Performance analysis in the DataDistributor system

Ulrich Bæch-Laursen

EVU Master Thesis, University of Aarhus, 2018-06-17

Confidential
1 Introduction

Service Level Agreement (SLA) requirements for large customer projects have long been introduced by the customer as a standard part of every tender that KMD submit a bid on.

For the DataDistributor project there were originally over 60 non-functional requirements related to service performance which makes quality attributes (QA) related to performance of paramount importance in all aspects of the system.

To achieve high performance in software systems the software needs to be tested after each iteration of the development process to create a performance baseline and deltas to measure any increase or decrease in performance.

This project is written to identify the mechanics of how to collect and analyze performance data across all end-user services and how each component can be tested in relation to software testing and component interaction to achieve the highest performance possible.
2 Abstract

This thesis will show how to implement logging SLA metrics from services and how to choose and implement an Application Performance Monitoring (APM) platform to monitor and improve performance of software components affecting SLA negatively.

It will show what performance metrics to log and how to utilize those metrics once gathered for SLA metrics conformity and performance measurements.

It will be shown how implementing SLA metric logging across all REST services in the DataDistributor system can be achieved as well as the methods and techniques used to do so.

Once a viable method of logging performance metric data is chosen it will be used to produce SLA compliance data showing whether services meet the DataDistributor SLA compliance or not.

To substantiate whether SLA is met or violated by individual software components performing less than optimally, their performance will be shown analyzed by using an applicable APM platform.

To choose an APM platform a list of viable APM platforms will be compiled. From that list one or more Application Performance Monitoring platforms will be chosen, implemented and their capabilities compared to a list of DataDistributor requirements. The Application Performance Monitoring platform will be used to show how it can be used to gather performance metrics from code running in the DataDistributor system.
3 Table of contents

1 Introduction 2
2 Abstract 3
3 Table of contents 4
4 Motivation 6
  4.1 SLA requirements .......................................................... 6
  4.2 Software performance metrics ........................................ 6
  4.3 Logging in the DataDistributor ....................................... 8
5 Terms and definitions 9
6 Problem statement 10
  6.1 Identifying, logging and analyzing SLA ......................... 10
  6.2 Analyzing software component performance .................. 11
7 Method 12
  7.1 Implementing SLA metrics ............................................. 12
  7.2 Implementing an APM tool ............................................ 12
8 Analyses and Results 13
9 Dividing SLA requirements in the solution into QAS 14
10 Test environments 17
  10.1 Local test environments ............................................. 17
  10.2 Cloud test environments ......................................... 18
11 Implementing SLA metrics 20
  11.1 Analyzing the DataDistributor architecture and source .... 20
  11.2 Extending the DataDistributor solution ......................... 22
  11.3 Implementing SLA metrics as middleware ..................... 24
  11.4 Pipeline ordering ..................................................... 25
  11.5 Logging SLA metrics ................................................... 26
  11.6 Ensuring SLA Metrics accuracy .................................... 35
  11.7 Building, deploying and testing architectural prototypes.... 36
  11.8 Running load tests with extended logging ..................... 37
  11.9 Collecting compliance data: cloud test environment ....... 41
  11.10 Compliance calculations using LogPoint ....................... 42
12 Implementing an APM tool 45
   12.1 APM tool requirements .............................................................. 45
   12.2 APM listing and selection .......................................................... 48
   12.3 DynaTrace APM Installation ..................................................... 50
   12.4 DynaTrace APM usage ............................................................... 51
   12.5 APM conclusions ......................................................................... 56

13 Conclusion 57

14 References 59

15 Appendix 61
   15.1 Appendix 1 -SLA requirements snippets ...................................... 61
   15.2 Appendix 2 – Datafordeler refactoring ....................................... 64
   15.3 Appendix 3 – Application PerformanceManagement Buyer’s Guide and Reviews ......................................................... 70
4 Motivation

4.1 SLA requirements

To meet the service performance requirements of the contract (appendix 15.1) for the DataDistributor platform that are under SLA, a set of components must be implemented to record and gather performance metrics for analysis.

The DataDistributor is built on a set of technologies residing on both Linux and Windows platforms of which only some are developed by KMD. Because of that, it cannot be asserted that all components can be extended in a way for all to use the same software library/component to create and log performance data. This fact leaves the project with the question of how to implement this across platforms and components.

SLA based performance analysis is binary in its interpretation of whether performance is compliant or not. It does not state why the performance is or is not compliant nor does it state the performance metrics of sub-components composing a given service.

4.2 Software performance metrics

It is interesting for KMD to know the performance metrics of individual components that contribute to the SLA being compliant or not – optimally this should be monitored from one iteration to the next to identify and improve non-performant components. This report will reflect the investigation into methods, techniques and software that can help gather performance metrics for services and their sub-components in the project and how to use those metrics to improve software performance.

There are several different service types in the DataDistributor project of which the main system-to-system services consist of the following types of services:

- Representational State Transfer (REST)
- Web Map Service (WMS)
- Web Tile Map Service (WMTS)
- Web Coverage Service WCS
- Tile Map Service (TMS)
- Web Feature Service (WFS)
In the view below the external services are shown along their associations to the internal technology layer. All UML in this thesis are created in Enterprise Architect [5] due to this tool being the default for KMD and the DataDistributor project. Models conform to the ArchiMate 2 template [6] and definitions.

Figure 1 - Technology, Services Layered View 1
4.3 Logging in the DataDistributor

The DataDistributor uses the LogPoint SIEM system to gather information from all servers and subcomponents in the system.

Usages of LogPoint in the DataDistributor system:

- **Data aggregation**: LogPoint aggregates data from all aspects of the project, including security and application logs from member servers, databases, and applications.

- **Correlation** of data: Data is normalized using a set of attributes linking events together across multiple servers from authentication, authorization and DataDistributor services.

- **Alerts**: LogPoint is setup to alert stakeholders in the project about problems in the system.

- **Retention**: All data is stored for a minimum of six months where after some data must be deleted and other data is moved to secondary storage.

For the DataDistributor system to become SLA compliant it is requested that the **Compliance** capability of the LogPoint system is utilized. Once SLA metrics described in 4.1 are logged this should be possible which this thesis will show.
## 5 Terms and definitions

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>APM</td>
<td>Application Performance Monitoring</td>
</tr>
<tr>
<td>GIS</td>
<td>Geographical Information System</td>
</tr>
<tr>
<td>LogPoint</td>
<td>SIEM Software Solution used by KMD to store log vast amounts of log data, including event data, application data for numerous usages.</td>
</tr>
<tr>
<td>OGC [3]</td>
<td>An international voluntary consensus standards organization where various organizations collaborate in implementation of open standards for geospatial content and services. Services standardized by OGC are often called OGC services.</td>
</tr>
<tr>
<td>SIEM</td>
<td>Security Information and Event Management software</td>
</tr>
<tr>
<td>SLA</td>
<td>Service Level Agreement: an official commitment that prevails between a service provider and a client. In this thesis particularly related to the aspect of the service quality: ‘performance’.</td>
</tr>
<tr>
<td>WMTS Service [3]</td>
<td>Web Map Tile Service: provides map image tiles / pre-rendered georeferenced map tiles</td>
</tr>
</tbody>
</table>
6 Problem statement

Service level agreements and software performance are tightly coupled in the DataDistributor system and as such this thesis will show how to implement logging SLA metrics from services and how to choose and implement an APM platform to monitor and improve performance of software components affecting SLA negatively.

6.1 Identifying, logging and analyzing SLA

To show that the DataDistributor system complies with SLA requirements, SLA metrics must be recorded for all services uniformly in the LogPoint SIEM system.

In appendix 15.1 a snippet of four individual webservice SLA requirements for the DataDistributor solution are stated, which are essential to the understanding of SLA requirements throughout this thesis. Some of these are described below.

The SLA requirements from appendix 15.1 are essentially the customer’s way of constructing rules to govern how much time can pass before data is returned for one of the four named SLA types.

**The Names, types and requirements are linked directly to these requirements:**

1. Performance requirement for a *simple* without authorization webservice request that requires no authorization or is called with pre-authorized credentials, defined by returning data within 60ms.
2. Performance requirement for a *simple* with authorization webservice request with authorization, defined by returning data within 90ms.
3. Performance requirement for a *normal* webservice request, defined by returning data within 1000ms.
4. Performance requirement for a *complex* webservice request, defined by returning data within 2500ms.

The names for the SLA categories were chosen by the customer and can seem quite arbitrary but the following rules apply to the types:

A webservice method is categorized as:

- ‘*simple*’ if the query for the method returns one record from one table in an indexed result-set using a primary key search variable.
• ‘normal’ if the query for the method returns < 100 objects from one table in an indexed result-set using a primary key search variable.

• ‘complex’ if the query for the method returns < 100 objects in an indexed result-set using a primary key search variable joining data together from two or more tables.

The services can, from the SLA requirements be categorized into the following types:

<table>
<thead>
<tr>
<th>Service type:</th>
<th>Simple</th>
<th>Normal</th>
<th>Complex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data type:</td>
<td>Structured</td>
<td>Unstructured</td>
<td></td>
</tr>
</tbody>
</table>

Queries that for one or more reasons cannot be placed in one of the categories listed above are automatically placed in the out-of-category type.

**Out-of-category example:** a query in an indexed result-set using a primary key search variable joining data together from two or more tables yielding > 100 objects

SLA requirements will be analyzed and divided into individual Quality Attribute Scenarios (QAS) and load testing performed on one or more services to show that SLA metrics are logged and can be used for calculating compliance data using the LogPoint SIEM compliance tools.

### 6.2 Analyzing software component performance

Software components affecting DataDistributor services performance, and thus SLA requirements in a negative way must be identified using an Application Monitoring Platform [7] (APM).

The platform must monitor individual software components and their contribution to the performance of a service request measured in milliseconds. The recorded data will enable the project to refactor and improve performance of components identified.

The APM system must show detailed component information to improve performance based on the source code.

