CHALMERS

Analys av dataloggar

Kategorisering och prioritering av loggar

Examensarbete inom Högskoleingenjörsprogrammet i datateknik

Ola Andersson

Daniel Edsinger

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2016

Examensarbete 2016

Analys av dataloggar

Kategorisering och prioritering av loggar

OLA ANDERSSON
DANIEL EDSINGER

CHALMERS

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2016

Analys av dataloggar
Kategorisering och prioritering av loggar
OLA ANDERSSON
DANIEL EDSINGER

© OLA ANDERSSON, DANIEL EDSINGER 2016.

Handledare: EMIL AXELSSON, Chalmers
Handledare: MIKAEL TÖNNBERG, Antura
Examinator: CHRISTER CARLSSON, Department of Computer Science and Engineering

Examensarbete 2016
Software Technology
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2016

Analys av dataloggar
Kategorisering och prioritering av loggar

OLA ANDERSSON
DANIEL EDSINGER
Department of Computer Science and Engineering
Chalmers University of Technology

Sammanfattning

Detta projekt är till för att underlätta Anturas felsökningsprocess genom att på ett lämpligt sätt tillämpa en algoritm som ska göra det enkelt för Antura att behandla de mest kritiska felen i deras produkt.

När företag ska felsöka buggar och fel i sina mjukvaror efter hundratals felmeddelanden från sina kunder, görs det i en långsam process genom att läsa igenom dem en efter en. Deras innehåll hjälper till att felsöka problemet och möjligtvis återskapa felet. Detta arbetsätt är mycket vanligt hos företag som utvecklar applikationer, även hos Antura, företaget som arbetet gjordes för.

Det vi har gjort är att förenkla denna process med ett program som med hjälp av Graylog, kunna prioritera och kategorisera alla loggar som skickas från kunderna. Vår slutprodukt består av: ett plugin till Graylog, en databas och ett externt program som heter Titan. Pluginet analyserar alla loggar som kommer in till Graylog och skickar dessa till databasen. Titans uppgifter är att visa alla loggar i olika kategoriseringar och prioritera dessa beroende på dess innehåll.

Nyckelord: Graylog, loggar, plugin, databas, externt program, Titan.

Förord

Detta examensarbete är skrivit av Ola Andersson och Daniel Edsinger som för närvarande är studenter på Chalmers Tekniska Högskola. Arbetet är framtaget åt Antura Projects vägnar som en förenkling av deras felhanteringsprocess.

Vi vill passa på att tacka alla som har varit involverade i detta projekt, från början till start. Det har varit en oerhört lärorik och spännande period med många timmars planering, kodande och rapportskrivande. Vi vill speciellt tacka Mikael Tönnberg på Antura som alltid sett till att vi haft de verktyg för att fullborda uppgiften, men även klargjort många av de frågor vi haft på vägen. Vi vill även tacka Emil Axelsson för hans stöd genom rapportskrivandet, kommentarer och nödvändiga ändringar som har hjälp till att skapa denna rapport. Till sist vill vi passa på att tacka Antura än en gång för all den kunskap ni har bistått med och all support de anställda på Antura har gett.

Ola Andersson, Daniel Edsinger, Göteborg, Juni 2016

Betecklingar

GELF-client Graylog Extended Log Format

IDE Integrated Development Enviroment

JDBC Driver Java Database Connectivity Driver

JSON JavaScript Object Notation

REST-API Representational State Transfer Application Programming Interface

SQL Structured Query Language

XML Extensible Markup Language

Innehåll

1 Introduktion **1**
1.1 Bakgrund . . . 1
1.2 Syfte . . . 1
1.3 Mål . . . 2
1.4 Avgränsningar . . . 2

2 Metod **3**
2.1 Inledning . . . 3
2.2 Val av språk . . . 3
2.3 Debugging och felsökning . . . 4
2.4 Versionshantering . . . 4

3 Teknisk bakgrund **5**
3.1 Graylog2 . . . 5
3.1.1 Programstruktur . . . 5
3.1.2 Plugins . . . 6
3.2 Microsoft .NET Framework . . . 6
3.2.1 C# . . . 6
3.2.2 Visual Studio . . . 6
3.3 Maven . . . 7
3.4 GELF-client . . . 7
3.5 Bitbucket/Git . . . 7

4 Genomförande **8**
4.1 Plugin eller REST API . . . 8
4.2 Prioriteringar och möjliga överblickar . . . 9
4.2.1 Kategorisering . . . 9
4.2.2 Översikt över felmeddelanden . . . 10
4.2.3 Prioriteringar . . . 10
4.3 Utveckling av plugin . . . 10
4.3.1 Analysklasser . . . 11
4.3.2 Dynamiska inställningar . . . 12
4.3.3 Testning/Produktverifiering . . . 12
4.4 Utveckling av externt program . . . 12
4.4.1 GraylogFilter . . . 13
4.4.2 Titan . . . 14
4.4.2.1 ILogsFilter . . . 14
4.4.2.2 TreeView med Stacktrace . . . 15

4.5 Databasen 16

5 Resultat **17**
5.1 Kategorisering 17
5.2 Plugin till externt program 18
5.3 Produktverifiering 18

6 Diskussion **19**
6.1 Baskunskaper och svårigheter 19
6.2 Arbetsuppdelning 19
6.3 Kritik mot arbetet 19
6.4 Hållbarhetsaspekt 20

7 Slutsats **21**

Bibliography **21**

A Database **I**
A.1 Diagram I
A.2 Schema II

1
Introduktion

1.1 Bakgrund

Antura är ett mindre men växande företag, som utvecklar en mjukvaruapplikation. Antura Projects är en helhetslösning för projekt-, portföljs- och resurshantering för företag. Deras affärsidé går ut på att hyra ut nämnda program, som en tjänst mot en månadsavgift baserat på antalet användare på den arbetsplats som hyr tjänsten.

Allt eftersom programmets komplexitet ökar i takt med dess storlek, blir risken att fel inträffar under dess exekvering större, vilket kan orsaka diverse problem. För att komma underfund om vad det är som orsakar programmets felaktiga beteende, krävs en felsökning av koden. I mjukvarumarknaden är det viktigt att kunden är så nöjd som möjligt med sin produkt och det är då viktigt att reda ut alla problem snabbt och effektivt. Buggar kan hittas under testningen av programmet innan kunden finnner dem. Däremot om buggen uppstår på klientens sida behöver så mycket information som möjligt om felet skickas till utvecklarna för granskning.

Effektivisering är viktigt för att vara ett framstående och ledande företag inom IT-branschen. Dagens programvaror blir allt mer komplexa och medför att felhanteringsprocessen blir svårare att hantera. Ett drömscenario för alla utvecklare är ett program utan buggar, men detta är i praktiken omöjligt. Frågan som således uppstår är hur man kan skapa ett hjälpmedel som säger åt en utvecklare vilka buggar som ska prioriteras, men också vilka buggar som är mindre viktiga.

1.2 Syfte

Detta projekt är till för att utveckla ett hjälpmedel vars ändamål är att underlätta Anturas felhanteringsprocess. Målet är att de enklare ska kunna prioritera fel efter kritisk nivå, samt hur ofta felen har inträffat och andra relevanta kategorier. I slutändan ska resultatet innebära att Antura snabbare får en bättre överblick över varje fel som loggas och med hjälp av detta kan behandla felen utefter kritisk ståndpunkt och antal användare som har drabbats.

1.3 Mål

Under projektets gång ska det skapas en applikation/mjukvara som ska integreras med Antura Projects befintliga system. Denna mjukvara ska hantera felmeddelanden och kunna kategorisera loggarna efter: antal inträffanden, de kunder som är påverkade och vilken del av programmet som är drabbat. Innan den slutgiltiga produkten överlämnas, skall tester utföras för att kunna verifiera att produkten fungerar enligt ovan nämnda funktionaliteter och krav.

