fyra olika kolumner och det hade varit intressant för problembilden att se hur applikationen hade fungerat med en mer realistisk databas, med sökningar som sträcker sig över flera olika tabeller med ett större antal kolumner.

5.3.4 Testning

Det finns mycket annan testning att önska utöver den vi gjort på applikationen, antalet fall i en industriell databas som används för att lagra personuppgifter ligger ofta på två miljoner rader medan vi endast testade på tvåhundra tusen rader. Det hade varit intressant att se hur applikationen hade betett sig på en datamängd som motsvarar ett verkligt användarscenario.

Det hade även varit intressant och se hur tiderna hade förändrats för sökningar över flera olika tabeller med större antal kolumner. Det är svårt att ge ett konkret svar huruvida vi är nöjda med resultaten som applikationen levererar för den krypterade databasen eftersom en tidsökning för sökning var väntad, men huruvida prestandan är tillräcklig är en bedömningsfråga för varje enskilt fall. Med de resultat som finns idag kan en eventuell produktägare fatta ett beslut om huruvida implementationen är värdefull. Ett företag i behov av en säkrad data kan fatta ett beslut om huruvida de tidsförluster en krypterad databas medför är värd den ökade säkerheten. Likaså kan företaget göra en avvägning om vilka kolumner som är nödvändiga att kryptera jämfört med hur mycket tiden för en sökning ökar.

5.3.5 Säkerhet

Det finns direkta säkerhetsrisker som vi inte har tagit hänsyn till när vi tagit fram applikationen. Just nu finns nyckeln som används vid krypteringen hårdkodad och synlig i källkoden. Det skulle kunna lösas till exempel med att ha en ruta i användargränssnittet där man anger vilken nyckel man vill använda sig av för att kryptera och dekryptera raderna, detta hade dock medfört en annan säkerhetsrisk med att överföra nyckeln från klient till servern. En annan direkt säkerhetsrisk är att databasen utför krypteringen och därför måste ta emot nyckeln och data i klartext i SQL-format, vilket betyder att datan kan hamna i serverloggar.

6 Slutsats

Sett till resultaten av volymtesterna blir det relevant att lyfta ytterligare en frågeställning om huruvida vår modell för fullständig kryptering av en databas överhuvudtaget är värdefull. De stora ökningarna i transaktionstider gör systemet opraktiskt i industriella sammanhang där databaserna är av betydligt större storlek, högre komplexitet och trafiken gentemot databasen är högre. Vårt system skulle kunna vara av värde i sammanhang där krav på säkerhet är hög men datamängderna är små och trafiken låg.

Utifrån den nya frågeställningen blir det också relevant att analysera den faktiska säkerheten i vår modell. Under projektets gång har kryptering av data varit i primärt fokus, men det finns fler aspekter att ta hänsyn till när det kommer till datasäkerhet. Att framgångsrikt kryptera data och låta den skickas över en säker kanal är bara en dellösning på ett större problem.

Under projektets gång har mängder av eventuella risker uppkommit som vi inte har tagit hänsyn till i vår modell, så som osanitär kod i klientlagret, åtkomstkontroll av databasen via klienten, autentisering av klienten och hotmodeller. Det finns alltså mängder av variabler att ta hänsyn till som inte tagits med men som är viktiga för den totala säkerheten i ett system.

Så trots att målen för projektet är uppnådda krävs det att man antar ett större perspektiv på vad datasäkerhet faktiskt är och hur den uppnås för att lösa problemet i sin helhet. För att lösningen ska vara värdefull krävs också att man tar i beaktning hur säkerhet ska uppnås utan att förlora för mycket prestanda. Det alltså finns gott om utrymme för vidareutveckling av vårt system, men framförallt krävs ett utvidgande av själva problembilden.

6.1 Vidareutveckling

6.1.1 Säkerhetsrisker

Eftersom syftet med arbetet att är höja säkerheten när man sparar data i en relationsdatabas är blir är det intressant att diskutera ytterligare säkerhetsrelaterade aspekter av applikationen. Det finns vissa säkerhetsrisker som vi inte har haft tid för att utforska t.ex. hur vidare applikationen är säker emot SQL-injections eller side channel attacks. I ett fortsatt arbete mot en “säker” databas hade det varit intressant att utforska applikationens öppenhet gentemot dessa och i så fall utforska eventuella möjligheter att täppa till dessa säkerhetshål.

6.1.2 Icke-proprietär lösning

Lösningen är idag proprietär till PostgreSQL, det hade varit intressant men lösning som man kan koppla emot flera olika implementationer av relationsdatabaser. Hibernate som används som ORM på servern fungerar till många andra typer av relationsdatabaser vilket gör den till ett lämpligt val av ORM ifall man skulle försöka koppla upp servern mot en annan typ av databas. Detta hindras idag dock av att krypteringen sker i PostgreSQL, en eventuell

vidareutveckling mot en icke-proprietär lösning hade förmodligen inneburit att man lät krypteringen ske i serverlagret.

6.1.3 Mutual trust

Nuvarande version av applikationen skickar data över HTTPS som är en krypterad och säker kommunikationskanal. Däremot tillåts att vem som helst kommunicera med databasen via REST API:t så länge man har vetskap om dess URL och medföljande endpoints. Alltså kan vem som helst med rätt information göra hämtningar från, och manipulera databasen.