The APM should be able to show live performance metrics of chosen components to quickly identify problems related to congestion and scalability while performing load tests as well as the ability to compare performance data for multiple versions of monitored software components.
7 Method

7.1 Implementing SLA metrics
The first main objective of implementing a component for recording SLA metrics across all services will be achieved by:

1. Dividing SLA requirements into Quality Attribute Scenarios.
2. Analyzing the DataDistributor architecture and source code
3. Identifying ways of extending the solution to log SLA metrics in the LogPoint SIEM system.
4. Applying TDD principles [4] for refactoring the code base for introducing SLA metrics with as few changes to the existing code as possible.
5. Creating architectural prototypes to:
   a. Show that the chosen implementation method works as expected
   b. Build knowledge about the components used by the DataDistributor platform
   c. Implement conformity between the chosen prototype architecture and existing components
   d. Ensure maintainability by choosing implementation methods that are the same or similar to the ones already used in the project.
6. Testing the prototypes in local and private cloud environments to show that collected data can be used for compliance calculations.
7. Running load tests for selected services using the extended logging.
8. Showing actual compliance calculation results from the LogPoint SIEM software.

7.2 Implementing an APM tool
The second objective involving implementing an APM system into the DataDistributor system will be achieved by:

1. Defining APM tool requirements
2. Gathering information and knowledge about different APM’s their characteristics, the functionality they offer, and which the selected platforms that suit the DataDistributor solution best.
3. Selecting one APM to work with during this thesis
4. Implementing the APM in local and/or private cloud testing environments
5. Using the APM to identify components that have performance problems.
6. Concluding and wrapping up of the selection process and experience working with a chosen APM.
8 Analyses and Results

This thesis will show how complex systems like the DataDistributor can implement a common SLA logging mechanism across multiple service types to calculate compliance data using the LogPoint SIEM system.

SLA data will be logged while performing load testing on selected services.

The thesis will also show the benefits and drawbacks of using a third party APM system to identify software component improvements for the DataDistributor. This is done for the system to show that it is SLA compliant and to show how live application monitoring may help in identifying performance bottlenecks as they occur in the system.

The results are shown in sections 8 – 11.
9 Dividing SLA requirements in the solution into QAS

To set the stage for describing SLA in the DataDistributor system this thesis will use Quality Attribute Scenarios to precisely define the Quality Attribute in question, the properties of the QA and any relevant questions or issues related to it.

For the first three types of services stated in section 6.1:

1. simple without authorization
2. simple with authorization
3. normal

there are two measures of compliance. One for structured data and one for unstructured data. For the fourth type: complex services, only structured data is covered by the SLA. The structured/unstructured data types however, do not affect the response time requirements of the services and will be covered in section 0 where the gathered compliance data will be investigated.

For a variety of services, the three performance requirements simple, normal or complex are not applicable due to several reasons:

- Data is not indexed and cannot be fetched from the database in deterministic time.
- Dataset returned is too large to be returned within even the required time of a complex service type.
- Data must be created upon request – this is the case for several OGC services where custom image tiles or entire pyramids of image layers must be created on first request to a specific map.

These services are categorized as ‘out of category’ and no SLA requirements is required for them. This thesis will not cover ‘out of category’ services in depth.
QAS for the chosen requirements are stated in the next section and created according to Barbacci [1].

The first scenario is for a simple without authorization web service request type with structured or unstructured data returned without authorization needed or with pre-authorized credentials.

<table>
<thead>
<tr>
<th>Scenario(s):</th>
<th>Webservice request – Simple with pre-authorization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relevant Quality Attributes:</td>
<td>Performance</td>
</tr>
<tr>
<td>Source:</td>
<td>Web service client</td>
</tr>
<tr>
<td>Stimulus:</td>
<td>Requests data from simple webservice with authorization</td>
</tr>
<tr>
<td>Artifact</td>
<td>Structured or unstructured data</td>
</tr>
<tr>
<td>Environment:</td>
<td>Normal operation</td>
</tr>
<tr>
<td>Response:</td>
<td>Duration from request sent to response returned measured in milliseconds</td>
</tr>
<tr>
<td>Response Measure:</td>
<td>For the 95th percentile of requests, data is returned within 60ms for structured data. For the 90th percentile of requests, data is returned within 60ms for unstructured data.</td>
</tr>
<tr>
<td>Questions:</td>
<td>None identified</td>
</tr>
<tr>
<td>Issues:</td>
<td>None identified</td>
</tr>
</tbody>
</table>

Table 1 - QAS 1: Web service performance: simple, pre-authorized

The second scenario is for a simple web service request type with structured or unstructured data returned including authorization.

<table>
<thead>
<tr>
<th>Scenario(s):</th>
<th>Webservice request – Simple with authorization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relevant Quality Attributes:</td>
<td>Performance</td>
</tr>
<tr>
<td>Source:</td>
<td>Web service client</td>
</tr>
<tr>
<td>Stimulus:</td>
<td>Requests data from simple webservice with authorization</td>
</tr>
<tr>
<td>Artifact</td>
<td>Structured or unstructured data</td>
</tr>
<tr>
<td>Environment:</td>
<td>Normal operation</td>
</tr>
<tr>
<td>Response:</td>
<td>Duration from request sent to response returned measured in milliseconds</td>
</tr>
<tr>
<td>Response Measure:</td>
<td>For the 95th percentile of requests, data is returned within 90ms for structured data. For the 90th percentile of requests, data is returned within 90ms for unstructured data.</td>
</tr>
<tr>
<td>Questions:</td>
<td>None identified</td>
</tr>
<tr>
<td>Issues:</td>
<td>None identified</td>
</tr>
</tbody>
</table>

Table 2 - QAS 2: Web service performance: simple, with authorization
The third scenario is for a *normal* web service request type with structured or unstructured data returned.

<table>
<thead>
<tr>
<th>Scenario(s):</th>
<th>Webservice request – Normal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relevant Quality Attributes:</td>
<td>Performance</td>
</tr>
<tr>
<td>Source:</td>
<td>Web service client</td>
</tr>
<tr>
<td>Stimulus:</td>
<td>Requests data from simple webservice with authorization</td>
</tr>
<tr>
<td>Artifact</td>
<td>Structured or unstructured data</td>
</tr>
<tr>
<td>Environment:</td>
<td>Normal operation</td>
</tr>
<tr>
<td>Response:</td>
<td>Duration from request sent to response returned measured in milliseconds</td>
</tr>
<tr>
<td>Response Measure:</td>
<td>For the 90th percentile of requests, data is returned within 1000ms for <strong>structured</strong> data. For the 90th percentile of requests, data is returned within 1000ms for <strong>unstructured</strong> data.</td>
</tr>
</tbody>
</table>

Questions: None identified

Issues: None identified

Table 3 - QAS 3: Web service performance: normal

The last scenario is for a *complex* web service request type with structured or unstructured data returned.

<table>
<thead>
<tr>
<th>Scenario(s):</th>
<th>Webservice request – Complex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relevant Quality Attributes:</td>
<td>Performance</td>
</tr>
<tr>
<td>Source:</td>
<td>Web service client</td>
</tr>
<tr>
<td>Stimulus:</td>
<td>Requests data from complex webservice with authorization</td>
</tr>
<tr>
<td>Artifact</td>
<td>Structured or unstructured data</td>
</tr>
<tr>
<td>Environment:</td>
<td>Normal operation</td>
</tr>
<tr>
<td>Response:</td>
<td>Duration from request sent to response returned measured in milliseconds</td>
</tr>
<tr>
<td>Response Measure:</td>
<td>For the 90th percentile of requests, data is returned within 2500ms for <strong>structured</strong> data.</td>
</tr>
</tbody>
</table>

Questions: None identified

Issues: None identified

Table 4 - QAS 4: Web service performance: complex
Test environments

To test the developed middleware, test applications and APM platforms - both local and cloud testing environments are used.

The SLA metrics middleware itself can be tested on a local version of the DataDistributor software running as a website using local databases for service configuration and service data. However, to get ‘real’ performance data the environment where the performance logs are collected must offer load balancing, authentication, authorization as well as representative datasets for the services tested. This task is overwhelming for a local test environment and thus the test environments located in KMD’s private cloud will be used for performance tests and log collection.

The software developed will be deployed to the webservers in one or more environments from the list: DEV01, TEST03 and TEST05

10.1 Local test environments

For the local test environment, laptop 1 with the following specifications was used:

<table>
<thead>
<tr>
<th>Server name:</th>
<th>H8750</th>
<th>Role:</th>
<th>Active Directory Domain controller</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lenovo P50</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32GB RAM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Quad Core @ 2.7GHz</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Software:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Windows 10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IIS 8.5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Visual Studio</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PostgreSQL server</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQL Server Express</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DataDistributor components:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- SwitchBoard</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>- REST Services</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Description:
Used for local testing of the switchboard component and the SlæMeasurementElement middleware developed.
The databases are a local copy of the test configuration or REST services and various test register data.
10.2 Cloud test environments

The cloud testing environment is created in an on-premise cloud solution built upon Azure-Stack offering a variety services such as infrastructure as a service (IaaS) know from the Internet version of Azure. IaaS is the main service used for the DataDistributor solution to rollout member servers on-demand. This thesis will not engulf in explaining the Azure-Stack setup of the private KMD cloud.

The private cloud test environments ‘DEV01’, ‘TEST03’ and ‘TEST05’ are comprised of the following servers located within the same VLAN. All environments are identical – with only server names varying: DV1->DT3->DT5 because of this only the TEST05 environment is described in this section.