Det sista målet, men även det som avgör projektets framgång, är att utöver de ovannämnda metoderna att visa data, ska det utvecklade programmet ta reda på och analysera vad som är viktigt och ska prioriteras högre än resten. För att applikationen ska kunna klara av den uppgiften, gäller det att undersöka vad loggarna innehåller och hur man kan avgöra värdet av dess innehåll. Majoriteten av informationen om själva felet kommer att befinna sig i dess stacktrace, om den är närvarande vilket inte alltid är fallet. Därför filtreras loggar ut som inte innehåller kritisk information för att göra en fullständig analys av loggen.

Vidareutveckling av analys-applikationen måste tas i åtanke under utvecklingen. Programmens struktur skall vara lättåtkomlig för att nya funktioner ska läggas till. Önskvärt är om endast en rad behövs läggas till i redan existerande kod för att infoga en enkel funktion på det färdigutvecklade programmet.

1.4 Avgränsningar

Det kommer inte ske någon ändring av den befintliga koden i Antura Projects, utan all kod som skrivs ska vara självständig och fungera innan den integreras med huvudprogrammet. Någon form av hårdvarubaserad kodning kommer ej ske. Projektets omfång innefattar endast mjukvara.

2
Metod

2.1 Inledning

Insamling av loggar hos Antura görs med hjälp av Graylog [1]. Anturas program Antura Projects, skickar en logg vid varje fel som inträffar hos en av deras kunder. Denna logg samlar Graylog-servern in. För debugging undersöker programmerarna meddelandet för att sedan lösa felet. När många meddelanden kommer blir det tyvärr svårt att enkelt gå igenom alla loggar på ett effektivt sätt, vilket leder till ett behov av ett annat alternativ.

Antura kom fram med fyra föreslag på hur uppgiften kan lösas som en startpunkt. För att göra sig underrättad om vilken lösning som är den mest effektiva, krävs en grundlig undersökning om vad lösningen innebär och hur det kan integreras med Graylog. Det måste också tas i åtanke om den lämpar sig bäst till Antura och hur de senare kan distribuera programmet i deras miljö. Vid samtal med Antura framtogs två metoder som är lämpligast att användas för projektet. Den första metoden är att skapa ett plugin till Graylog som körs tillsammans med själva servern. Lösningen innefattar att för varje logg som skickas till servern, så ska den lägga in och möjligtvis sortera informationen på en realtionsdatabas. På det sättet uppdateras informationen i realtid och blir direktintegrerad med Graylog och körs parallellt. Detta är möjligt med en Output-plugin till Graylog som låter oss vidarebefordra varje meddelande som kommer [2].

Den andra metoden består av att ladda ner loggar som JSON-objekt direkt från Graylog-servern med hjälp av ett REST API. De två möjligheter som finns är att sedan analysera dem antingen i en extern applikation eller i den redan befintliga applikationen som ska visa datan grafiskt. Detta generar en tyngre belastning än ett plugin, men är samtidigt enklare att uppdatera eftersom att den är självständig. I fallet att det blir en egen applikation utvecklas programmet i C#.NET i en Windows-miljö med hjälp av Microsofts egna IDE, Visual Studio. Ur Anturas synvinkel hade en databas varit fördelaktigt på så sätt att de själva kan bestämma hur applikationen, vars syfte är att visa datan, ska utformas rent grafiskt.

2.2 Val av språk

Språken som kommer att användas under utvecklingen är Java för utvecklingen av pluginet, SQL för relationsdatabasen och C# för programmering av programmet som ska visa det grafiska. I fallet för plugin är det ej möjligt att utveckla i något annat språk än Java eftersom Graylog själv är Java-baserat från grunden. C# valdes p.g.a. att Antura huvudsakligen utvecklar sina egna program i C#.

2.3 Debugging och felsökning

Eftersom ett plugin måste installeras på en avlägsen dator till ett redan körande program, finns det inte möjligheter att testa programmets pålitlighet på ett standardiserat sätt, exempelvis skriva ut till en kontrollpanel med felmeddelanden och en stacktrace. På grund av detta ansågs GELF-client, ett open-source program för Graylog, vara ett alternativ. När ett meddelande skickas till Graylog kommer färdiga meddelanden i koden för pluginet att skickas till Graylog igen. Dessa är till för att se vilka delar av koden som exekveras efter vilka meddelanden som skickas. Det kommer även i utvecklingsmiljön Eclipse kunna göras JUnit-tester, men dessa kommer att vara begränsade eftersom den riktiga servern fungerar annorlunda. Där måste alla delar kunna arbeta tillsammans och kunna hantera hela meddelandet, inte bara klass för klass.

2.4 Versionshantering

För versionshantering kommer Bitbucket att användas som ett alternativ till GitHub eftersom Antura själva vill kunna kontrollera kodens tillgänglighet. Antura kommer att tillförse en Bitbucket-server som endast kommer att nås lokalt från deras kontor.

3
Teknisk bakgrund

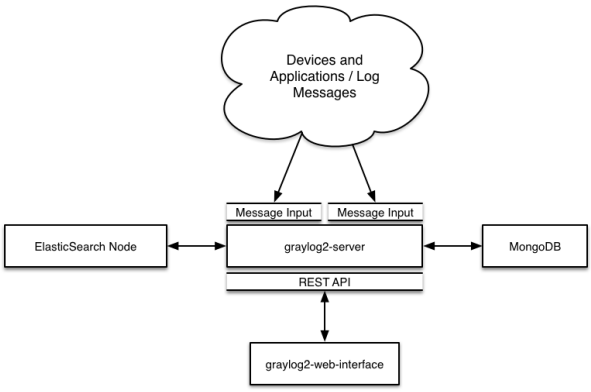
3.1 Graylog2

Graylog är ett kraftfullt Java-baserat program som genom att använda centralisering, på ett effektivt sätt kan samla och lagra loggar som skickas till en server. Det medför att man enkelt kan nå och hitta de individuella loggarna och all information som de innehåller. Det finns även ett hjälpmedel för att se statistik över loggarnas information. Detta gör Graylog till ett passande verktyg, om det utvecklade programmet implementerar den funktionen, för insamling av fel som inträffar i andra program. Det kan hjälpa utvecklare att enkelt nå alla loggar på en enda plats för att sedan åtgärda det påträffade felet.

3.1.1 Programstruktur

Grundprincipen bakom Graylog, som illustreras i Figur 3.1, består av flera delar som tillsammans bygger upp anläggningen. En applikation sänder iväg en logg till Graylog-servern som tar emot loggen i ett av dess Input, se förklaring i avsnitt 3.1.2 om vad det innebär. Inmatningen behandlas därefter och lagras i en MongoDB-databas som JSON-objekt [3].

Graylogs REST API är en resurs som tillåter andra applikationer att få hämta alla JSON-objekt som är lagrade i databasen. Som en standard använder sig Graylogs web-gränsnitt till stor del av den för att sedan grafiskt kunna visa det för användaren, men även självstående applikationer kan utnyttja de funktionaliteter som Graylog erbjuder. Detta kan göras genom att ansluta till Graylog servern via en URL som hämtar, beroende på dess innehåll, ett JSON-objekt som man behandlar efter förfrågan. Detta är möjligt att göra i nästan vilken utvecklingsmiljö som helst.