I en framtida version skulle certifikat även appliceras på klientsidan för att upprätta ett Mutual Trust förhållande mellan klient och server. Det innebär att servern konfigureras så att den kräver ett specifikt certifikat för att möjliggöra kommunikation, i detta fallet certifikatet som kopplats till klienten.

I det fall då applikationen har flera användare finns det även möjlighet för att utse roller, för olika nivåer av tillgång. Exempelvis kan servern konfigureras så att endast användare med en administrativ roll ska få lov att manipulera databasen och att vanliga användare endast ska tillåtas göra hämtningar.

6.1.4 Pålitliga certifikat

Eftersom applikationen i dagsläget inte ska lanseras används självsignerade certifikat, eftersom de är gratis och räcker väl för en proof-of-concept. Det medför risker vid skarp användning eftersom klienten inte med säkerhet kan fastställa serverns pålitlighet. En framtida lösning är att få ett pålitligt certifikat utfärdat från en CA och därmed upprätta en säker HTTPS-anslutning mellan klient och server.

Källhänvisning

[1] K. Ferguson, C. McKenzie, "REST (REpresentational State Transfer," 2020. [Online]. Tillgänglig: https://searchapparchitecture.techtarget.com/definition/REST-REpresentational-State-Transfer, hämtad: 2020-03-20.

[2] C. Bedell, E. Hannan, S. Wilson, "RESTful API (REST API)," 2020. [Online]. Tillgänglig: https://searchapparchitecture.techtarget.com/definition/RESTful-API, hämtad: 2020-03-30.

[3] J. Katz, *Introduction to Modern Cryptography.* 2nd ed., Boca Raton, USA: Chapman & Hall/CRC, 2014, pp. 3.

[4] T. Dierks, E. Rescorla, "The Transport Layer Security (TLS) Protocol version 1.2," 2008. [Online]. Tillgänglig: https://www.ietf.org/rfc/rfc5246.txt, hämtad: 2020-04-07.

[5] Oracle, "Public Keys, Private Keys, and Certificates," 2010. [Online]. Tillgänglig: https://docs.oracle.com/cd/E19509-01/820-3503/ggbgc/index.html, hämtad: 2020-04-15

[6] Oracle, "keytool - Key and Certificate Management Tool," NA. [Online]. Tillgänglig: https://docs.oracle.com/javase/7/docs/technotes/tools/solaris/keytool.html, hämtad: 2020-04-15

[7] K. Siva Prasad Reddy, *Spring Boot 2: Applications and Microservices with the Spring Framework.*, Hyderabad, India: Apress, 2017, pp. 1-10.

[8] Eclipse Foundation, "Eclipse documentation - Current Release Eclipse IDE 2020-03," 2020 [Online]. Tillgänglig: https://help.eclipse.org/2020-03/index.jsp, hämtad: 2020-04-15.

[9] The PostgreSQL Global Development Group, "PostgreSQL 12.3 Documentation," 2020 [Online]. Tillgänglig: https://www.postgresql.org/files/documentation/pdf/12/postgresql-12-A4.pdf, hämtad: 2020-04-02

[10] Hibernate, "Hibernate ORM - Your relational data. Objectively.," NA [Online]. Tillgänglig: https://hibernate.org/orm/, hämtad: 2020-04-20

[11] B. Varanasi,*Introducing Maven: A Build Tool for Today's Java Developers* 2nd ed., Salt Lake City, UT, USA: Apress, 2019, pp. 1-5. [Online]. Tillgänglig: https://link-springer-com.proxy.lib.chalmers.se/content/pdf/10.1007%2F978-1-4842-5410-3.pdf, hämtad: 2020-05-01

[12] R. Chan, "The 10 most popular programming languages, according to the ‚Facebook for programmers'," 2019. *Business Insider*. [Online]. Tillgänglig: https://www.businessinsider.de/international/the-10-most-popular-programming-languages-according-to-github-2018-10/?op=1, hämtad: 2020-05-05

[13] The PostgreSQL Global Development Group, “PostgreSQL: Documentation: 9.4 pgcrypto,” 2020. [Online]. Tillgänglig: https://www.postgresql.org/docs/9.4/pgcrypto.html, hämtad: 2020-04-03.

[14] H. Dobbertin V. Rijmen A. Sowa, *Advanced Encryption Standard - AES: 4th International Conference, AES 2004, Bonn, Germany, May 10-12, 2004, Revised Selected and Invited Papers.* Berlin, Germany: Springer, Berlin, Heidelberg, 2005, pp. VI. [Online]. Tillgänglig: https://link-springer-com.proxy.lib.chalmers.se/content/pdf/10.1007%2Fb137765.pdf, hämtad: 2020-04-12.

[15] W. Stallings, *Cryptography and Network Security Principles and Practice*, 7th ed., Harlow, England: Pearson Education Limited, 2017, pp. 294-297

[16] E. Rescorla, A. Schiffman, "The Secure HyperText Transfer Protocol," 1999. [Online]. Tillgänglig: https://tools.ietf.org/html/rfc2660, hämtad: 2020-05-14

[17] Encyclopædia Britannica ImageQuest, ”Computer security,” 2016. [Elektronisk bild]. Tillgänglig: https://quest-eb-com.proxy.lib.chalmers.se/search/computer-security/1/132_1308916/Computer-security/more. Hämtad: 2020-05-29.