<table>
<thead>
<tr>
<th>Server name: DT5ADCFSAW40</th>
<th>Role: Active Directory Domain controller</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Hardware:</strong></td>
<td><strong>Software:</strong></td>
</tr>
<tr>
<td>- 3.5GB RAM</td>
<td>- Active Directory</td>
</tr>
<tr>
<td>- Dual Core</td>
<td></td>
</tr>
<tr>
<td><strong>Description:</strong></td>
<td></td>
</tr>
<tr>
<td>Used for user and service management. To access a given REST service the user is given a service privilege role that resides in the AD. After logging on to the system using either a username/password or certificate-based authentication mechanism, the user is authorized to access the REST service if the user has been assigned the correct service privilege role associated to the service.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Server name: DT5ADFW4001</th>
<th>Role: Active Directory Federation services</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Hardware:</strong></td>
<td><strong>Software:</strong></td>
</tr>
<tr>
<td>- 3,5GB RAM</td>
<td>- Active Directory Federation Services (ADFS)</td>
</tr>
<tr>
<td>- Dual Core</td>
<td></td>
</tr>
<tr>
<td><strong>Description:</strong></td>
<td></td>
</tr>
<tr>
<td>Used for handling authentication and authorization throughout the DataDistributor system. Mechanisms for user authentication link to a user located in the Active Directory database and include:</td>
<td></td>
</tr>
<tr>
<td>- Username/password authentication</td>
<td></td>
</tr>
<tr>
<td>- Direct 2-way SSL Client authentication using FOCES[10] certificates</td>
<td></td>
</tr>
<tr>
<td>- Anonymous access</td>
<td></td>
</tr>
<tr>
<td>Server name: DT5WEBFSAW40</td>
<td>Role: Web server</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>Hardware:</td>
<td>Role:</td>
</tr>
<tr>
<td>- 3.5GB RAM</td>
<td>Windows Server 2012 R2</td>
</tr>
<tr>
<td>- Dual Core</td>
<td>IIS 8.5</td>
</tr>
<tr>
<td></td>
<td>APM Website</td>
</tr>
<tr>
<td></td>
<td>APM Client</td>
</tr>
<tr>
<td></td>
<td>DataDistributor components:</td>
</tr>
<tr>
<td></td>
<td>• REST Services</td>
</tr>
<tr>
<td></td>
<td>(SwitchBoard)</td>
</tr>
</tbody>
</table>

**Description:**
Used for hosting all IIS Websites including
- REST services (Switchboard)

<table>
<thead>
<tr>
<th>Server name: DT5DBSPGMU40</th>
<th>Role: Database server</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware:</td>
<td>Role:</td>
</tr>
<tr>
<td>- Standard A4 series VM [8]</td>
<td>Database server</td>
</tr>
<tr>
<td>- 14GB RAM</td>
<td>Ubuntu 14.04, Linux</td>
</tr>
<tr>
<td>- Octo Core</td>
<td>PostgreSQL server</td>
</tr>
<tr>
<td></td>
<td>DataDistributor components</td>
</tr>
<tr>
<td></td>
<td>• Service data</td>
</tr>
<tr>
<td></td>
<td>• Service configurations</td>
</tr>
</tbody>
</table>

**Description:**
Used to store register test data and REST service configurations.
11 Implementing SLA metrics

In this section the inner workings of the DataDistributor code will be analyzed to identify how and where to implement SLA metrics.

11.1 Analyzing the DataDistributor architecture and source

The external infrastructure services illustrated in Figure 1 are all accessed through a DataDistributor software component named ‘Switchboard’. The Switchboard handles a variety of functions including authentication, authorization and invoice/security audit logging i.e. who has accessed a service and when to perform security audits and to bill users for accessing payed services.

The switchboard component processes HTTP requests for all service URLs whether they are processed directly by the switchboard as is the case for REST services or whether the service resides in another context.

The Switchboard is a C# ASP.NET solution wrapped in an OWIN [14] container.

OWIN stands for Open Web Interface for .Net. and is a community-owned specification. OWIN defines a specification for de-coupling applications from the webserver using a delegate structure.

References to the OWIN interface can be found in Lattanzi [12].

![Figure 2 - OWIN Architecture according to Lattanzi [12]](image-url)
**OWIN middleware**

The Switchboard implements a set of OWIN middleware components that each implement OWIN, heightening separation of concerns for each component and making each of them reusable in other projects. The components are unaware of each other’s existence and do not know whether they communicate directly with the webserver or the next OWIN component in the pipeline.

The OWIN host process is run in four components:

```
Host → Server → Middleware → Application
```

The DataDistributor solution implements both Middleware and Application components.

To use OWIN, the DataDistributor solution uses Katana [13] - the Microsoft implementation of the OWIN specification running the Server, Middleware and Application layers in the host process, which in this thesis is the Internet Information Services web server on Windows server 2012.

For the DataDistributor solution a simplified OWIN server, middleware and application process view could be depicted in the following

![Diagram](image-url)

*Figure 3 - Switchboard OWIN process view*
11.2 Extending the DataDistributor solution

OWIN middleware is already being used for other types of logging in the DataDistributor solution and its ability to process the request both before and after application processing this implementation type can be used for logging SLA metrics for services.

In the next sequence diagram, the first proposal created is to have the SLA Metrics logging middleware handle the incoming request and start a timer before the REST service executor handles the request. When finished the REST executor returns the response to the Katana component and notifies the SLA logging middleware that it has finished execution. Note that the response is completed before SLA metrics are logged ensuring that every millisecond is accounted for.

![SLA Metrics sequence diagram](image)

Figure 4 - SLA Metrics sequence diagram

In the DataDistributor OWIN pipeline there are already several configured elements in the StartupInstaller class which is responsible for registering types in the Unity container via interfaces and their concrete implementation classes.
To ‘inject’ a new pipeline element containing SLA metrics logging, a new element ‘SlaMeasurementElement’ will have to be registered with a new interface and concrete implementation of the OWIN middleware class. The lifetime manager implementation for this should be of type \texttt{TransientLifetimeManager} which will ensure that the instance of the object is a new one for each incoming request.

\textbf{Alternatives to the suggested approach:}

1. Refactor the DataDistributor REST executor to implement a similar request timer.
2. Refactor all concrete service types to implement a similar request timer.

For the alternatives stated above the ‘timing’ aspect could be removed and simply log the time of the incoming request and the time when the response has been sent. This however requires additional correlation of request/response timings as well as computation of SLA metrics being done by new functionality, something which this thesis will not consider as the LogPoint SIEM system checking the SLA compliance has no such functionality and only looks at data already available in the log database.

The first proposal is by far the least intrusive and can be implemented without modification to the original solution by copying the \texttt{StartupInstaller} class to a new refactored version: \texttt{SlaStartupInstaller} which injects the new middleware element, becoming a new prototype of the SwitchBoard solution.
11.3 Implementing SLA metrics as middleware

To implement the SLA metrics the approach chosen is Test Driven Development (TDD) as described by [Christensen, 4] to create one or more functional architectural prototypes [Bardram, 18] to be the decision base for implementing SLA logging middleware in the DataDistributor system. This next section describes the efforts in this pursuit.

Unit testing

The DataDistributor solution uses Moq [15] for unit testing.

Moq is a third-party library that enables the developers in the DataDistributor solution to create dummy classes and dummy implementation of their methods without creating a class with actual method implementation by providing the interface name.

It is used to test functionality of classes and methods without implementing the classes that the tested class is dependent on.

The tests in this thesis will use Moq for the DataDistributor developers team to easily absorb the created tests into the main solution if qualified for usage once completed.

Appendix 15.2 shows verbose information about the tests created to refactor the DataDistributor code to implement SLA metrics.

The initial test list comprising 4 elements will be approached by creating a test for step 1 and gradually, using small steps moving toward the remainder of steps 2 through 4, while assuring that refactoring is done in each iteration as well as running all tests.

1. Test that middleware exists and can be called
2. Test that the middleware can log ‘something’
3. Test that middleware can log SLA service type
4. Test that middleware can log SLA metrics

This thesis will not cover all minor tests but only show the ones contributing most to the progress of the testing process.
11.4 Pipeline ordering

To be sure that the SLA metrics are calculated at the right time, it’s important to inject the SlaMeasurementElement in the pipeline as soon as possible. If done too late processing time from previous middleware components could affect the SLA metrics by yielding fewer milliseconds than factual processing time resulting in an inaccurate measurement.

The NetworkLatencyElement however MUST return as the first Element in the pipeline because it’s used to measure network latency and must thus not be slowed by other middleware elements.

![Figure 6 - The NetworkLatencyElement class](image)

The fact that the NetworkLatencyElement is added to the pipeline before the SlaMeasurementElement can be accepted as it has no overhead for REST service calls other than checking whether a special request header is present and if not, moving on to the next middleware element delaying computation of the SLA metrics by what should be only fractions of a millisecond. Testcases:

- Test_20_execute_CallRequest_LogPerformanceWithoutNetworkLatency()
- Test_21_execute_CallRequest_LogPerformanceWithNetworkLatency()

where a ‘dummy’ middleware class (making the middleware class sleep for 10ms) was inserted into two separate pipelines, ‘faking’ a method with a constant timespan.

Tests with and without the NetworkLatencyElement added show no measurable difference in the number of milliseconds used to complete the execution of both pipelines.

Without the worker process the pipeline always returns in 0ms adding plausibility to the claim that the NetworkLatencyElement indeed adds no measurable overhead to the pipeline.

Details substantiating these claims can be seen in appendix 15.2.1 in the ‘NetworkPerformanceElement performance testcases’.
11.5 Logging SLA metrics

The DataDistributor solution already has a comprehensive logging mechanism which must be extended to fit the SLA Logging mechanism requirements.

For the LogPoint SIEM system to ‘know’ whether logged SLA metrics meet SLA requirements both the SLA category type and the performance measurement data must be logged. This way the metrics logged can be compared to the expected response time of the SLA category.

While SLA categories are considered final and will probably never be extended, the same may not hold true for service- and data types, thus to keep the codebase flexible and expandable an interface: ISLACategory could be created with the SLA requirement types from section 6.1:

![Figure 7 - The ISLACategory interface](image)

This will ensure that no matter what future service- authentication- and data types would be introduced we can extend the code without modification by adding another concrete version of the interfaces like the suggested ComplexServiceType, StructuredDataType and NoAuthenticationType:

![Figure 8 - Interfaces for SLA category types](image)
A concrete implementation of the ISLACategory interface could be the following:

```java
public enum SlaCategoryType {
    NotAssigned = 0,
    SimpleStructured = 1,
    SimpleUnstructured = 2,
    OrdinaryStructured = 3,
    OrdinaryUnstructured = 4,
    Complex = 5,
    OutOfCategory = 6,
    SimpleStructuredNoAuthentication = 7,
    SimpleUnstructuredNoAuthentication = 8,
    SimpleStructuredPreAuthenticated = 9,
    SimpleUnstructuredPreAuthenticated = 10,
    SimpleUnstructuredWithAuthentication = 11,
    SimpleUnstructuredWithAuthentication = 12
}
```

Figure 9 - SLA Categories, inheritance approach

An enumeration approach for each of the types would offer less flexibility at the cost of future extension by modification, with low complexity and with less ‘flexibility overload’ but could be chosen in this case:

```
public class ConcreteSlaCategory extends SlaCategory {
    private ISlaServiceType _slaServiceType;
    private ISlaDataType _slaDataType;
    private ISlaAuthenticationType _slaAuthenticationType;

    public ConcreteSlaCategory(ISlaServiceType _slaServiceType, ISlaDataType _slaDataType,
                                ISlaAuthenticationType _slaAuthenticationType) {
        this._slaServiceType = _slaServiceType;
        this._slaDataType = _slaDataType;
        this._slaAuthenticationType = _slaAuthenticationType;
    }

    public ISlaAuthenticationType getAuthenticationType() {
        return _slaAuthenticationType;
    }

    public ISlaDataType getDataType() {
        return _slaDataType;
    }

    public ISlaServiceType getServiceType() {
        return _slaServiceType;
    }
}
```

Figure 10 - SLA Categories, enumeration approach

This however, would only be advisable if there is absolute guarantee that the types are not extended in the future as this could result in a combination explosion that would be impossible to manage using the enumeration approach.
The combination of the identified SLA types comprising an SLA category using the flexible interface approach is chosen for SLA logging along with performance data.