Figur 3.1: *Graylog uppbyggnad*

3.1.2 Plugins

Graylog innehåller ett brett stöd för utvecklare att kunna implementera sitt egna plugin. När ett plugin skapas i Graylog, integreras det med själva huvudprogrammet och körs så länge programmet är igång, vilket i detta fall är Graylog-servern. Syftet med ett plugin är att utvecklaren ska kunna skapa ett tillägg i programmet vars funktionalitet inte existerar som standard, utan den är skräddarsydd efter behov och utvecklas därefter. De plugin som går att skapa visas i Tabell 3.1 [2]. Det plugin som lämpar sig för detta projekt kommer att vara av typen *Output*. En *Output* gör det möjligt för Graylog att vidarebefordra meddelandets innehåll, exempelvis till en databas.

Tabell 3.1: De olika typer av plugin som går att skapa till Graylog2.

Input	Skriver ett meddelande till Graylog2.
Output	Skickar vidare ett meddelande till en annan mjukvara eller tjänst.
Services	Körs vid uppstart och kan innehålla all typ av funktionalitet.
Alarm callbacks	Aktiveras när ett alarm har skett i en ström i Graylog2.
Filters	Transformerar/slänger ingående meddelanden med givna villkor.
REST API Resources	En REST-resurs för att tillhandahålla alla resurser som finns med i Graylog2:s REST API.
Periodical	Ett plugin som körs med en periodisk angiven exekveringstid.

3.2 Microsoft .NET Framework

.NET är ett ramverk tillhörande operativsystemet Microsoft Windows och består av komponenter vars uppgift är att hantera exekveringen av program som är skriva för just .NET. Dess innehåll är bland annat ett stort klassbibliotek som tillhandahåller kodade lösningar inom programmering såsom kryptografi och algoritmer.

3.2.1 C#

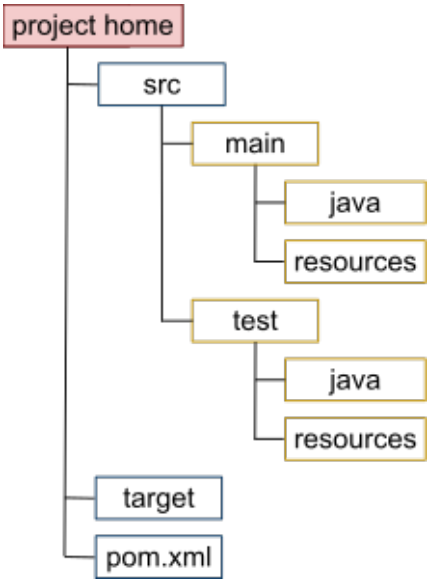
C# är ett objektorienterat programmeringsspråk som finns med i .NET. Eftersom att det är objektorienterat är det inte svårt att gå från ett annat liknande språk, till exempel Java, då det i grund och botten är väldigt likt att koda i.

3.2.2 Visual Studio

Visual Studio är en programutvecklingsmiljö skapad av Microsoft. Med detta verktyg är det möjligt att skapa Windows- och internetbaserade applikationer. Visual Studio tillhandahåller en del programmeringsspråk, men det mest använda och som även kommer att användas i detta projekt är C#.

3.3 Maven

Apache Maven används inom systemutveckling av mjukvaruprogram skrivna i Java, som ett verktyg för att paketera programmet till en distribuerbar fil. Maven innehåller ett bibliotek som består av flera källor. Dessa källor består av jar-filer som Maven-projekt kan hämta ner och använda för kompilering. De källor som ska användas specifieras i en POM-fil (Project Object Model). Detta är en XML-fil som beskriver projektet för Maven och hur jar-filen ska byggas upp. Projektet måste byggas upp enligt en viss modell som visas i Figur 3.2, för att kunna paketera filen korrekt och även för att kunna utföra eventuella tester [4].



Figur 3.2: *Maven projekt struktur*

3.4 GELF-client

GELF-client [5] gör det möjligt att skicka meddelanden efter ett visst mönster till en Graylog-server och därefter visa meddelandet som en logg. Fördelen med GELF-client är dess förmåga att skicka meddelanden från en server till sig själv. På det sättet finns det möjligheter att testa programmets funktionaliteter även när det är distribuerat på servern.

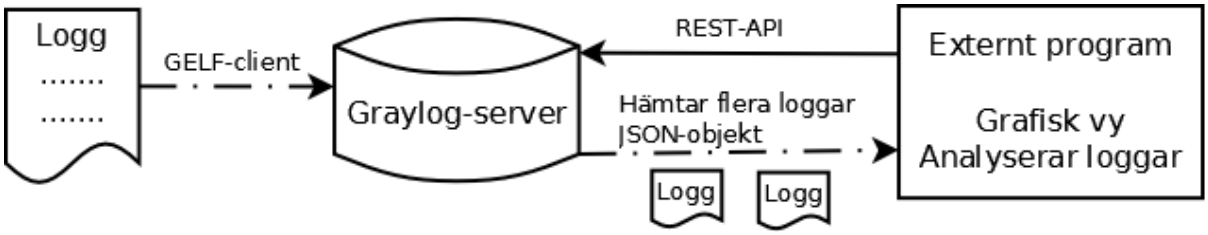
3.5 Bitbucket/Git

Bitbucket är ett webbhotell för versionshantering av kodprojekt [6] och är lämpat för privat utveckling. Till skillnad från GitHub kan man skapa en egen Bitbucket-server på sitt nätverk. Eftersom Antura vill ha koden privat så är Bitbucket är bra verktyg som har samma funktionaliteter som GitHub för att hjälpa till och underlätta utvecklingen.

4
Genomförande

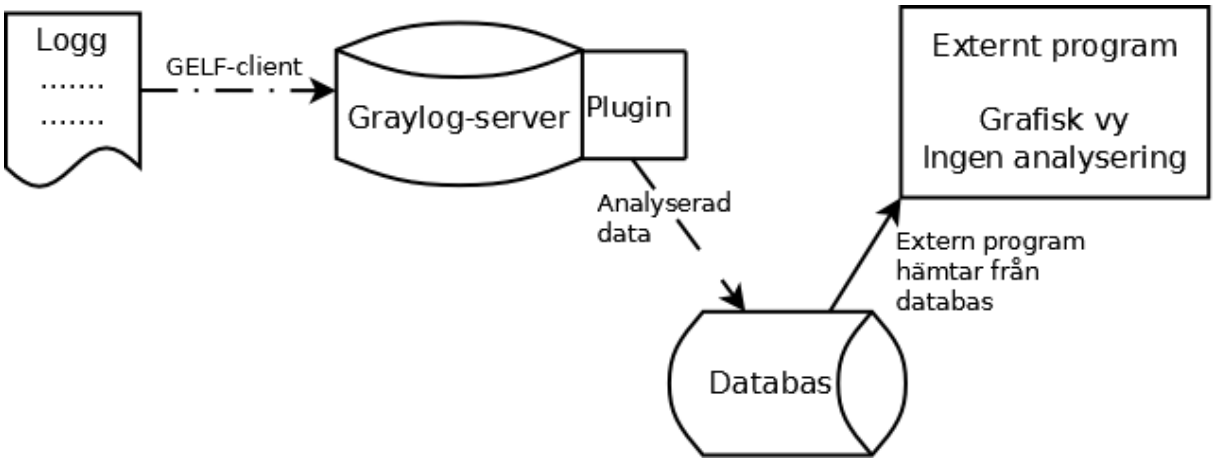
4.1 Plugin eller REST API

I syftet av att skapa optimerad programkod som enkelt går att bygga ut, krävs en del planering för hur man ska gå till väga för att nå önskat resultat. Vare sig man använder ett plugin eller Graylogs REST API, så kan en av dessa lösningar resultera i en färdig produkt som klarar av de specifikationer som förklaras tidigare i avsnitt 1.3. De olika lösningarna har två olika strukturer på hur de kommer att se ut. Strukturen för lösningen med REST API och ett externt program som analyserar all data, visas i Figur 4.1. Den innebär att det externa programmet hämtar hem alla loggar samtidigt från Graylog-servern och analyserar dess innehåll, därefter visas dess resultat för användaren.



