TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Vladimir Potašenkov 123687IAPB

LINNWORKS SYSTEM STATUS MONITORING

Bachelor's thesis

Supervisor: Deniss Kumlander
PhD
Vanemteadur

Tallinn 2018

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Vladimir Potašenkov 123687IAPB

LINNWORKS SÜSTEEMI STAATUSE SEIRE

bakalaureusetöö

Juhendaja: Deniss Kumlander

PhD
Senior researcher

Tallinn2018

Author’s declaration of originality

Annotatsioon

Käesoleva lõputöö eesmärgiks oli leida viis Linnworks informatsioonisüsteemide ja staatuste kujutamiseks infopaneeli kaudu ja rakendada selle prototüüpi. Rakendus oli loodud LinnSystems nõudmisel. Antud ettevõte tegeleb oma tarkvara arendamisega, mille eesmärk on lihtsustada ärimeeste tegevus e-kommertsi valdkonnas.

Antud projekti eesmärgiks oli süsteemiga seotud informatsiooni kogumine ja arusaadaval moel kujutamine suurel teleekraanil, mis võimaldaks töötajatel jälgida Linnworks rakenduste seisundit(serverite jälgimine, veadest teadaandmine jne). Informatsioon on esitatud graafikute ja tabelite kujul sõltuvalt valitud visualisatsiooni tööriistast.

Lõpptulemuseks oli leitud sobilik lahendus ja esitatud infopaneeli prototüüp, mida LinnSystems oma töös kasutab.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 65 leheküljel, 4 peatükki, 63 joonist, 2 tabelit.

Abstract

The main purpose of this thesis was to find an optimal solution for presenting information about Linnworks-related systems status on the dashboard and implementing a service for this data collection. This task was requested by LinnSystems. LinnSystems is a company that offers software for e-commerce merchants.

The aim of this project was to deliver and show information about the system in appropriate way on big display so that company’s workers would be able to get the general information about Linnworks system stability. Information is represented in a form of charts, tables and other elements available in chosen data vizualization tool.

As a result of this project, an appropriate solution has been found and working prototype has been presented. Dashboard is currently in use by LinnSystems.

The thesis is in English and contains 65 pages of text, 4 chapters, 63 figures, 2 tables.

Glossary of terms and abbreviations

E-commerce	is a process of selling and buying products or services over the internet [1]
Elasticsearch	is a free and open source search engine based on Apache Lucene. It provides a full-text search engine and works with JSON documents. [2]
Logstash	is a tool that converts received information to JSON format and saves it in Elasticsearch cluster [3]
InfluxDB	is an open-source database which is specialized on time series data. It is optimized for fast retrieval of time series data [4]
Kibana	is a data visualization plugin for Elasticsearch. It allows to visualize indexed data which is stored in Elasticsearch cluster [5]
Grafana	is an open source software for dashboards creation [6]
Jenkins	is an open source tool for building and deploying any project [7]
NoSQL	is a nonrelational databases for storage and retrieval of data [8]
Quartz.NET	is an open source job scheduling system [9]
WCF service	is a framework for creating service-oriented applications [23]
AWS CloudWatch	is a monitoring tool for applications hosted on Amazon Web Services [10]

Table of images

Table of tables

Table of Contents

Introduction

The main purpose of this thesis was to find optimal tools and provide a complete solution for general system health and status monitoring for LinnSystems company. Information about the system should be shown in appropriate way on TV screen so that company’s workers would be able to get the general information about Linnworks, system stability and customers’ satisfaction.

1. About the company

LinnSystems is a company that develops and supports software for e-commerce merchants who sell on online marketplaces such as Amazon and eBay. Main products of the company are Linnworks and Linnworks Desktop. They are used to simplify and automate everyday work of online sellers. Applications have similar functionality with the only difference is that Linnworks Desktop is an old version of the product and has to be installed on PC while Linnworks is a newer web-based product which is developed as a single page application.