The SLACategory for the ServiceType ‘simple’ can only be classified on runtime because there is no way of knowing whether the end user will access the service with, without, or with pre-authenticated credentials.

**Logging SLA metrics**

The DataDistributor ISwitchboardServicesLogger interface has five levels of logging in declared methods: Critical, Error, Information, Verbose and Warning.

As SlaCategory is not part of the current logging interface this must be extended to reuse this for the intended purpose.

The ISwitchboardServicesLogger interface with the Information method:

```java
void Information(
    string message,
    string dataAuthority,
    string serviceName,
    string mimeTypeFormat,
    LogType logType = LogType.Activity,
    string fileName = null,
    SwitchboardServicesOptions optionalS = null,
    string topic = null,
    [callerFilePath] string callerFilePath = null,
    [CallerLineNumber] int callerrLine = 0);
```

Figure 11 - The original ISwitchboardServicesLogger interface

This can be overloaded by extending the interface without modification into a new one called ISwitchboardSlaServicesLogger with the same signature + SlaCategory which is used instead of the previous class ISwitchboardServicesLogger for the startup of the prototype switchboard application.

```java
public interface ISwitchboardSlaServicesLogger : ISwitchboardServicesLogger
{
    void Information(
        string message,
        SlaCategory slaCategory,
        string dataAuthority,
        string serviceName,
        string mimeTypeFormat,
        LogType logType = LogType.Activity,
        string fileName = null,
        SwitchboardServicesOptions optionalS = null,
        string topic = null,
        [callerFilePath] string callerFilePath = null,
        [CallerLineNumber] int callerrLine = 0);
```

Figure 12 - The extended ISwitchboardSlaServicesLogger interface
Unfortunately, most of the helper methods in the concrete implementing class: `EnterpriseLibraryLogger` have been declared `private` leaving two choices for implementing extensions with an additional `Information` method that includes SLA logging information:

1. Extend the `EnterpriseLibraryLogger` class without modification and implement all helper methods again (copy/paste) leaving duplicate code in both implementations - this is not recommended as bug fixes and future changes would have to be done in ‘all’ copies of the code leading to poor maintainability.

2. Refactor the `EnterpriseLibraryLogger` class changing the protection level of the helper methods to `protected` making them accessible from inherited classes. This requires extension by modification which is not advisable but, in this case, the chosen method as we’re not accessing the package from outside but merely allowing simple inheritance of the object’s helper methods.

The current implementation shows traces of previous refactoring using extension by modification as there are numerous overloaded methods for logging `Information`, `Error`, `Warning`, `Verbose` and `Critical` log messages but only one `DoWriteXmlEntry` method used by all methods.

The method has been extended to fit all versions of the log methods by setting variables to `null` or empty string like the case where the method cannot know whether the `logTopic` has a value or not and is thus tested and populated with a single space as text if it is either `null` or `whitespace`:

```csharp
```

This thesis will not go deeper into refactoring of the logging framework but acknowledge that the current implementation is not optimal for creating flexible and reliable software and will require further refactoring to support these traits.
As stated earlier in this section, some SLA categories can only be classified on runtime and with the SlMeasurementElement being second in line we cannot rely on this to set the SLA category correctly as the information to do so may not be present in the context. This could happen in the instance where a given service requires authentication but this has not been completed yet and the service type has not been identified yet.

To ensure that the SLA category is present in the context when the SLA measurement has to be calculated this information needs to be stored where it can be accessed by middleware components.

The obvious implementation is to put the SlCategory object on the CallContext object which is the only object created for sharing information between middleware components. This, however cannot be achieved without extending the ICallContext interface by modification, adding a getter/setter to the interface and implementing class:

```csharp
namespace DataDistributor.Switchboard.Common
{
    public interface ICallContext : IDisposable
    {
        IDataDistributorClaimsPrincipal Principal { get; }
        Logging.Loggers.Enums.SlaCategory Slacategory { get; set; }
        IServiceExecutor CurrentServiceExecutor { get; set; }
        ...
    }
}
```

**Figure 14 - Extended ICallContext interface snippet**

```csharp
namespace DataDistributor.Switchboard.Pipeline
{
    public sealed class CallContext : ICallContext
    {
        ...
        public IPipelineRequest Request { get; }
        public IPipelineResponse Response { get; private set; }
        public IRequestData ParseRequest { get; set; }
        public SlaCategory Slacategory { get; set; }
        ...
    }
}
```

**Figure 15 - CallContext implementation snippet**

This is an acceptable tradeoff, because the CallContext class is already used by all middleware components and the addition of the field will not break implementation of other components.
Now the SlaMeasurementElement class can refer to the SlaCategory by using the implementation of the ISwitchboardServicesLogger interface in the following way:

```csharp
callContext.Logger.Information("Information goes here",
callContext.SlaCategory,
callContext.GetDataAuthorityName(),
callContext.GetServiceName(),
callContext.GetMimeResponseType());
```

**Figure 16 - Usage of the refactored ISwitchboardServicesLogger interface**

### Using the logger

The task of mocking up the ISwitchboardServicesLogger was implemented in testcase #2 with the following code:

```csharp
public void Test_1_execute_CallRequest_SomethingShouldBelogged()
{
    // Arrange
    var loggerMock = new Mock<ISwitchboardServicesLogger>();
    loggerMock.Setup(logger => logger.Information(
        It.IsAny<string>(),
        It.IsAny<string>(),
        It.IsAny<string>(),
        It.IsAny<string>(),
        It.IsAny<string>(),
        It.IsAny<string>(),
        It.IsAny<string>(),
        It.IsAny<string>(),
        It.IsAny<int>()));

    var loggerFactoryMock = new Mock<ILoggerFactory>();
    loggerFactoryMock.Setup(f => f.CreateSwitchboardLogger()).Returns(loggerMock.Object);

    var callContextMock = new Mock<ICallContext>();
    callContextMock.Setup(ctx => ctx.Logger).Returns(loggerMock.Object);

    var pipeline = new SlaMeasurementElement_v1();

    // Act
    pipeline.Execute(callContextMock.Object);

    // Assert
    loggerMock.Verify(logger => logger.Information("Something...",
        It.IsAny<string>(),
        It.IsAny<string>(),
        It.IsAny<string>(),
        It.IsAny<string>(),
        It.IsAny<string>(),
        It.IsAny<string>(),
        It.IsAny<string>(),
        It.IsAny<string>(),
        It.IsAny<int>()));
}
```

**Figure 17 - Mocking up logging with Moq**
Testcase #2 is complete and passes validation, showing that the `Logger.Information` method was called once with the text “something”.

The class `SwitchboardServicesOptionals` shown in Figure 19 - `SwitchboardServicesOptionals additions` is a class created with the clear intention of being extendable by modification as the team knew that a placeholder class would be needed to keep track of optional elements in different middleware elements and only create the optional elements needed for a given middleware component.
The concrete implementation of the previously extended interface `ISwitchboardServicesLogger` is already capable of receiving the optional elements and must only use the ones created for this specific middleware element:

```csharp
public void Information(
    string message,
    LogType logType = LogType.Activity,
    string fileName = null,
    ServicesOptionals optionals = null,
    string topic = null,
    [CallerFilePath] string callerFilePath = null,
    [CallerLineNumber] int callerLine = 0
)
{
    ValidateServicesParameters(logType);

    var opt = optionals ?? new ServicesOptionals();

    DoWriteXmlEntry(
        TraceEventType.Information,
        opt.SlaCategory ?? SlaCategory.NoCategory,
        opt.SlaTarget,
        opt.SlaCompliance,
        opt.SlaPercentOver,
        ...
    );
}
```

**Figure 20 - Using the new SwitchboardServicesOptionals**

The correct implementation however, would be to accept that this class will most definitely change in the future and because of that isolate the variation of elements in an interface `ISwitchboardServicesOptionals`, creating concrete implementations of this interface as needed.

Implementing the new interface will require modifications in the classes using the `SwitchboardServicesOptionals` class using the interface instead of the previous concrete class including our newly created `ISwitchboardServicesLogger` interface:

```csharp
public interface ISwitchboardSlaServicesLogger : ISwitchboardServicesLogger
{
    void Information(
        string message,
        SlaCategory slaCategory,
        string dataAuthority,
        string serviceName,
        string mimeResponseFormat,
        LogType logType = LogType.Activity,
        string fileName = null,
        ISwitchboardServicesOptionals optionals = null,
        string topic = null,
        [CallerFilePath] string callerFilePath = null,
        [CallerLineNumber] int callerLine = 0;
    )
}
```

**Figure 21 - ISwitchboardServicesLogger with ISwitchboardServicesOptionals interface**
The optional elements are iterated and logged if available in the context.

With these properties the middleware can log detailed SLA information about a given service by referencing the `SlaCategoryType` assigned to the requested service type.

The mapping of the `SlaCategoryType` to services is added to the database model to create services with the correct `SlaCategoryType` – this will have to be done in the “Administration Portal” project, a sub-project in the DataDistributor solution that will not be covered by this thesis.

```csharp
public class SwitchboardServicesOptionals : ISwitchboardServicesOptionals
{

    public long? SlaTarget { get; set; }

    public bool? SlaCompliance { get; set; }

    public decimal? SlaPercentOver => CalculateSlaPercentOver();

    public long? ResponseTime { get; set; }
}
```
11.6 Ensuring SLA Metrics accuracy

To ensure that the sequence diagram in Figure 4 - SLA Metrics sequence diagram is followed the SlaMeasurement-Element middleware must be injected in the correct order of the pipeline as stated in section 11.4 but also ensure that the measurement is completed when all other CallContext handlers have finished and all other middleware components have been invoked and have finished execution. This is a required to ensure that the log timer does not stop until the execution of the request has truly finished.