Figur 4.1: *Arbetet gjort med REST API utan plugin*

Den andra lösningen som visas i Figur 4.2, innebär att ett plugin till Graylog ska analysera loggarna var för sig och spara informationen i en databas. När det externa programmet sedan vill visa datan, hämtas detta enkelt från databasen. Det har enbart fokus på det grafiska och behöver inte göra något extra arbete till skillnad lösningen med REST API.



Figur 4.2: *Arbetet gjort med plugin*

Under arbetets start testades möjligheterna med ett plugin och REST API parallellt för att se vad som krävs för att skapa ett fungerande program. Det gjordes även som en säkerhetsåtgärd: om den ena lösningen inte fungerade, så fanns det alltid ett alternativ att falla tillbaka på. I slutändan valdes lösningen med ett plugin över REST API.

4.2 Prioriteringar och möjliga överblickar

Den absolut viktigaste delen i projektet är att ta fram ett eller flera sätt att visa data för att underlätta felsökningsprocessen. Det enklaste sättet som går snabbast att utveckla är att visa en enkel lista av varje fel som skickas in. Det är viktigt att den filtrerar ut de överflödiga meddelanden som inte är ett resultat av en bugg. Antura använder sig av Nlog [7] för att skicka meddelanden från sin applikation till Graylog-servern och tillämpar olika nivåer på sina meddelanden [8]. De olika nivåerna beskriver meddelandets betydelse och hur viktigt det är. De meddelanden som inte tillhör gruppen av typen buggar, kan filtreras ut med hjälp av meddelandets nivå på varningen. Det meddelande som pluginet ska hantera ska vara av typen *Error* och översätts till heltalet 3 när man hämtar det från loggen. Figur 4.3 visar en logg hämtad med Graylogs REST API som innehåller information om dess nivå, men även två viktiga komponenter, StackTrace och ExcpetionMessage.


```
{
  "message": {
    "level": 3,
    "StackTrace": "at Antura.Data.Command.ExecInternal(Int32 n)\n   at Antura.Data.Command.Exec(Int32 n)\n
    "streams": [
      "56e19463e4b04c23fd760bf4"
    ],
    "adress": "project.husqvarna.net",
    "source": "DARPA-SRV-WEB",
    "message": "SqlException: \nCommand: TASK_SET_A_Delete, InParameter:@i_RequestUserID (43), OutParameter
    "gl2_source_input": "56e19228e4b04c23fd760988",
    "version": "1.0",
    "ExceptionMessage": "SqlException: \nCommand: TASK_SET_A_Delete, InParameter:@i_RequestUserID (43), Out
    "hostingenv": "Ipeer",
    "full_message": "SqlException: \nCommand: TASK_SET_A_Delete, InParameter:@i_RequestUserID (43), OutPara
    "appid": "antura-projects",
    "displayname": "Husqvarna (Production)",
    "gl2_source_node": "00726e85-cb79-4f6c-8770-18d3368f062b",
    "_id": "5bf84c20-20b4-11e6-8d14-00155d0024a7",
    "facility": "console-runner",
```


Figur 4.3: *Exempel av REST-API meddelande*

4.2.1 Kategorisering

För att kunna göra någon sorts analys av loggarna, krävs det att de kan tillhöra en eller flera kategorier och användaren kan söka efter dessa. Med den information som ett meddelande i Graylog erbjuder, finns det ett fåtal sätt att kategorisera loggarna. Dessa inkluderar vem som har fått felet (displayname) och vilket sorts fel som har skett (ExceptionMessage). Dessa delar är närvarande i exempelmeddelandet i Figur 4.3.

4.2.2 Översikt över felmeddelanden

En stacktrace består av en enda lång sträng med en beskrivning på var i programmet felet skedde, d.v.s. i vilket paket, klass och metod. Delar man upp den efter varje radbyte är det möjligt att göra en visualisering av programmets alla fel och var de har skett. Den lämpligaste metoden för att visa denna informationen i användargränssnittet, är med en trädvy som låter användaren att gå djupare i programmet i de olika delarna som framförs till användaren.

4.2.3 Prioriteringar

En av de viktigaste delarna i projektet är att kunna särskilja på felets betydelse eller vikt och säga hur kritiskt felet är. Detta var det viktigaste kravet som framfördes av Antura. Lösningen till det är att använda kategoriseringen som förklaras i avsnitt 4.2.1, och sätta vikt på de olika kategorierna, d.v.s poängsätta felen beroende på vilken sorts fel det är, och hur högt Antura värdesätter/prioriterar kunden.

4.3 Utveckling av plugin

Ett stort hinder för utvecklingen av pluginet var att komma underfund hur det faktiskt fungerar och hur det kan distribueras till en avlägsen dator. Det finns mycket information från Graylogs officiella sida med dokumentation, men programmet har inte en omfattande användarbas. Det ledde till att inlärningstiden ökade och att problemlösningen under utvecklingen var begränsad. Många problem som att kunna kompilera och testa koden fördröjde arbetet mer än vad tidigare tänkt.
Strukturen av ett plugin måste följa ett mönster som kommer i en form av ett interface. De olika sorters plugin, som visas i Tabell 3.1, har alla olika strukturer och metoder i sin kod. I fallet för en *Output* Plugin, finns två viktiga metoder:

- `void write(Message message);`
- `void write(List<Message> messages);`

Dessa två metoder får loggar inmatade och ska därefter behandla dem för att sedan kunna vidarebefordra datan till databasen. För att kunna göra detta behövs en anslutning till databasen. Anslutningen görs med hjälp av en JDBC Driver för den sortens databas som ska användas mellan programmet och databasen. Som en följd av tidigare kännedom och erfarenheter, ansågs en MySQL databas vara passande. För att ett kompilerat program ska registrera en JDBC Driver under exekvering, måste en Classpath (en sökväg för externa jar-filer) defineras i den kompilerade jar-filens Manifest-fil. Det är att specificera vart programmet ska läsa in nämnda driver för att sedan lyckas skapa en anslutning med databasen. För att under utvecklingen göra det smidigare för testning av programmet, fastställdes platsen som drivern är placerad med hjälp av POM-filen som kompilerar projektet.
Anslutningen till databasen sker när pluginet skapas, d.v.s när det första meddelandet kommer sedan pluginet lades in i servern. Detta gör den med hjälp av de inställningar som beskrivs senare i kapitel 4.3.2.

4.3.1 Analysklasser

För att underlätta vidareutveckling av programmet, behövde hela utvecklingsprocessen vara så dynamisk som möjligt. Målet var att göra det möjligt att bygga programmet genom att dynamiskt ladda in alla klasser som är av en specifik typ. Det låter nästa utvecklare som ska arbeta med projektet lägga till funtionaliteter utan att röra grundkoden. Det gör det även lättare och snabbare att skapa nya analysklasser eftersom koden är mer avskild från resten av programmet. Senare i detta avsnittet står beskrivningar om vad de olika klasserna ska göra och analysera. De fält som kommer analyseras från början är följande:

- id: Alla loggar har ett unikt id som gör det enkelt att söka efter loggen i Graylog2:s REST API. Detta utnyttjas även av databasen då det är enkelt att koppla ihop varje logg med id samt ett annat attribut, till exempel stacktrace.
- occurred: Representerar ett tidstämpel då loggen registrerades av Graylog2.
- stacktrace: Stacktrace delas upp i databasen med splittring på ny rad (\n). Detta stacktrace representerar den översta raden, alltså den rad som är "djupast" i programmet.
- displayname: Företagets namn.
- exception: Vilken typ av exception som har sparats undan för loggen.
- appid: Appid för loggen. Från vilket program av Antura som loggen skickas från.