LinnSystems has one office in Chichester, UK and one in Tallinn. Tallinn office has a development team and support representatives team. Support representatives team help customers to solve problems with software configuration. Communication is performed over phone calls, chats or tickets. Usually, customers choose a method of communication depending on the importance of their issue. New phone call, chat or ticket is automatically assigned to the first available support agent. It is important to know the actual load on the support team in order to plan their working schedule and provide qualified service to company's clients. Also, spikes of complaints may be caused by a bug in the software. In addition to listed communication forms, customers have an option to leave a feedback directly from the application interface. Feedback has a positive or negative type assigned by customer and message. Feedback from customers plays a significant role in planning further development process.

C# is used as a standard language for all internal development. For product applications company mainly use SQL databases and MongoDB. For internal needs the company has instances of Logstash and InfluxDB databases. Various exceptions and system logs from product applications are stored in ELK (Elasticsearch, Logstash, Kibana) stack. A very basic chart in Kibana is used to monitor an amount of exceptions. It is often used by Quality Assurance team after releases to ensure that everything went as planned and a new patch does not cause any issues.

2. Requirements

2.1 Company's requirements to the software

Company's main requirement was to create a dashboard that would allow company workers to check system status and general events connected with Linnworks at any given time on the display on the wall. Design has to be simple, intuitive and at the same time should contain enough information about different aspects.

The main goal of this project was to develop the idea, choose appropriate tools, design and to create dashboard for LinnSystems using data visualization tool that will show information and statistics about Linnworks system, related processes, and events. Some solution ways are dictated by the company and others are in the free choice.

Requirements:

- Create a dashboard with different information about the system (below are some ideas to choose from):
    - Status of servers – to show current load or/and status (operational or failed) of servers where Linnworks and related applications are hosted
    - Logs – to show an amount of unhandled exceptions in company's applications such as Linnworks.net and Linnworks desktop
    - Registration statistics – to show how many users registered in last month (preferably show estimated location on world map)
    - Feedback – to show a few examples of feedback from customers
    - Current online – to show an amount of currently active users in Linnworks and in Linnworks Desktop
    - Opened support tickets – to show an amount of support tickets raised by customers (grouped by their status: opened or resolved)
    - Active chats – to show an amount of chats with support representatives and amount of total available support agents
    - Active calls – to show an amount of calls from customers in support department

- Implement a windows service that will be able to regularly request needed for dashboard information from different database sources, aggregate it and pass to database which will be connected to the dashboard where information will be shown in appropriate format according to data type.
- Dashboard elements should be placed in such way that they will fit in big display
- Displayed information should be refreshed at least once in 5 minutes
- Choose TV that will be capable of displaying dashboards 24/7
- Set up TV to show dashboards in the office

Things to be considered:

- It is desired that during this project we use a combination of tools already in use by the company but at the same time, author has the freedom to choose other tools if needed
- Information is being kept in different sources and in different formats - so every type of data should be requested, handled and saved separately
- Information has to be aggregated, parsed to format required by data representation tool and passed to appropriate database

2.1.1 Functional requirements

- The panel should be able to present data depending on the nature of data with help of the following elements: grids, charts, labels, tables
- Dashboard views should refresh data using a predefined interval
- Information for the dashboard should be collected by a service
- It should be possible to specify the time range for data displayed on the dashboard

2.1.2 Nonfunctional requirements

- Dashboard elements should fit on one display
- Elements should be properly allocated on the dashboard, so it should be easy for workers to understand their meaning
- Dashboard should work on TV
- Service should be compatible with Windows Server
- Dashboard should be available only in company domain group
- Extensibility options should be available
- Service which collects information should execute itself automatically without interaction with human
- Dashboard should be available 24/7

3. Analysis

3.1 Possible solutions

In this paragraph several solution ideas are described and some of the possible approaches are be reviewed and the most suitable is chosen. Some conditions have been set by LinnSystems.

3.1.1 Collecting data

The dashboard should represent information which is originally being held in different formats and in different databases. That information should be collected into one place by certain tool. Company's desired option was a windows service as compatibility with windows server was required and it does not need much effort to support. Because of that, other options won't be considered.

3.1.2 Storing data

The main problem of storing data is that there are many different log types from different sources and they have to be classsified properly so that it would be easy to find them later.