This implementation of the Execute method ensures that the timer is started as soon as possible in the pipeline and will not calculate SLA metrics until the remaining pipeline elements have finished execution:

```csharp
public override void Execute(ICallContext callContext)
{
    var stopWatch = new Stopwatch();
    try
    {
        stopWatch.Start();
        Next.Execute(callContext);
    }
    finally
    {
        stopWatch.Stop();
        LogPerformanceInformation(callContext, stopWatch);
    }
}
```

Figure 23 - SLA metrics timer implementation

With all identified refactoring elements identified the remaining task at hand is to implement the definitive version of the middleware class method ‘LogPerformanceInformation’.
Building, deploying and testing architectural prototypes

The ‘experimental architectural prototypes’ as defined by [Bardram, 18] and used in this thesis are software versions of the Switchboard component configured and refactored using the new SLA metrics middleware described in the previous sections.

Characteristics of the experimental APs suggested:

Issues addressed:

- Performance; without this middleware we cannot assert that our REST services are complying with SLA requirements. While the AP will not be used to measure performance per se, it will enable the project to log information about whether REST services perform to specification or not IF the AP work is successful.

Concerns:

- Integration with the Switchboard pipeline making the SLA metrics middleware conform to the other components of the system.

Addressed risks:

- Architect unfamiliarity with OWIN middleware and programming.
- Ensuring maintainability by building the SLA metrics middleware using the same approach as the remaining SwitchBoard components.

The AP’s are to some extent created for the architect to explore ways of using OWIN components in the SwitchBoard solution instead of implementing SLA metrics using patterns that might be familiar to the architect but potentially ill-advised for OWIN projects.

The prototypes are installed in the test environments described in section 10 - Test environments.

The DataDistributor solution uses [SaltStack, 19] for installation and configuration in environments, a configuration management and orchestration tool which this thesis will not go into details about other than stating that it facilitates automated deployment of software binaries, configuration and transformations of environments specific settings.
11.8 Running load tests with extended logging.

This thesis will not be covering performance testing of individual services per se but concentrate on collecting and showing the logs generated from load tests. For this purpose, SmartBear [SoapUI, 17], a multipurpose testing tool will be used for generating service requests.

Two categories of tests will be run in this thesis:

**Single test:** Create one REST call querying one method in one service one time.

*The objective of this testcase is to assert that the configured test executes and logs SLA metric data that can be collected in the LogPoint system.*

**Load test:** Create \( x \) number of REST calls for one method in one service during \( y \) minutes using \( z \) number of simultaneously running threads.

*The objective of this testcase is to collect SLA metric data for the tested method in a configured timeframe where the service is called continuously by the configured number of threads.*

To collect SLA metrics, the following REST services and methods have been selected for testing. The services themselves, and the data they provide are not interesting, but their SLA type is.

<table>
<thead>
<tr>
<th>Testcase #</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service name</td>
<td>Matrikel</td>
</tr>
<tr>
<td>Service method</td>
<td>samletFastEjendom</td>
</tr>
<tr>
<td>Type</td>
<td>Complex</td>
</tr>
<tr>
<td>Data type</td>
<td>Structured</td>
</tr>
<tr>
<td>SLA requirement</td>
<td>2500ms</td>
</tr>
<tr>
<td>Authentication</td>
<td>With authentication</td>
</tr>
</tbody>
</table>

*The Matrikel services provide cadastral information about Danish land property, either governmental, private or industry owned. The data is owned and maintained by GeoDataStyrelsen.*
Testcase # | 2  
---|---  
Service name | DAR  
Service method | adresseTilHusnummer  
Type | Simple  
Data type | Structured  
SLA requirement | 60ms  
Authentication | Pre authenticated

The DAR services provide address information about, private, governmental and industry entities in Denmark. The data is owned and maintained by 'Denmark’s Adresse register' governed by ‘Styrelsen for Dataforsyning og Effektivisering’ SDFE.

A new SOAPUI project is started and configured with the two selected URLs:

![soapui project](image)

Figure 24 - SOAP UI REST test project

The single tests produce JSON output and the Soap UI tool yields the following for the DAR service:

![soapui test](image)

Figure 25 - DAR single test

In the screenshot of the single DAR test above the following REST query output can be seen (snippet):
<table>
<thead>
<tr>
<th>Method</th>
<th>GET</th>
</tr>
</thead>
<tbody>
<tr>
<td>Endpoint</td>
<td><a href="https://test03-services.datafordeler.dk">https://test03-services.datafordeler.dk</a></td>
</tr>
<tr>
<td>Resource</td>
<td>/DAR/DAR/1/REST/adresseTilHusnummer</td>
</tr>
<tr>
<td>Parameters</td>
<td>format=xml, adresseid=0a3f50c7-0a07-32b8-e044-0003ba298018</td>
</tr>
<tr>
<td>Response time</td>
<td>94ms (client turnaround time)</td>
</tr>
<tr>
<td>Output</td>
<td>&lt;adresseTilHusnummer&gt;0a3f5099-66c3-32b8-e044-0003ba298018&lt;/adresseTilHusnummer&gt;</td>
</tr>
</tbody>
</table>

Load tests are run for 45 seconds with 8 threads and a start delay of 800ms to ensure ‘some’ load on the services. As this thesis does not cover load testing per se, the number of tests and actual performance of the services is uninteresting in this thesis and as such, of no consequence. The only objective is to generate SLA metrics data to be collected for SLA compliance analysis.

The load tests were run consecutively after each other and the output from the tests show that the services may be struggling in being SLA compliant the Matrikel load test reaching no more than 357ms but DAR reaching 1147ms – which is far higher than the 60ms goal:

![Figure 26 - Matrikel load test 45 seconds](image1)

![Figure 27 - DAR load test 45 seconds](image2)
With both single and load tests showing that max response time may not meet SLA requirements, the collection of SLA metrics is of high importance.

However, until SLA metrics are gathered from the log repository there is no way of knowing whether the services are SLA compliant as the tests shown above include network latency and do not show the response time of individual requests. The next sections will cover this in detail.
11.9 Collecting compliance data: cloud test environment

The DataDistributor Switchboard components, and all other .Net implemented components running on Windows in the solution log to the Windows event log.

The LogPoint SIEM solution collects, normalizes and stores these logs continuously using a log client installed on all servers.

![Figure 28 - LogPoint overview](image)

For this thesis the inner workings, reporting, alerting etc. will not be covered. The only part of LogPoint interesting to this thesis are the search and compliance functions for showing the SLA metrics logged by the Switchboard SLA middleware implemented. These functions are available through both a web API and a webpage interface of which this these will focus on the webpage interaction.
11.10 Compliance calculations using LogPoint

To collect SLA metrics the LogPoint SIEM system is used to execute a series of queries to collect the logs from the load tests run in section 11.8

The LogPoint web-interface

The normalized DataDistributor log data created by the architectural prototype looks like the for a DAR service request (unimportant details omitted):

```
<LogEntry>
  <Timestamp>2018-04-05T07:53:57.8697330Z</Timestamp>
  <LogLevel>Information</LogLevel>
  <LogType>Performance</LogType>
  <SourceType>Switchboard</SourceType>
  <SLAcategory>SimpleStructured</SLAcategory>
  <SLAtarget>60</SLAtarget>
  <SLAcompliant>1</SLAcompliant>
  <SLAPercentOver>0.00</SLAPercentOver>
  <MethodName>adresseTilHusnummer</MethodName>
  <ResponseTime>7</ResponseTime>
  <ServiceName>DAR</ServiceName>
</LogEntry>
```

In the LogEntry the following information is needed to calculate SLA compliance:

1. ResponseTime
2. MethodName
3. ServiceName

With this data we can gather a list of logs and use third-party applications to calculate SLA compliance by inputting the search parameters into the LogPoint web-interface search form and outputting the result to an XML file.

There are, however also chart tools embedded in the LogPoint SIEM system that allow users to create quick compliance charts helping to visualize whether SLA compliance is met or not. For this we will use some of the ‘extra’ LogEntry data created that states whether an individual request is SLA compliant or not and which SLA category a service request belongs to:

1. SLAcompliant
2. SLAcategory
3. SLAtarget

The LogPoint web-interface is used in the following way to gather the SLA metrics data for the two services tested over one minute:
1. Specify time-range for the query – this is set to the minute where the load tests were run.

   **Note:** Collection of logs, over one minute is only useful for this thesis, as it is unimportant to have representative performance data – we just need to show that we log performance data that can be used for calculation.
   For production environments, SLA would be calculated monthly.

2. Specify which environment we want to see logs from (TEST03 in this case): "device_name"="DFD_DT3***"

3. Select the logs with the correct service types:
   
   (("ServiceName"="DAR" "SLAcategory"="SimpleStructured") OR "ServiceName"="MATRIKEL" response time: "ResponseTime"=*  

4. Create a timechart using the ‘extra’ LogEntry data:
   
   timechart count(SLAcompliant=1) as COMPLIANT, count(SLAcompliant=0) as NON_COMPLIANT, max(ResponseTime

5. Order the query by SLA Category, SLA Target and service name:
   
   by SLAcategory, SLAtarget, ServiceName

6. Range the time chart to 1 minute:
   
   every 1 minutes
Output from the combined query: "ResponseTime"=* \n(("ServiceName"="DAR" "SLAcategory"="SimpleStructured") OR \n"ServiceName"="MATRIKEL") "device_name"="DFD_DT3*" | timechart \ncount(SLAcompliant=1) as COMPLIANT, count(SLAcompliant=0) as NON_COMPLIANT, max(ResponseTime) by SLAcategory, SLAtarget, 
ServiceName every 1 minutes:

The time-chart shows that all complex queries for the *Matrikel* service 
are SLA compliant without no further calculation needed as not one 
request has a response time over the accepted 2500ms limit.

For the *DAR* Simple service 31 request were non-compliant i.e. did not 
meet the 60ms response time limit. To check that the required 95 
percentiles of requests are within SLA compliance we can do the 
following calculation:

\[
\text{percentile compliant requests} = 100 - \left( \frac{\text{# of non_compliant requests}}{\text{total # of requests}} \right) \Rightarrow \]

\[93.981\% = 100 - \left( \frac{31}{484 + 31} \right) / 100\]

Looking at the results it is seen that the DAR service would not be SLA 
Compliant IF it can be said that the load tests performed are 
representative for the usage of the service over a period of one month. 
The comparison of service usage and SLA compliance is out of scope for 
this thesis, but it is shown that the SLA metrics are logged, acquired 
from the LogPoint system and can used to calculate SLA compliance for 
the DataDistributor system using either third-party applications to 
calculate the SLA compliance data or by using the built-in chart features 
of the LogPoint SIEM system.
12 Implementing an APM tool

In this chapter the efforts of implementing an APM tool in the DataDistributor solution is covered.

12.1 APM tool requirements

In this section the APM tool requirements will be listed and explained ending up in a short concrete list of requirements.