Om programmet dynamiskt ska ladda in rätt klasser och utföra analysen med reflektion, måste klasserna innehålla ett specifik interface: MessageAnalyzer. Denna klass utvecklades specifikt för detta arbete. Även om reflektion i program kan vara opålitligt, så är det den bästa lösningen för att göra allt så dynamiskt som möjligt. Det ger möjlighet att inte behöva modifiera redan existerande kod när extra funktioner ska läggas till. Annars är man tvungen att ladda in klasserna genom hårdkodning. Alla de klasser som har utvecklas för pluginet listas här nedan:

- IDAnalyzer: Hämtar meddelandets id och tidsstämpel.
- StacktraceAnalyzer: Eftersom att meddelandets stacktrace kommer i en enda lång sträng, så behöver den delas upp för att sedan läggas upp till databasen.
- DisplaynameAnalyzer: Läser från vem meddelandet kommer ifrån.
- ExceptionMessageAnalyzer: Analyzera efter nyckelord i meddelandets Exceptionmessage och läggas i en kategori eller flera efter det.
- AppidAnalyzer: Kolla från vilken applikation som felet kommer ifrån.

Alla de olika klasserna, förutom IDAnalyzer, kommer i databasen koppla den data klassen behandlar, till meddelandets id för senare återkoppling i det externa programmet. Dessa klasserna bygger i sin tur upp databasen. Meningen är att klasserna ska ta hand om en del i loggen var för en smidig uppdelning av arbetet och enkelt kunna skapa nya, byta ut delar eller ersätta dem helt. Den viktigaste delen i en logg är hur dess Stacktrace ser ut, då detta ger var felet i koden ligger.

4.3.2 Dynamiska inställningar

För att kunna applicera ett plugin på flera system behöver vissa variabler vara utbytbara för att exempelvis ändra inloggningsuppgifter till databasen och dess adress, eller vilken JDBC Driver man tänker använda sig av beroende på vilken sorts databas som används. För detta använder Graylog plugins en intern config-klass som visas i Figur 4.4. Denna är helt dynamisk och låter utvecklaren infoga obligatoriska som valfria inställningar som matas in när pluginet skapas. För detta projektet behövdes just de inställningar som är närvarande i Figur 4.4. Resultatet blir att när man installerar pluginet, finns fält som användaren kan skriva in sina inställningar för att pluginet ska fungera på just deras servers konfiguration.

```
public static class Config extends MessageOutput.Config {
    @Override
    public ConfigurationRequest getRequestedConfiguration() {
        final ConfigurationRequest configurationRequest = new ConfigurationRequest();
        configurationRequest.addField(new TextField("driver", "Driver to use", "", "D
        configurationRequest.addField(new TextField("ip", "Ip and port", "", "Ip and
        configurationRequest.addField(new TextField("url", "JDBC URL", "", "Fully qua
        configurationRequest.addField(new TextField("dbName", "Database name", "", "D
        configurationRequest.addField(new TextField("username", "Username", "", "User
        configurationRequest.addField(new TextField("password", "Password", "", "Pass
        return configurationRequest;
    }
}
```

Figur 4.4: *Config klassen*

4.3.3 Testning/Produktverifiering

Produktkontroll är en viktig del av produktutveckling. För detta plugin till Graylog testades alla analysklasserna enskilt. Det betyder att alla klasser får en egen testklass för att kunna göra utförliga JUnit-tester. Dessa tester ska följa “The Builder Pattern” [9] för en strukturerad uppbyggnad av testerna som producerar enkel och lättläst kod. Referensen beskriver användingen i ett C# program men principen följer densamma för Java, vilket är att förkorta koden i själva testerna.

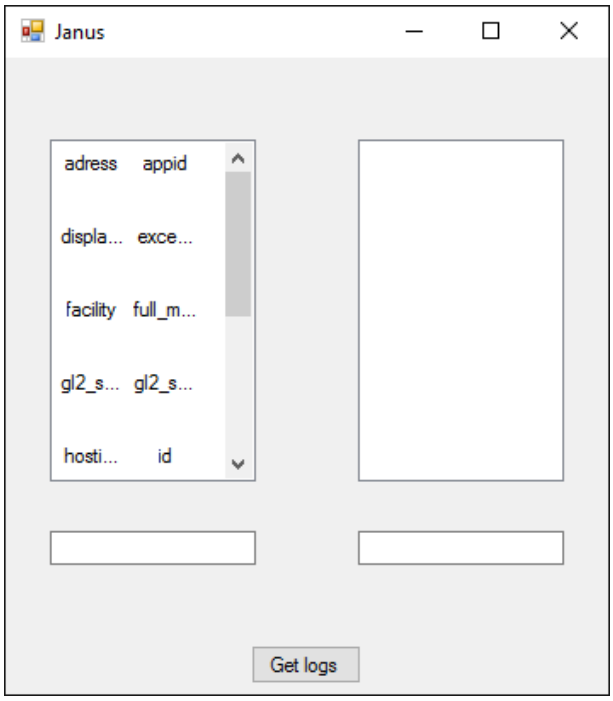
4.4 Utveckling av externt program

Det externa programmet har utvecklats i två olika faser. Meningen med det första programmet (som förklaras mer utförligt i stycke 4.4.1), GraylogFilter, var att programmet skulle hämta loggar från Graylog genom att använda Graylog:s välutvecklade REST API. På detta sätt kunde data hämtas som JSON-objekt vars innehåll var den information man sökte efter i varje logg. Det svåra var nu hur datan skulle behandlas. All data i loggarna var inte relevant ur Anturas perspektiv och det fanns inget sätt att behandla detta på ett smart sätt. Antingen skulle datan behandlas av pluginet och lagras i databasen, eller så var det externa programmets uppgift att skicka förfrågningar till Graylog via dess REST API. De loggar som returnerades skulle därefter behandlas efter vilken typ av information som var relevant. Detta för att ta fram vilket fel som är mest kritiskt och således även mest intressant för Antura att lösa. Detta var exakt vad ett plugin kunde göra utan att

behöva hämta hem loggar från Graylog servern. Det var lämpligare skriva ett program som skulle skicka förfråggningar till databasen och därefter visualisera den resulterade datan med ett grafiskt användargränssnitt.

4.4.1 GraylogFilter

GraylogFilter tillhörde en av de första idéer om hur själva uppgiften som Antura tillhandahållit kunde lösas på. Denna lösning kom dock endast till utvecklingsstadiet då det var ett opraktiskt sätt att lösa problemet på. Behandlingen av data skulle enbart ske av det externa programmet och således skulle detta program snabbt bli väldigt omfattande. Så rent grafiskt hann det inte bli speciellt avancerat utan det är ett väldigt enkelt användargränssnitt. Tanken var att användaren skulle ange vilka fält, listan till vänster i Figur 4.5, som var av intresse. Dessa fält lades sedan till av logiken i programmet till en URL som skickade en förfrågan till Graylog om loggar med givna fält. Listan till höger i Figur 4.5 ska visa valda fält och de två fälten nedanför listorna är textboxar vars syfte är att ange ett tidsintervall för de loggar som skulle hämtas.



Figur 4.5: *Config klassen*

Tanken med filtrering var att det skulle finnas logik i programmet som sparade undan vissa delar av varje meddelande från alla loggar, utförde en algoritm på detta och jämförde sedan datan med den befintliga lagrade datan i databasen. Detta sätt gjorde det möjligt att dela upp själva logiken med det grafiska på ett enkelt sätt om programmet behövde uppdateras då till exempel Antura valt att lägga till fler fält. Det grafiska kunde åsidosättas då fokus skulle läggas på logiken; att ta fram en effektiv algoritm som underlättar för Anturas felsökningsprocess. Denna lösning slutade utvecklas då pluginet började att fungera såsom var tanken.