3.1.2.1 Database choice

Taking into account the fact that data used by dashboard has different nature, originally it is stored in different types of databases. Modern data visualization tools are capable of working with multiple data sources. Therefore, the author is not limited to one database and can use multiple databases in this project.

SQL-based databases do not seem to be the best option for this project because of the nature of the data. Almost all required for dashboard information has a form of time series: logs, exceptions, servers' status metrics, etc. NoSQL databases are more oriented for such kind of data and there is no need to manually create separate tables for each type of information. In addition to that, there may appear scaling problems with SQL database over time. Also, some visualization tools such as Kibana cannot be easily integrated with SQL database.

Among NoSQL databases, MongoDB has been considered as an option as the company already has MongoDB instance running for one of the applications. MongoDB stores data in JSON format. This database was not designed for time series data. This is a document-oriented database and it is not possible to integrate MongoDB directly to data visualization tools such as Kibana, Grafana, and Graphite. Such disadvantage makes it pointless to use of this database for this dashboard project.

Various logs from Linnworks and other related applications are already being saved in Elasticsearch database. This is an established process with the help of ELK stack. In order not to re-save information about logs from one database to another it would be a wise decision to use already existing Elasticsearch database for logs storage. Moreover, it is possible to directly integrate Elasticsearch with some analytics and monitoring platforms such as Grafana, Graphite, and Kibana.

After further research, it appeared that InfluxDB might be a very good database choice. InfluxDB is a relatively new open source database with easy setup. It was designed for time series data, such as logs and statistics. This database is capable of performing fast real-time analysis on a large volume of data. InfluxDB's strong side is the speed of write and read operations. Based on benchmark results [11], InfluxDB has much faster write speed compared to Elasticsearch and a bit slower read speed. Also, it is said that InfluxDB has better performance 'out-of-the-box'. InfluxDB can be integrated with visualization tools such as Grafana. On DB-Engines ranking [12] InfluxDB is on the 1st place.
OpenTSDB database has been considered as a pretender. OpenTSDB is a time series database which main purpose is to be used for dashboard visualizations. But based on benchmarks [13] it is not outperforming InfluxDB in any way. Author of this thesis has not found any major advantages of other NoSQL databases over InfluxDB and as company was able to quickly provide instance of this database, it was decided to choose InfluxDB. In addition to that, manager had a desire to hold on to already used software rather than jumping to something completely new and unknown if there is no justified reason for that.

As a result, Elasticsearch and InfluxDB databases will be used to store data required for dashboard. Elasticsearch will be used to fetch information about unhandled exceptions and InfluxDB will be used to store other information (specific information for dashboard project).

3.1.3 Visualizing data

A third-party tool will be used for data visualization. The main pretenders are: Kibana (Figure 1), Zoomdata and Grafana (Figure 2). Zoomdata is not free so author decided to look into sKibana and Grafana.