Language / Platform support

In the previous sections it was explained that the primary bulk of the DataDistributor solution code is created in C# projects making the ability to monitor C# code in both OWIN (asynchronous) and standalone applications of paramount importance.

To ensure that identified performance problems in .Net code is easily identified and fixed the APM should offer code-level insight to pinpoint the origin of the problem.

Although a fair number of components run on Linux, most of these components are third party applications and services leaving the information one could gather with the APM from these services of great interest but also limited ability to improve performance in these components in the DataDistributor team, and a thus the APM must support Microsoft Windows server 2012 servers.

Non-performing third-party software will in most cases be reported to the vendor by opening a support ticket and implementing an update of that software if/once the performance has been improved.

A huge amount of performance optimization has already been done on REST service configurations. The service configurations are primarily implemented in SQL and data stored in PostgreSQL server clusters running on Ubuntu Linux servers. Because of this the APM must also support monitoring of PostgreSQL SQL performance on Ubuntu Linux servers to show when bottlenecks in REST services are caused by SQL performance issues.
Monitoring of components on different servers

The DataDistributor solution is a very large solution comprising over 15,000 files whereof over 6,900 files are .cs source files. This means that APMs that require refactoring or additions to existing code to monitor a given component are less desirable than APMs without this requirement. It also means that the APM must be able to support many components running simultaneously on many servers.

The clients using the DataDistributor solution interact with a wide variety of components not all located on the same server. Because of this the APM must support end-to-end monitoring, correlating for instance, authentication on one server with REST service execution on another to give the APM monitor user as much information about an entire workflow as possible.

Monitoring of components by non-developers

Because performance optimization is primarily a developer task, this does not mean that others shouldn’t be able to identify bottlenecks and other performance issues in the DataDistributor solution. The DataDistributor team has many operations specialists, delivery managers etc. that have an interest in the solution running to the best of its ability. Because of this, the APM should offer a ‘non-technical’ graphical interface that will allow ‘non-developers’ to monitor the selected components in the DataDistributor solution to identify if/when a performance issue occurs.

APM installation types

APM’s should offer local implementation of the controller service making it easy to start using the products. Agents solely implemented by sending metric data to SaaS collection servers however, pose a potential security risk.

If the agents installed on SaaS servers that are monitored are in some way compromised, production data could leak into malicious hands and servers could be compromised.

There could also be legal issues involved as the DataDistributor contains sensitive information on file shares and in databases from which no outbound connections to foreign countries is allowed. A detailed investigation for each APM only offering the controlling service as a SaaS would have to be undergone to qualify or disqualify the product.

For this thesis such an undertaking is not realistic and as a result, any APM chosen will have to offer an on-premise version of the controller software to be qualified. A SaaS version for non-production
environments would be ‘nice-to-have’ to speed up the installation of the APM and will also be considered.

**APM Platform stability, support and development**

The DataDistributor solution must have access to support for most third-party components used to fix problems that arise in components that the development team cannot fix themselves. As such an APM must provide a proven stable and supported platform that shows active continuous bug-fixing, and product development.

**Requirements summary:**

1. Must support IIS
2. Must support C# applications
3. Must support OWIN (asynchronous) applications
4. Must support code-level insight for .Net
5. Must support PostgreSQL SQL performance monitoring
6. Must run on Windows server 2012
7. Must run on Ubuntu 14.04 Linux server
8. Must support many components running simultaneously
9. Must support end-to-end workflow monitoring of client interaction with the solution.
10. Must support graphical interface that can be used by non-developers to identify performance issues
11. SaaS support (nice-to-have)
12. On premise installation support
13. Must have stable platform version
14. Must have dedicated support that can be contacted in case of problems with the APM
15. Must have continuous development and bug-fixing of the APM product.
12.2 APM listing and selection

In this section a listing of identified APMs matching the requirements is displayed along with the decisions made to choose a suitable platform.

The list has been created by crawling the web using the search term: “.net application performance monitoring” and manually screening the APMs from the search results.

<table>
<thead>
<tr>
<th>Name and vendor</th>
<th>AppDynamics, AppDynamics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Website</td>
<td><a href="https://www.appdynamics.com/">https://www.appdynamics.com/</a></td>
</tr>
<tr>
<td>APM service type</td>
<td>SaaS/On premise service</td>
</tr>
<tr>
<td>APM data collection</td>
<td>Agent based</td>
</tr>
</tbody>
</table>

Figure 29 - AppDynamics APM

<table>
<thead>
<tr>
<th>Name and vendor</th>
<th>DynaTrace, DynaTrace</th>
</tr>
</thead>
<tbody>
<tr>
<td>Website</td>
<td><a href="https://www.dynatrace.com/solutions/">https://www.dynatrace.com/solutions/</a></td>
</tr>
<tr>
<td>APM service type</td>
<td>SaaS/On premise service</td>
</tr>
<tr>
<td>APM data collection</td>
<td>Agent based</td>
</tr>
</tbody>
</table>

Figure 30 - Dynatrace APM

<table>
<thead>
<tr>
<th>Name and vendor</th>
<th>Retrace, Stackify</th>
</tr>
</thead>
<tbody>
<tr>
<td>Website</td>
<td><a href="https://stackify.com/retrace-for-net/">https://stackify.com/retrace-for-net/</a></td>
</tr>
<tr>
<td>APM service type</td>
<td>SaaS service/On premise service (not verified when this list was initially created)</td>
</tr>
<tr>
<td>APM data collection</td>
<td>Agent based</td>
</tr>
</tbody>
</table>

Figure 31 - Retrace APM
The APMs listed in above all offer a SaaS implementation of the controller service making it easy to start using the products.

The Retrace APM however, because it seems that it does not offer an on-premise installation, from what the online documentation and web-searches describe leaving only AppDynamics and DynaTrace as the final candidates from the selection list. Further investigation into the Retrace APM was abandoned.

The following table shows information gathered about the candidates: AppDynamics (AP), DynaTrace (DT) and Retrace (RT):

<table>
<thead>
<tr>
<th>Requirements</th>
<th>AD</th>
<th>DT</th>
<th>RT</th>
</tr>
</thead>
<tbody>
<tr>
<td>IIS support</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>C# Support</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>OWIN support</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>.Net code-level insight support</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>PostgreSQL support</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Windows server 2012 support</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Ubuntu 14.04 Linux support</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Support for monitoring many components running simultaneously</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>End-to-end workflow monitoring support</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>SaaS installation support</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>On premise installation support</td>
<td>x</td>
<td>x</td>
<td>-</td>
</tr>
<tr>
<td>Graphical interface</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Stable platform version</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Dedicated product support</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Continuous development and bug-fixing</td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

From the requirements point of view both AppDynamics and DynaTrace cover the identified items and both platforms and are at the top of the APM industry according to section 15.3 Appendix 3 – Application Performance Management Buyer’s Guide and Reviews [20].

This thesis will cover usage of the DynaTrace APM, selection solely based on both AppDynamics and DynaTrace APMs being qualified and because a ‘quick-dive’ into the APMs documentation shows potential quicker initial startup time for the DynaTrace APM.
12.3 DynaTrace APM Installation

A DynaTrace account is created, lasting for 15 days automatically setting up an Amazon Web Services (AWS) DynaTrace account where all data regarding monitored applications and infrastructure components is displayed. As this is a new account, only sample applications are available upon first invocation of the service.

Using the local test environment, DynaTrace OneAgent.exe is installed and configured according to the documentation.

The Agent for the server automatically establishes a connection to the correct DynaTrace account and starts sending trace data from the server to the DynaTrace Web Service account.

![Figure 32 - DynaTrace AWS home screen](image-url)
12.4  DynaTrace APM usage

In this section the usage of the DynaTrace APM will be covered by:

1. Installing DynaTrace agent onto local test environment
2. Creating OWIN application that will show up on the DynaTrace SaaS site with notable fake ‘performance issues’.
3. Validating that the OWIN application is monitored correctly and performance metrics can be seen in the DynaTrace SaaS.
4. Installing DynaTrace agent onto cloud test environment(s)
5. Verifying that DataDistributor services are monitored and performance metrics are gathered correctly down to .net code level – identifying performance metrics at method level.

12.4.1  Local testing

Before embarking on full-scale test environment roll-out, one of the previously created test OWIN applications was modified to implement some delays that ‘should’ show up in an APM tool. This application was subsequently published onto the IIS server.

The purpose of this task is just to create a quick proof-of-concept, showing that the DynaTrace in fact shows performance metrics from a locally developed application.

Changes to the test OWIN application consists of a simple static method that is called from the main application startup counting to 10,000,000 and sleeping for two seconds:

```
private void PrintCurrentIntegratedPipelineStage(IOwinContext context, string msg)
{
    OwinSlepperClass.SleepAndWork;
    context.GetTextWriter("HostTraceOutput").WriteLine("Current IIS event: " + currentIntegratedpipelineStage + " Message: " + msg);
}
```

```
namespace owin

    1 reference | 0 changes | 0 authors, 0 changes
    public class Slepper
    {
        1 reference | 0 changes | 0 authors, 0 changes
        public static void SleepAndWork()
        {
            var i = 0;
            while (i < 10000000)
                i++;
                Thread.Sleep(2000);
        }
    }
```

Figure 33 - Snippet from test OWIN application startup method and Slepper.class
The DynaTrace APM Application monitor overview shows the initial tests with the OWIN middleware (owintest:80 in the screenshot below) before drilling down into the application performance details:

Figure 34 - OWIN test application DynaTrace overview
12.4.2 Cloud environment testing

The tests run in the cloud test environment hosting the new SLA middleware prototype are all connected to the ‘DAR’ register which was previously mentioned in section 11.8 testcase #2:

<table>
<thead>
<tr>
<th>Testcase #</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Service name</td>
<td>DAR</td>
</tr>
<tr>
<td>Service method</td>
<td>adresseTilHusnummer</td>
</tr>
<tr>
<td>Type</td>
<td>Simple</td>
</tr>
<tr>
<td>Data type</td>
<td>Structured</td>
</tr>
<tr>
<td>SLA requirement</td>
<td>60ms</td>
</tr>
<tr>
<td>Authentication</td>
<td>Pre-authenticated</td>
</tr>
</tbody>
</table>

Testing is done to assert whether DynaTrace can meet any/all of the following requirements stated in section 6.2 ‘Analyzing software component performance’

**Requirement #1** The platform must monitor individual software components and their contribution to the performance of a service request measured in milliseconds