4.4.2 Titan

Den slutgiltiga lösningen var att utveckla Titan, ett program som visar datan från databasen. Titan är, med sitt användarvänliga grafiska gränssnitt, en lösning där användaren kan hämta relevanta loggar beroende på deras sökning. Det går även att visualiseras loggarnas stacktrace i en trädvy. De loggar eller den stacktrace som hämtas visualiseras sedan i ett rutnät till höger om filtermenyn. Figur 4.6 visar hur varje sökning kopplas ihop med en klass av typen ILogsFilter. ILogsFilter är ett gränssnitt som innehåller metoder för att hämta relevant data för det unika filter som användaren har valt att applicera på all data i databasen. ILogsFilter kommer att förklaras mer utförligt senare. Som det visas i Figur 4.6 finns det fyra olika typer av filter. Detta ger i sin tur fyra olika listor med sökalternativ där varje sträng representerar en nyckel till respektive Dictionary. Det FilterMapping gör är att läsa in varje filter som blir skapat och göra det möjligt för en användare av Titan att ta del av Titans dynamiska filterlösning. Det är enkelt att lägga till ett nytt filter. Det som behövs är att skapa en ny klass som ärver ILogsFilter och lägga till filtret i en relevant Dictionary med filtrets namn som nyckel. Filtret kommer sedan att finnas med över tillgängliga filter när Titan startar om. Tanken med detta är att Antura enkelt ska kunna gå in och lägga till filter efter behov.

Till varje filter skickas det minst med tre parametrar:

- En sträng som innehåller IP-adress till servern, användarnamn, lösenord och namn på databasen.
- Ett tidsintervall från det datum loggar ska hämtas.
- Ett tidsintervall till det datum loggar ska hämtas.

Tidsintervallen är till för att minska belastningen på sökningen då databasen kommer att innehålla flera tusen loggar efter ett tag och för att loggar efter ett visst datum kan vara irrelevanta för Antura. Detta då dessa fel kanske redan har blivit lösta och inte är aktuella längre för Anturas räkning. I vissa filter förekommer det en extra sträng som parameter vars syfte är att möjliggöra en sökning på ett specifikt företag.

4.4.2.1 ILogsFilter

ILogsFilter är det gränssnitt vars uppgift är att göra det enkelt att lägga till fler, mer specifika och mer relevanta filter för hur all data i databasen ska presenteras. Detta interface visas i Figur 4.7 och innehåller två metoder:

- DataTable updateDataTable(): Skickar en förfrågan till databasen och fyller ett DataTable med den efterfrågade informationen.
- ILogsFilter getNewILogsFilter(string company): Returnerar en klass som implementerar ILogsFilter. Strängen company representerar namnet på det företag som klassen är ämnad för.

Fördelen med att skapa ett interface är att programmet blir dynamiskt. För att lägga till ett filter, är det endast att skapa en ny klass som implementerar ILogsFilter och ange för varje metod vad som ska hämtas samt vad som ska returneras. Filtret kopplas sedan ihop med sin nyckel som är en sträng vars namn fungerar som söknamn i en box med olika filter, se Figur 4.8.

```
public class FilterMapping
{
    public  Dictionary<string, ILogsFilter> MostImportantFiltersMappings { get; private set; }
    public  Dictionary<string, ILogsFilter> VIPCompaniesFiltersMappings { get; private set; }
    public  Dictionary<string, ILogsFilter> OrdinaryCompaniesFiltersMappings { get; private set; }
    public  Dictionary<string, ILogsFilter> OthersFiltersMappings { get; private set; }

    private readonly string connString;

    private DatabaseConnector DatabaseConnector;
    private List<Tuple<string, bool>> companies;

    private DateTime dateTimeFrom;
    private DateTime dateTimeTo;

    public FilterMapping(string connString)
    {
        this.connString = connString;
        DatabaseConnector = new DatabaseConnector(connString);

        MostImportantFiltersMappings = new Dictionary<string, ILogsFilter>();
        VIPCompaniesFiltersMappings = new Dictionary<string, ILogsFilter>();
        OrdinaryCompaniesFiltersMappings = new Dictionary<string, ILogsFilter>();
        OthersFiltersMappings = new Dictionary<string, ILogsFilter>();
    }
```

Figur 4.6: *Titans filter-klass*

```
public interface ILogsFilter
{
    /// <summary>
    /// Sends a query to the database and fills a DataTable with the requested information.
    /// </summary>
    /// <returns></returns>A DataTable containing the information from the query.
    DataTable updateDataTable();
    /// <summary>
    /// Changes to a new relevant ILogsFilter.
    /// </summary>
    /// <param name="company"></param>The name of the company.
    /// <returns></returns>A new class that implements ILogsFilter or null if no such options is needed.
    ILogsFilter getNewILogsFilter(string company);
}
```

Figur 4.7: *ILogsFilter*

4.4.2.2 TreeView med Stacktrace

En stacktrace kan delas upp i tre olika delar: ett paket, en klass och en metod. Detta visualiseras enkelt med en TreeView som innehåller noder. Varje nod representerar antingen ett paket, en klass eller en metod för att ge en bättre överblick på hur ofta varje stacktrace har förekommit. Gränssnittet för noderna döptes till INode och innehåller de metoder som visas i Figur 4.9.

- INode add(INode node): Lägger till en INode i en Dictionary där namnet på node läggs till som nyckel och värdet är node. Om namnet redan har förekommit returnerar metoden samma nod som skickas som parameter.
- string getName(): Returnerar namnet på den efterfrågade noden.
- System.Windows.Forms.TreeNode getTreeNodes(): Bygger upp ett träd med noder och returnerar en TreeNode som representerar roten av trädet.

```
private void updateMostImportantFiltersMappings()
{
    MostImportantFiltersMappings.Add("Most fatal errors", new MostFatalErrorsLogsFilter(connString, dateTimeFrom, dateTimeTo));
    MostImportantFiltersMappings.Add("Most frequent stacktrace", new MostFrequentStacktraceLogsFilter(connString, dateTimeFrom, dateTimeTo));
    MostImportantFiltersMappings.Add("Company and occurrences (All)", new CompanyAndOccurrencesMessageLogsFilter(connString, dateTimeFrom, dateTimeTo));
    MostImportantFiltersMappings.Add("Company and occurrences (VIP)", new VIPCompanyAndOccurrencesMessageLogsFilter(connString, dateTimeFrom, dateTimeTo));
    MostImportantFiltersMappings.Add("VIP logs", new VIPLogsFilter(connString, dateTimeFrom, dateTimeTo));
    MostImportantFiltersMappings.Add("All Logs", new AllLogsFilter(connString, dateTimeFrom, dateTimeTo));
}
```

Figur 4.8: *Ett söknamn kopplas ihop med ett passande filter*

```
public interface INode
{
    INode add(INode node);
    string getName();
    /// <summary>
    /// Return a TreeNode containting all TreeNodes under itself.
    /// </summary>
    /// <returns>A TreeNode./returns>
    System.Windows.Forms.TreeNode getTreeNodes();
    int getNrErrors();
}
```