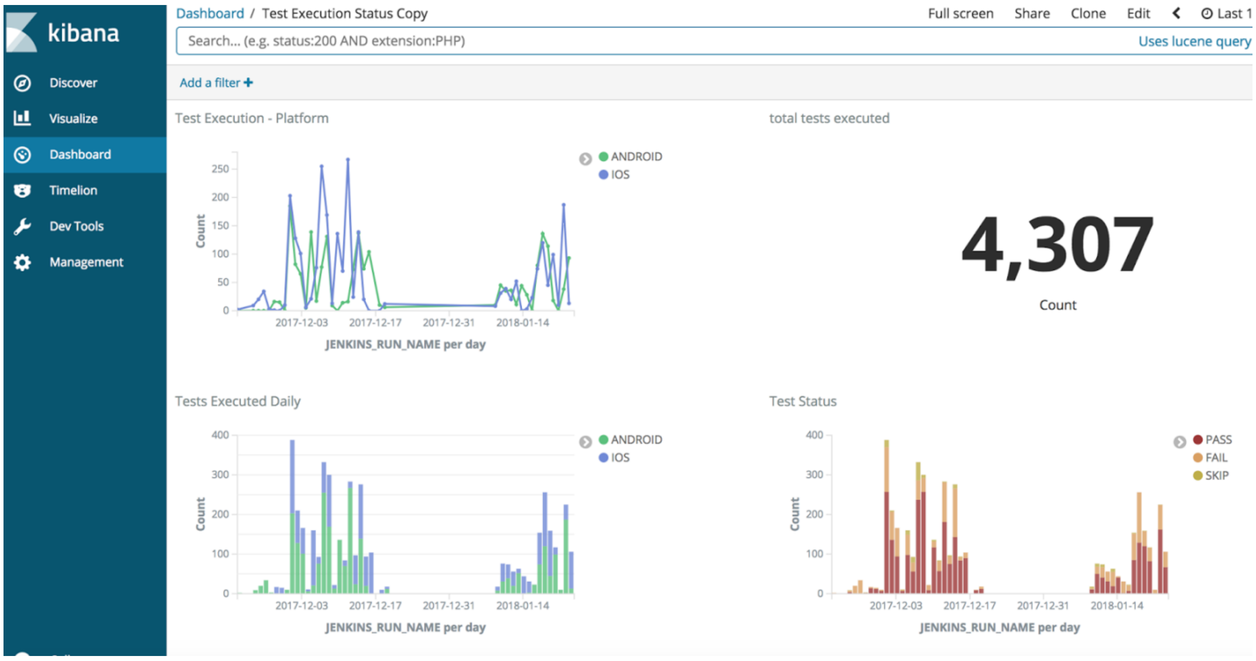
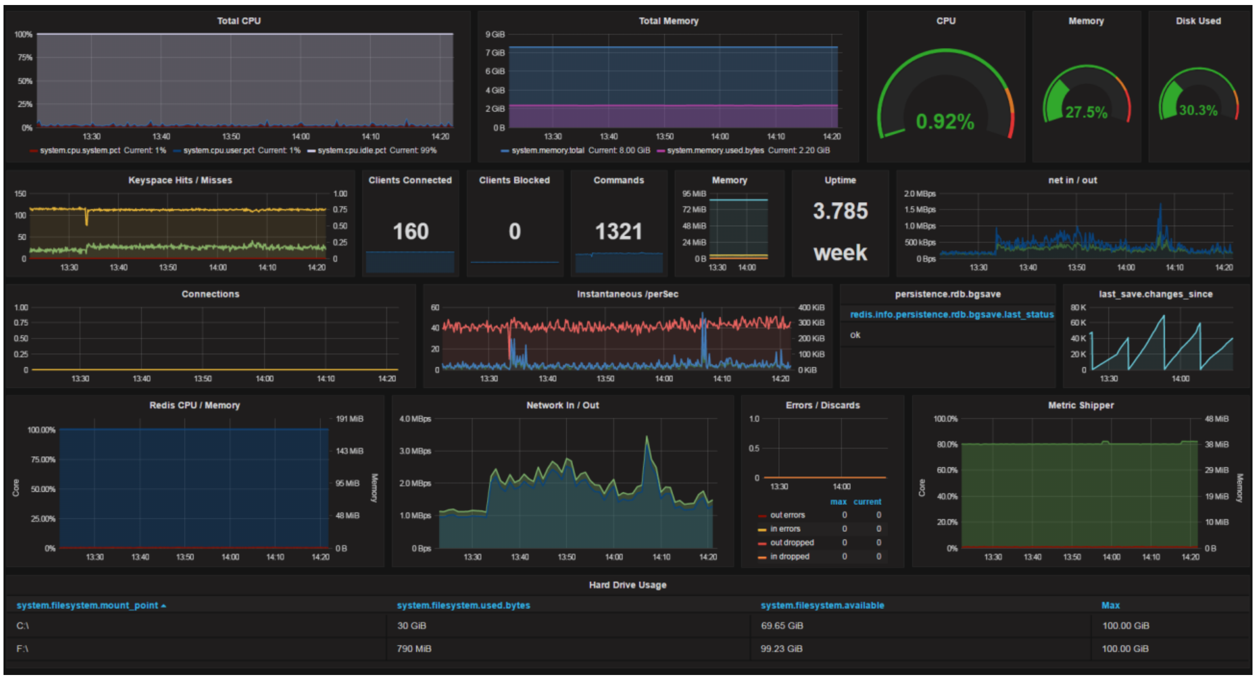