**Requirement #2** The APM system must show detailed component information to improve performance based on the source code

**Requirement #3** The APM should be able to show live performance metrics…

Test is initiated by a standard browser by querying the DAR ‘adresseTilHusnummer’ service with the URL: [https://dev01-services.datafordeler.dk/DAR/DAR/1/REST/adresseTilHusnummer?format=JSON&adressId=0a3f50c7-0a07-32b8-e044-0003ba298018&](https://dev01-services.datafordeler.dk/DAR/DAR/1/REST/adresseTilHusnummer?format=JSON&adressId=0a3f50c7-0a07-32b8-e044-0003ba298018&)
Data, comprised of a single identifier pointing to the object that contains the ‘husnummer’ data is, according to the browser’s built-in developer tools, returned in 43ms:

![Figure 35 - Browser developer tool view](image1.png)

DynaTrace recorded the request in the following view that, already at first glance, has a lot of information about service metrics. This ‘simple’ view meets part of requirements #1 and #2 of showing metrics in milliseconds for a given service in a live view:

![Figure 36 - Web requests view](image2.png)
Drilling down into the /DAR/DAR/1/REST/adresseTilHusnummer REST service reveals high-level component metrics:

![Figure 37 - Rest service details](image)

Database usage drilldown shows how long the queries to the publish database took:

![Figure 38 - Database usage drill down](image)

Lock wait time which contributes 20.6ms of the total response time shows on drilldown detailed method level metrics including the SLA middleware included in the prototype:

![Figure 39 - Lock wait time drill down](image)
12.5 APM conclusions

As discussed in section 12.1 The nearest, and thus selected, DynaTrace SaaS installation is in Ireland – validating the claim stating that data security must be prioritized as it is impossible to get a DynaTrace SaaS platform in Denmark at the time of writing this thesis.

Although there are a lot of proven and mature APM platforms available, only a few offered .Net asynchronous web application support needed to test the OWIN middleware used in the DataDistributor system. This had a dramatic impact on the number of identified, suitable platforms for the solution, it is this authors opinion however, that the DynaTrace APM offers all required functionality defined in this thesis.

Elaboration: The DynaTrace APM offers insight into the tested middleware at .net code/method level allowing the project to identify performance bottlenecks, compare REST service performance across different data vendors (registers) measured in milliseconds.

The APM offers ‘live’ coverage of the applications monitored to quickly identify performance bottlenecks by drilling down to code level to see which calls/functions etc. are causing a given problem.

The DynaTrace APM therefore meets the requirements of this thesis.
13 Conclusion

Introducing logging of SLA metrics across all services in the DataDistributor system.

The thesis has shown how implementing SLA metric logging across all REST services in the DataDistributor system was done as well as the methods and techniques used in doing so. Pitfalls, alternative implementations and necessary direct changes were discussed and substantiated.

The selected approach was implemented using TDD principles and its usability confirmed.

Creating compliance data from gathered SLA metrics.

The data logged in the LogPoint SIEM system was used to produce SLA compliance data showing whether services met DataDistributor SLA compliance or not.

It was shown that the chosen implementation works and can be refined for production usage using architectural prototypes containing the selected approach.

Implementing an APM platform for analyzing and improving software component performance.

Three top-tier APM platforms were identified and their capabilities compared to a list of DataDistributor APM requirements.

While implementing the DynaTrace platform the requirements set in the problems statement were tested and discussed:

- The DynaTrace platform is capable of monitoring individual software components and their contribution to the performance of services measured in milliseconds.
- The platform supersedes the expectations and requirements for showing component information at source/method level to be able to identify and thus improve performance of these components.
- Live monitoring of components during load testing showed that DynaTrace is a mature tool with a substantial amount of live performance metric capabilities.

DynaTrace is a huge product with hundreds of ways to monitor platforms, applications etc. This however also dictates specialist knowledge to apply the platform in a usable and secure manner because the amount of setup and monitoring capabilities, along with the documentation tied to it cannot be embraced by an individual developer.
It is the opinion of this author that APM platforms are best implemented as shared services in operations or ‘devops’ departments with specialists maintaining them to give the best quality of service to solutions like the DataDistributor system and to spread the costs of running the APM platform.

From a wider perspective this thesis has shown how SLA metric logging can be achieved generically using a central component in an OWIN application and how log data can be collected to show ‘real life’ SLA metrics using a SIEM system to ultimately calculate SLA compliance. The thesis also shows how one of the market leading APM platforms can be used to identify performance bottlenecks in an OWIN application with moderate investments in time and setup using architectural prototypes in a test environment.
14 References


https://www.simplethread.com/Beginning-Mocking-With-Moq-3-Part-1/


15 Appendix

15.1 Appendix 1 – SLA requirements snippets

This section contains snippets from the SLA requirements for the DataDistributor system

Web services

For opslag i Systemet med web service baserede Tjenester på Platformen differentieres mellem Simple, Almindelige og Komplekse forespørgsler.

<table>
<thead>
<tr>
<th>Kravnr.</th>
<th>7.1</th>
<th>Kravtitel</th>
<th>Svartider for web service baserede Tjenester</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kravtype</td>
<td>MK</td>
<td>Kravområde</td>
<td>Web services</td>
</tr>
<tr>
<td>Kravbeskrivelse</td>
<td>Web service-baserede Tjenester kategoriseres som enten Simple, Almindelige eller Komplekse. For hvert kald af en sådan Tjeneste registreres, om kaldet krævede autentificering af brugerens credentials eller om der anvendes præautentificerede credentials, og kaldet indgår i beregningen af driftseffektivitet baseret på det tilsvarende servicemål som specificeret i Tabel 1. Web service-baserede Tjenester, der ikke kræver autentifikation skal opfylde samme servicemål som Tjenester med præautentificerede credentials.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Tabel 1 Svartider for web service-baserede Tjenester

<table>
<thead>
<tr>
<th>Tjeneste-kategori</th>
<th>Beskrivelse/evt. forudsætninger</th>
<th>Data-type</th>
<th>Opfyldelsesgrad i %</th>
<th>Max Antal Sek.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simpel, præ-autentificert eller uden</td>
<td>Tjeneste der, ud fra inputparameter entydigt identifierer en specifik entitet, fremsøger og returnerer denne, som enten medtager præ-</td>
<td>Strukturet data</td>
<td>95</td>
<td>0,060</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ustruktureret data</td>
<td>90</td>
<td>0,060</td>
</tr>
<tr>
<td>Tjeneste-kategori</td>
<td>Beskrivelse/evt. forudsætninger</td>
<td>Data-type</td>
<td>Opfydelses-grad i %</td>
<td>Max Antal Sek.</td>
</tr>
<tr>
<td>-------------------</td>
<td>---------------------------------</td>
<td>-----------</td>
<td>---------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>autentificering</td>
<td>autentificerede credentials eller ikke kræver autentificering.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Simpel, med autentificering</td>
<td>Tjeneste der, ud fra inputparameter entydigt identificerer en specifik entitet, fremsøger og returnerer denne, som ikke medtager præ-autentificerede credentials. Et konkret eksempel på en simpel forespørgsel med autentificering er opslag af en virksomhed på baggrund af CVR-nummer. Applikationen der foretager kaldet medsender et VOCES/FOCES certifikat. Systemet er nødt til at bruge tid på at foretage autentificeringen af brugeren som en del af kaldet.</td>
<td>Struktureret data</td>
<td>95</td>
<td>0,090</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ustruktureret data</td>
<td>90</td>
<td>0,090</td>
</tr>
<tr>
<td>Almindelig</td>
<td>Tjeneste der, ud fra inputparametre, fremsøger data fra flere entiteter og returnerer disse, samt opslag der går på tværs af Registre, hvor opslag sker med nøgleværdier. Et konkret eksempel på en almindelig forespørgsel er fremsøgning af adresser på de ledende medarbejdere i en virksomhed fremsøgt på virksomhedens navn.</td>
<td>Struktureret data</td>
<td>90</td>
<td>1,0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Ustruktureret data</td>
<td>90</td>
<td>1,0</td>
</tr>
<tr>
<td>Kompleks</td>
<td>Tjeneste der fremsøger afledte data ud fra angivne inputparametre, og hvor</td>
<td>Struktureret data</td>
<td>90</td>
<td>2,5</td>
</tr>
<tr>
<td>Tjeneste-kategori</td>
<td>Beskrivelse/evt. forudsætninger</td>
<td>Data-type</td>
<td>Opfydelsesgrad i %</td>
<td>Max Antal Sek.</td>
</tr>
<tr>
<td>------------------</td>
<td>---------------------------------</td>
<td>-----------</td>
<td>-------------------</td>
<td>---------------</td>
</tr>
<tr>
<td></td>
<td>resultatet er en sammensat mængde af data, eller hvor data fremsøges fra flere forskellige Registre. Et konkret eksempel på en kompleks forespørgsel er fremsøgning af et kort over et sogn med alle virksomheder i en gruppe af brancher markeret.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
15.2 Appendix 2 – Datafordeler refactoring

15.2.1 Implementing Sla metrics

To get going the first test created simply defined an empty interface and a middleware class that threw a custom exception type to see that it was called correctly. The general idea of the test setup is:

1. Create new versions of the Middleware *SlaMeasureElement* class file for each test making it easy to go back in time, although bothersome to copy files – these are prepended ‘_v’ + version number. Ex: *SlaMeasureElement_v1.cs*
2. Create a set of unit tests linked to a specific version of the middleware by name.
3. Create a pipeline of x-number of middleware components. A version of the SlaMeasureElement being amongst them.
4. Run the *pipeline.execute()* method invoking all middleware components in turn as is the case of the entire ‘Switchboard’ application.

The first version of the test-class/-case/-outcome is shown here:

![Figure 40 - SlaMeasurementElement version 0](image-url)
The line:

```
[ExpectedException(typeof(System.NotImplementedException))]
```

is a built-in feature of .Net for ignoring Exceptions of the declared type which in this thesis helps with the test-driven development process, by allowing the test to expect the exception the test is approved. The opposite, i.e. just failing the test would have been just as valid to stay true to the TDD principles, but all green testcases make for a better overview of progress. This resolves testcase 1.

Creating a new Interface for the OWIN pipeline:

Fake it: create new empty interface and implementing class:
This middleware will simply call the next middleware in turn.