Figur 4.9: *INode*

4.5 Databasen

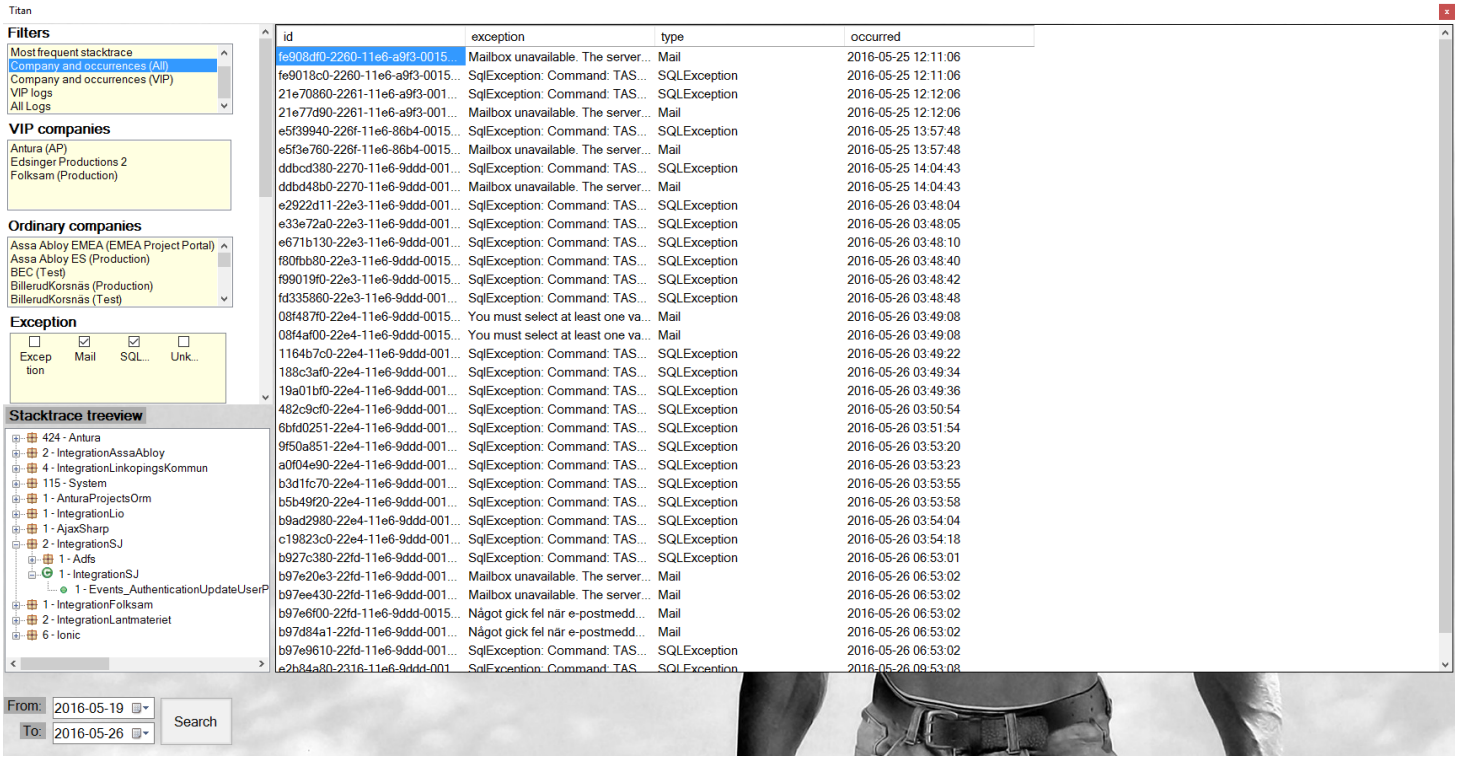
Databasens uppbyggnad avgör hur både det externa programmet Titan och själva pluginet ska arbeta. Detta leder till att dess struktur har stor påverkan under utvecklandet av de båda delarna. Därför måste den vara designad för att enkelt kunna skicka in data och hämta data. På grund av den potentiellt enorma datamängd som kan skickas in till Graylog, måste databasens design undvika överflöde och upprepningar. De områden som kommer att drabbas mest kommer att vara hur en stacktrace kommer att lagras. Hela strängen behövs, men även enskilt i vilket packet, klass och metod som den skedde i för att kunna skapa översikten som beskrivs i avsnitt 4.2.2. Den uppdelningen tar pluginet hand om, fast alla delar kommer att förekomma flertal gånger, speciellt paket och klass. Dessa måste sparas undan varje gång en nytt, unikt packet, paket eller metod kommer och återanvändas vid upprepningar.

5
Resultat

I detta avsnitt kommer slutprodukten av projektet att presenteras och hur pluginet och det externa programmet samarbetar.

5.1 Kategorisering

Det externa programmet Titan, som beskrivs i avsnitt 4.4.2, har förmågan att visa alla loggar som har skickats in till Graylog enskilt, logg för logg, såväl efter de grupperingar som pluginet hanterar och kategoriserar efter loggarnas innehåll. Kategorisering av relativt simpel information, exempelvis vilket företag som har fått felet, görs utan något extra arbete från pluginet. Namnen kopplas endast till loggens id och det är allt arbete som pluginet gör. En lite mer komplicerad kategorisering som kräver mer analys, är ExceptionMessage som letar efter nyckelord i meddelandet. När den hittar en match så kommer loggens exception att klassas som den sorten och går att söka efter i Titan. Resultatet och hur varje kategorisering kan väljas visas i Figur 5.1.



Figur 5.1: *Titan grafiskt*

5.2 Plugin till externt program

Prestandamässigt var plugin det bättre valet kontra det externa programmet som skulle läsa alla meddelanden varje gång med Graylogs REST API, avsnitt 4.1. Det är även mer pålitligt eftersom att ett plugin alltid arbetar så länge Graylog-servern är igång. På det sättet kommer inte några loggar saknas. Det var även bättre designmässigt, eftersom uppdelningen av arbetet programmen skulle utföra var rimligare än att låta ett program hantera allt. I slutändan ansågs det att ett plugin var det mer passande valet på grund av dess större möjligheter och större kontroll av varje felmeddelande som kommer in till Graylog-servern.
Samarbetet mellan det externa programmet och pluginet blir enhetlig med hjälp av databasens struktur och är en centralisering för de båda delarna, en struktur de måste följa. Detta medför att det som pluginet skickar in till databasen, kan det externa programmet utnyttja till fullo.

5.3 Produktverifiering

För att verifiera att en slutprodukt fungerar som den ska krävs det tester. Till Titan har Unit testing tillämpats och nästintill alla klasser har en testklass där varje metod testas. I .NET brukar man använda sig av de tre A:na för att bygga upp testmetoder och det har även tillämpats här. Dessa står för:

- Arrange: alla nödvändiga förberedelser görs.
- Act: agera på objektet eller metoden under test.
- Assert: verifiera att det förväntade resultatet överensstämmer med det faktiska resultatet.

Titan har runt 70 olika testmetoder och alla dessa tester blev godkända. Programmets funktionalitet är således verifierat.

6
Diskussion

Som analys av projektet, kommer diskussionen att involvera en mer personlig syn på vad som fungerade bra och dålig under arbetet, uppdelningen av arbetet och komplikationer som uppkommit. Det kommer även att tas upp en mer kritisk reflektion över arbetet om vad och hur det hade kunnat förbättrats.

6.1 Baskunskaper och svårigheter

I början av arbetet hamnade vi i en brant uppförsbacke. Vi utsattes för ett helt nytt sätt att arbeta på. Vi besatt inga tidigare kunskaper om Graylog och hur den fungerar. Ännu krångligare blev det när vi skulle bygga ett plugin med hjälp av Maven som krävs för att skapa ett Graylog-plugin [2]. Det kräver en speciellt uppbyggnad av projekten i Eclipse och extra externa filer för kompileringen. I längden hjälpte det oss att enkelt kompilera projektet och ge oss rätt bibliotek och jar-filer, men gjorde starten av arbetet mycket långsammare. Vi hade dock stor hjälp av vår handledare på Antura för att få tillgång till alla nödvändiga verktyg för att kunna starta arbetet.