Figure 1. Kibana dashboard



system.filesystem.mount_point	system.filesystem.used.bytes	system.filesystem.available	Max
C:\	30 GiB	69.65 GiB	100.00 GiB
F:\	790 MiB	99.23 GiB	100.00 GiB

Figure 2. Grafana dashboard

Although the very first prototype of dashboard has been made with Kibana visualization tools because it has already been used by company and there was a manager's desire to have a standard solution, after some investigations it was decided to use Grafana. It appeared to be the best option, because Kibana is more specialized on logs, while Grafana has better support for other design elements like displaying current status (On/Off), amount of something etc. Also Grafana has more attractive design which is quite important, as dashboard will be displayed on SmartTV in the office.

In addition to that, Grafana can be integrated with Elasticsearch, InfluxDB, with Amazon CloudWatch [10] (company has plans to use AWS CloudWatch for server monitoring, so possibility to integrate it with Grafana is a big advantage) and other sources.

Recently there was a release of Grafana 5 and that latest version is used in the project as it has advanced settings for positioning elements on dashboard (Figure 3) which is also an advantage over Kibana which has limited possibilities for elements positioning.

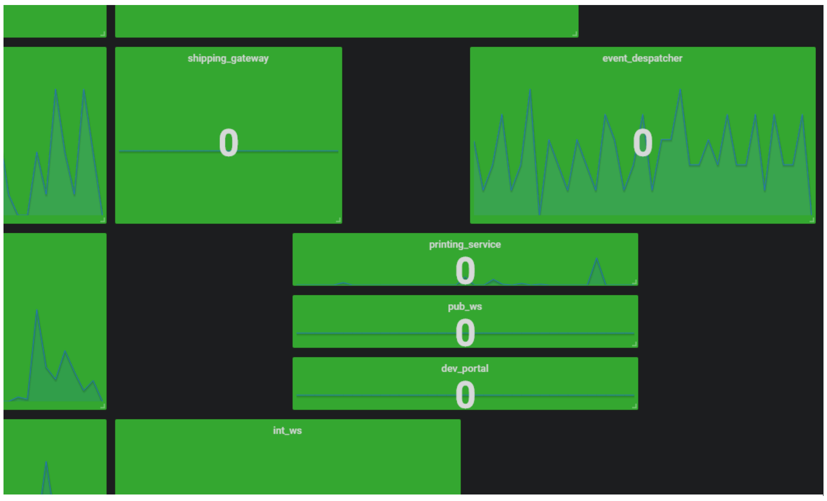


Figure 3. Grafana elements positioning

3.2 Development tools

3.2.1 Programming language

Almost all software in Linn Systems is written in C# using ASP.NET framework. One of the requirements was to use C# for this project so that it will be easier for developers to maintain it in the future.

3.2.2 Third-party libraries

Some of the third-party libraries are used in data-collecting service to simplify development process.

NewtonsoftJSON - Popular high-performance JSON framework for .NET. [14] It is used to convert data into JSON format. This library is used in other company's projects and in order to adhere to the standard it was decided not to use other JSON libraries.

Selenium WebDriver and Chrome driver - Selenium is a suite of tools designed for automating web browsers [15]. This tool is already used for automated tests in LinnSystems. This tool completely meets our needs in the scope of this project and there is no need to search for a better option.

Ninject - Open source dependency injector for .NET [16]. This is a standard library in LinnSystems company.

Quartz.NET - is a pure .NET library written in C# and is a port of very popular open source Java job scheduling framework, Quartz [17]. It is used to schedule recurring tasks in windows service.

InfluxDB.NET – library for InfluxDB database.

4. Implementation details

The whole project consists of three main parts:

- Windows Service – The purpose of the service is to collect data, convert and send it to database
- Dashboard – Displays data in a form of visual elements (tables, graphs, charts, etc)
- Int_ws_utils – Internal LinnSystem's services that have permission to get information from SQL database. It is used by Windows Service to get information about customers' feedback and Linnworks online.