Inject the middleware into the pipeline:

```csharp
using DataDistributor.Switchboard.Common;
using DataDistributor.Switchboard.Pipeline.Elements.Interfaces;

namespace DataDistributor.Switchboard.Pipeline.Elements
{
    public class SlmMeasurementElement : PipelineElement, ISlmMeasurementElement
    {
        public SlmMeasurementElement()
        {
        }

        public override void Execute(CallContext callContext)
        {
            Next.Execute(callContext);
        }
    }
}

// PipelineElement Registrations
container.RegisterType<FaviconElement>(new ContainerControlledLifetimeManager());
container.RegisterType<NetworkLatencyElement>(new TransientLifetimeManager());
container.RegisterType<SlmMeasurementElement>(new TransientLifetimeManager());
container.RegisterType<PipelineLoggingElement>(new TransientLifetimeManager());
container.RegisterType<ServiceFinderElement>(new TransientLifetimeManager());
container.RegisterType<ReceiptsElement>(new TransientLifetimeManager());
container.RegisterType<CapabilitiesElement>(new TransientLifetimeManager());

// Elements
container.RegisterType<NetworkLatencyElement, NetworkLatencyElement>();
container.RegisterType<FaviconElement, FaviconElement>();
container.RegisterType<ProtocolSecurityElement, ProtocolSecurityElement>();
container.RegisterType<PipelineLoggingElement, PipelineLoggingElement>();
container.RegisterType<InvoiceLoggingElement, InvoiceLoggingElement>();
container.RegisterType<SlmMeasurementElement, SlmMeasurementElement>();

Create tests to show that the middleware works as expected.
NetworkPerformanceElement performance testcases:

Arrangement of class variables

```java
private long averagePerformance = 0;
private int tryCount = 5000;
Mock<SwitchboardServicesLogger> loggerMock = null;
```

Helper methods

```java
private void setLogger()
{
    // Arrange
    if (this.loggerMock == null)
    {
        this.loggerMock = new Mock<SwitchboardServicesLogger>();
        this.loggerMock.Setup(logger => logger.Information(
            It.IsAny<string>(),
            It.IsAny<string>(),
            It.IsAny<string>(),
            It.IsAny<string>(),
            It.IsAny<string>(),
            It.IsAny<string>(),
            It.IsAny<string>(),
            It.IsAny<string>(),
            It.IsAny<string>(),
            It.IsAny<string>(),
            It.IsAny<string>());
        loggerMock.Setup(x => x.CreateSwitchboardLogger()).Returns(loggerMock.Object);
    }
}
```

```java
private void perform(PipelineElement element, Mock<CallContext> callContextMock,
    Mock<SwitchboardServicesLogger> loggerMock, String text)
{
    // Act
    for (int i = 0; i < tryCount; i++)
    {
        element.Execute(callContextMock.Object);
        // Assert
        loggerMock.Verify(logger => logger.Information(It.IsAny<string>(),
            It.IsAny<string>(),
            It.IsAny<string>(),
            It.IsAny<string>(),
            It.IsAny<string>(),
            It.IsAny<string>(),
            It.IsAny<string>(),
            It.IsAny<string>(),
            It.IsAny<string>(),
            It.IsAny<string>(),
            It.IsAny<string>(),
            It.IsAny<string>(),
            It.IsAny<string>(),
            It.IsAny<string>(),
            It.IsAny<string>(),
            It.IsAny<string>(),
            It.IsAny<string>(),
            It.IsAny<string>(),
            It.IsAny<string>(),
            It.IsAny<string>(),
            It.IsAny<string>(),
            It.IsAny<string>(),
            It.IsAny<string>(),
            It.IsAny<string>(),
            It.IsAny<string>()));
        console.WriteLine(text + (averagePerformance / tryCount));
        this.averagePerformance = 0; // reset calculation
    }
```

```java
private bool addResponseTime(SwitchboardServicesOptionals p)
{
    this.averagePerformance += (long)p.ResponseTime;
    return true;
}
```

Dummy worker class for testing:

```java
public sealed class TimeConsumerElement : PipelineElement, ITimeMeasurementElement
{
    [references | 2 passing | 0 changes | 2 authors, 0 changes]
    public TimeConsumerElement()
    {
    }
```

```java
    [89 references | 0 changes | 0 authors, 0 changes]
    public override void Execute(CallContext callContext)
    {
        Thread.sleep(10);
    }
```

Page 67 of 71
Test without NetworkLatencyElement

```java
public void Test_48_execute_CallRequest_LogPerformanceWithoutNetworkLatency()
{
    setupLogger();
    var callContextMock = new Mock<CallContext>();
    var requestMock = new Mock<PipelineRequest>();
    var responseMock = new Mock<PipelineResponse>();
    requestMock.Setup(req => req.RequestPath).Returns("/requestPath");
    requestMock.Setup(req => req.RequestHeaders).Returns(new Dictionary<string, string>());
    responseMock.Setup(res => res.ResponseStatusCode).Returns(200);
    callContextMock.Setup(ctx => ctx.Logger).Returns(loggerMock.Object);
    callContextMock.Setup(ctx => ctx.Request).Returns(requestMock.Object);
    callContextMock.Setup(ctx => ctx.Response).Returns(responseMock.Object);
    var pipelineElement1 = new SimpleMeasurementElement();
    var pipelineElement2 = new TimeConsumerElement();
    pipelineElement1.Add(pipelineElement2);
    perform(pipelineElement1, callContextMock, loggerMock, "Average execution time without Network latency element: ");
}
```

Test with NetworkLatencyElement

```java
public void Test_49_execute_CallRequest_LogPerformanceWithNetworkLatency()
{
    setupLogger();
    var callContextMock = new Mock<CallContext>();
    var requestMock = new Mock<PipelineRequest>();
    var responseMock = new Mock<PipelineResponse>();
    requestMock.Setup(req => req.RequestPath).Returns("/requestPath");
    requestMock.Setup(req => req.RequestHeaders).Returns(new Dictionary<string, string>());
    responseMock.Setup(res => res.ResponseStatusCode).Returns(200);
    callContextMock.Setup(ctx => ctx.Logger).Returns(loggerMock.Object);
    callContextMock.Setup(ctx => ctx.Request).Returns(requestMock.Object);
    callContextMock.Setup(ctx => ctx.Response).Returns(responseMock.Object);
    var pipelineElement3 = new NetworkLatencyElement();
    var pipelineElement4 = new SimpleMeasurementElement();
    var pipelineElement5 = new TimeConsumerElement();
    pipelineElement3.Add(pipelineElement4).Add(pipelineElement5);
    perform(pipelineElement3, callContextMock, loggerMock, "Average execution time WITH Network latency element: ");
}
```

Results with 5000 executions of each method without dummy worker process:

<table>
<thead>
<tr>
<th>Test Name</th>
<th>Test Outcome</th>
<th>Standard Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test_2_execute_CallRequest_LogPerformanceWithoutNetworkLatency</td>
<td>Passed</td>
<td>Average execution time without Network latency element: 0</td>
</tr>
</tbody>
</table>

Results with 5000 executions of each method WITH dummy worker process:

<table>
<thead>
<tr>
<th>Test Name</th>
<th>Test Outcome</th>
<th>Standard Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test_3_execute_CallRequest_LogPerformanceWithNetworkLatency</td>
<td>Passed</td>
<td>Average execution time WITH Network latency element: 0</td>
</tr>
</tbody>
</table>
Results with 5000 executions with NetworkLatencyElement without dummy worker process:
15.3 Appendix 3 – Application PerformanceManagement Buyer's Guide and Reviews

This appendix shows snippets regarding the AppDynamics and DynaTrace APMs from the Application PerformanceManagement (APM) Buyer's Guide and Reviews March 2018 – created by IT Central Station

Top Application Performance Management (APM) Solutions

Over 261,186 professionals have used IT Central Station research. Here are the top Application Performance Management (APM) vendors based on product reviews, ratings, and comparisons. All reviews and ratings are from real users, validated by our triple authentication process.

Chart Key

<table>
<thead>
<tr>
<th></th>
<th>Views</th>
<th>Comparisons</th>
<th>Reviews</th>
<th>Followers</th>
<th>Average Rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of views</td>
<td>Number of times compared to another product</td>
<td>Total number of reviews on IT Central Station</td>
<td>Number of followers on IT Central Station</td>
<td>Average rating based on reviews</td>
<td></td>
</tr>
</tbody>
</table>

Bar length

The total ranking of a product, represented by the bar length, is based on a weighted aggregate score. The score is calculated as follows: The product with the highest count in each area gets the highest available score. (20 points for Reviews, 16 points for Views, Comparisons, and Followers.) Every other product gets assigned points based on its total in proportion to the #1 product in that area. For example, if a product has 80% of the number of reviews compared to the product with the most reviews then the product's score for reviews would be 20% (weighting factor) * 80% = 16. For Average Rating, the maximum score is 32 points awarded linearly based on our rating scale of 1-10. If a product has fewer than ten reviews, the point contribution for Average rating is reduced (one-third reduction in points for products with five to eight reviews, two-thirds reduction for products with fewer than five reviews). Reviews that are more than 24 months old, as well as those written by resellers, are completely excluded from the ranking algorithm.

1 Dynatrace

37,538 views, 21,312 comparisons, 175 reviews, 2,244 followers, 8.8 average rating

2 AppDynamics APM

41,533 views, 10,945 comparisons, 75 reviews, 2,912 followers, 8.3 average rating

Overview

Dynatrace has redefined how you monitor today’s digital ecosystems. AI-powered, full stack and completely automated, it’s the only solution that provides answers, not just data, based on deep insight into every user, every transaction, across every application. More than 8,000 customers use Dynatrace to optimize customer experiences, innovate faster and modernize IT operations with absolute confidence.

SAMPLE CUSTOMERS

Audi, Best Buy, LinkedIn, CISCO, Intuit, KRONOS, Scottrade, Wells Fargo, ULTA Beauty, Lenovo, Swarovski, Nike, Whirlpool, American Express
Overview

AppDynamics is the Application Intelligence company. With AppDynamics, enterprises have real-time insights into application performance, user performance and business performance so they can move faster in an increasingly sophisticated, software-driven world. AppDynamics’ integrated suite of applications is built on its innovative, enterprise-grade App IQ Platform that enables its customers to make faster decisions that enhance custom...

SAMPLE CUSTOMERS

Cisco, Sony, Nasdaq, Reserve Bank of New Zealand, Edmunds.com, Puma, Fox News, DirecTV, Pizza Hut, T-Systems, Cornell University, OpenTable, BITMARCK, Green Mountain Power, Care.com, Overstock, Paddy Power, eHarmony, Kraft, The Motley Fool, The Container Store, and more

See more customers