När vi hade tagit oss förbi de flesta hinder och kunde skapa vår första version av pluginet, kom problemet med testning. Vi saknade båda kunskap för att testa produkter som måste distribueras till en avlägsen dator. Där finns inga möjligheter att debugga eller köra vanliga tester som försvårade utvecklingen. Andra metoder för att testa programmet kom fram senare under utvecklingen och berodde som sagt på ingen tidigare erfarenhet av denna sortens arbetssätt.

6.2 Arbetsuppdelning

Arbetsuppgifterna kunde delas upp på ett smidigt och effektivt sätt. Den ena jobbade med pluginet och den andra med det externa programmet. Eftersom båda delarna måste arbeta efter samma databasstruktur kunde man ignorera den andres arbete helt. Gjordes det en ändring i databasen, justerades båda delarna respektive efter de modifieringar som skett.

6.3 Kritik mot arbetet

Överlag blev vi båda väldigt nöjda med slutprodukten. Det finns många sätt att sortera alla meddelanden på och stöd för att lägga till många fler. En nackdel är att de olika söksätten inte samarbetar på någon nivå, t.ex. kan man inte visa alla med ett sorts fel som ett specifikt företag har fått m.m. Det finns lösningar till detta, men inga som är

bra kodmässigt med den befintliga strukturen. En annan del av projektet som kunde ha förbättrats var vårt arbetssätt. Vi gjorde många beslut i början och trodde att det var det slutgiltigt, men tyvärr ändrades databasens struktur gång på gång för att möta våra nya idéers krav. Detta ledde till att både Titan och pluginet behövde justeras och krävde mer tid vid varje ändring. Vi gjorde om allt väldigt många gånger, men resultatet blev i slutändan väldigt bra.

6.4 Hållbarhetsaspekt

Då databasen är uppbyggd med entiteter och relationer minskar minnesanvändningen. Detta leder till en lägre strömförbrukning då det inte krävs samma serverkapacitet vid användning.

7
Slutsats

Som helhet har arbetet varit oerhört roligt och lärorikt att göra. Vi känner att våra kunskaper inom Java, .NET och C# har förbättrats, samtidigt som målet med själva arbetet har uppfyllts. Vi kan gladeligen lämna över en produkt till Antura som vi är stolta över och kommer att hjälpa dem i deras felhanteringsprocess.

Vi är nöjda med såväl pluginet som det externa programmet, då vi lyckades skapa det som var själva uppgiften. Vi lyckades även skapa ett program vars utvecklingsmöjligheter är stora, vilket Antura tyckte var viktigt. Hela projektets tre stora delar - Titan, databas och plugin - är lätta att bygga vidare på då vi har försökt att bygga upp programmen så dynamiskt som möjligt.

Något extra nämnvärt är att det har varit givande att komma i från Chalmers trygga skolmiljö och få till en viss del testa på det “riktiga” arbetslivet med fasta rutiner och att få vara en del av Antura.

Litteraturförteckning

[1] Graylog, *Overview* [Internet] [citerat 2016 Maj 27] Tillgängligt från: https://www.graylog.org/overview

[2] Graylog, *Plugins* [Internet] 2015[citerat 2016 Apr 22] Tillgängligt från: http://docs.graylog.org/en/2.0/pages/plugins.html

[3] MongoDB, *Introduction to MongoDB* [Internet] [citerat 2016 Apr 22] Tillgängligt från: https://docs.mongodb.org/manual/introduction/

[4] Maven, *Introduction to the Standard Directory Layout* [Internet] [citerat 2016 Apr 28] Tillgängligt från: https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html

[5] Graylog, *GELF* [Internet] [citerat 2016 Maj 29] Tillgängligt från: http://docs.graylog.org/en/2.0/pages/gelf.html

[6] Bitbucket [Internet] [citerat 2016 Maj 29] Tillgängligt från: https://bitbucket.org/

[7] NLog [Internet] [citerat 2016 Maj 29] Tillgängligt från: http://nlog-project.org/

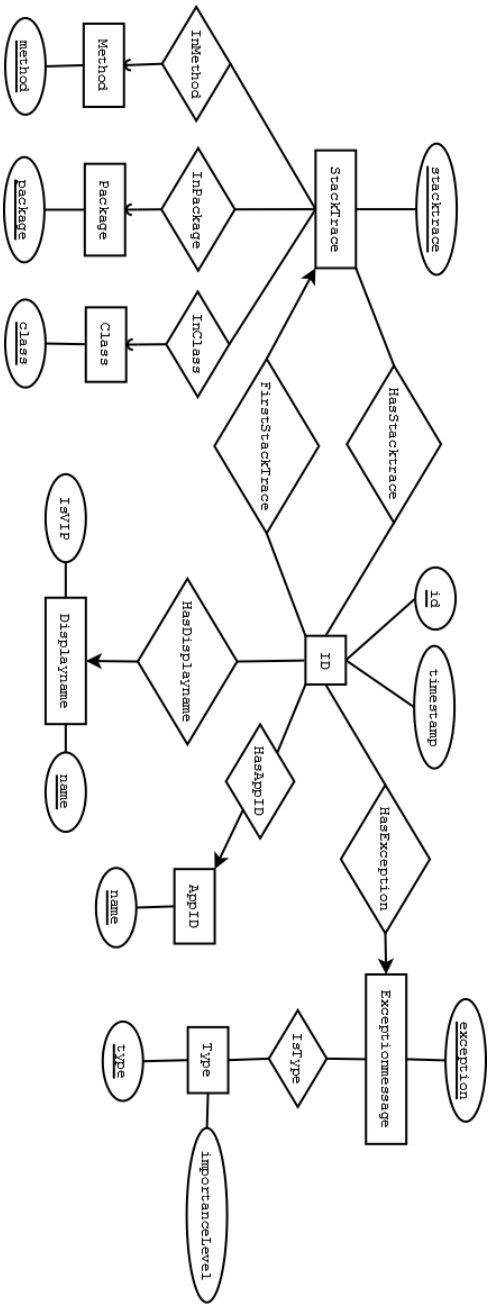
[8] NLog, *Log Levels* [Internet] [citerat 2016 Maj 17] Tillgängligt från: https://github.com/nlog/nlog/wiki/Log-levels

[9] Truyers, Kenneth. *Flexible and expressive unit tests with the builder pattern.* [Internet] 2013 Juli 15 [citerat 2016 Maj 3] Tillgängligt från: https://www.kenneth-truyers.net/2013/07/15/flexible-and-expressive-unit-tests-with-the-builder-pattern/

A

Database

A.1 Diagram



A.2 Schema

Schema

Entities

ID (id, timestamp)

StackTrace (stacktrace)

Displayname (name)

Exceptionmessage (exception)

AppID (name)

Package (package)

Class (class)

Method (method)

Type (type, importancelevel)

Relations

HasStacktrace (id, stacktrace)
stacktrace -> StackTrace.stacktrace
id -> ID.id

FirstStackTrace (id, stacktrace)
id -> ID.id
stacktrace -> StackTrace.stacktrace

HasDisplayname (id, displayname)
id -> ID.id
displayname -> Displayname.name

HasException (id, exception)
id -> ID.id
exception -> Exceptionmessage

IsType (exception, type)
exception -> Exceptionmessage.exception
type -> Type.type

HasAppID (id, appid)
id -> ID.id
appid -> AppID.name

InPackage (stacktrace, method)
stacktrace -> Stacktrace.stacktrace
method -> Method.method