More detailed information is shown on Figure 4.

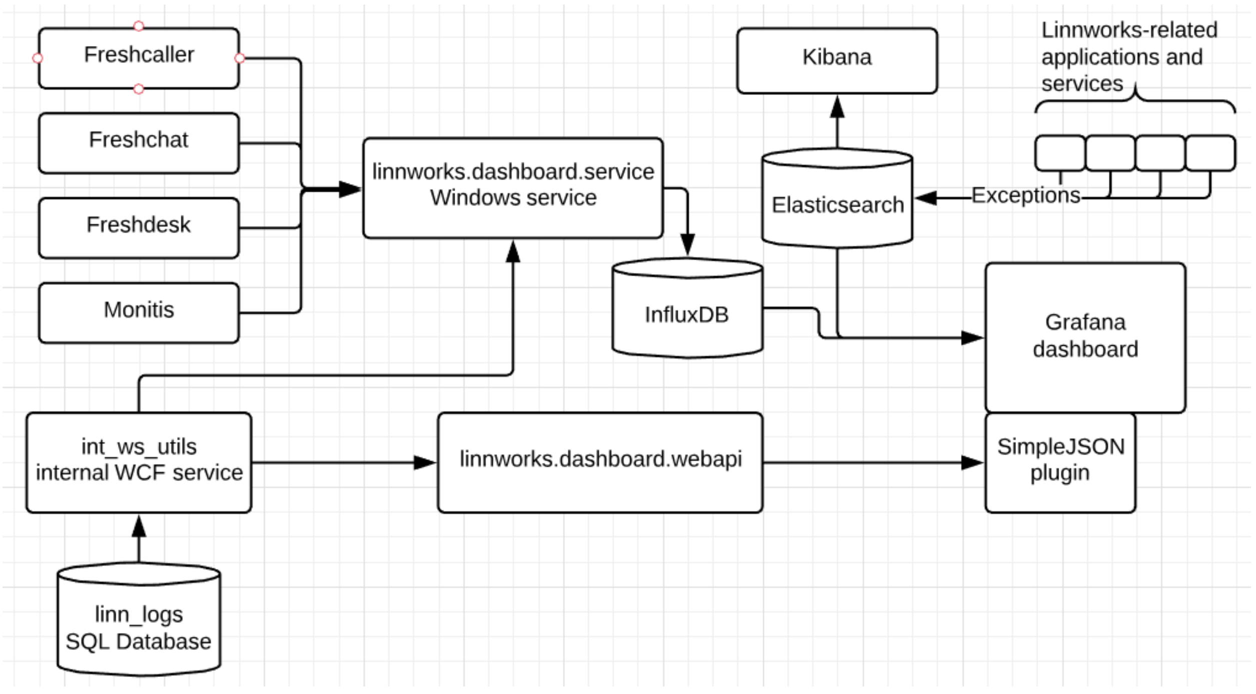

Freshcaller
Freshchat
Freshdesk
Monitis
linnworks.dashboard.service Windows service
Kibana
Linnworks-related applications and services
Elasticsearch
Exceptions
InfluxDB
Grafana dashboard
int_ws_utils internal WCF service
linnworks.dashboard.webapi
SimpleJSON plugin
linn_logs SQL Database

Figure 4. General structure

4.1 Windows service details

For collecting statistics, system logs, feedback and other information types it is planned to create a windows service that will regularly request information, handle it and forward to database. It is easy to set up windows service on windows server (one of the requirements) and it should not cause additional problems with installation.

The main purpose of the windows service is to collect information from different sources and save it. As information is taken from differrent databases and is presented in different formats, it should be aggregated and converted to format which is apporpriate for saving. In order to achieve that, service consists of different job types. System is designed to run only one job simultaneously. Purpose of each job is to send request to only one specific data source, individually and correctly handle response, convert received information to appropriate format and pass it to InfluxDB database. General data-collecting windows service's process is as follows (Figure 5):

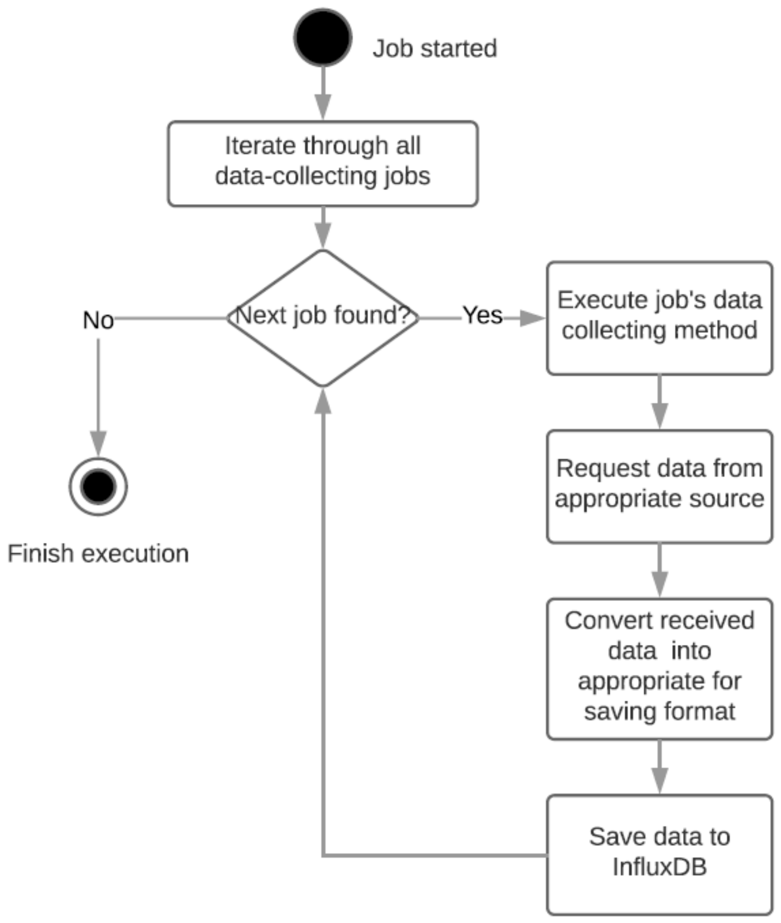

Job started
Iterate through all data-collecting jobs
Next job found?
No
Yes
Execute job's data collecting method
Request data from appropriate source
Convert received data into appropriate for saving format
Save data to InfluxDB
Finish execution

Figure 5. Data-collecting job workflow

Main purposes of the service:

- Collect data from different sources (from SQL and NoSQL databases, freshdesk, Jira using API, etc.)
- Aggregate data from received format to appropriate for storing
- Send data to database that will be connected with dashboard
- Execute itself on a regular basis

In order to get all available jobs, adapter searches for all classes inherited from *BaseDataSource* and then executes *GetData()* method on them (Figure 6).

```
public static void Run(DateTime startTime)
{
      var assembly = Assembly.GetExecutingAssembly();
      var baseType = typeof(BaseDataSource);
      var types = assembly.GetTypes().Where(t =>
baseType.IsAssignableFrom(t) && t != baseType);

      using (var kernel = new
StandardKernel(SolutionWideNinjectBindings.GetBindings()))
      {
            var settings = kernel.Get<AppSettings>();

            var connectionInfo = new
      BaseConnectionInfo(settings.LinnStatsDatabaseServer,
      settings.LinnStatsDatabaseName, settings.LinnStatsUserId,
      settings.LinnStatsPassword);

            foreach (var type in types)
            {
                  var dataSource = kernel.Get(type) as BaseDataSource;

                  // influxdb
                  var influxDb = dataSource as IInfluxDbDataSource;
                  if (influxDb != null)
                  {
                        try
                        {
                              var data = influxDb.GetData(startTime);
```

Figure 6. Get available jobs

4.1.1 Windows service structure

Windows service solution consists of two main projects (Figure 7):

- linnworks.dashboard.logic
- linnworks.dashboard.service

And some code is being held in LinnSystems internal projects such as:

- linnworks.influxdb
- linnworks.logger
- linnworks.services

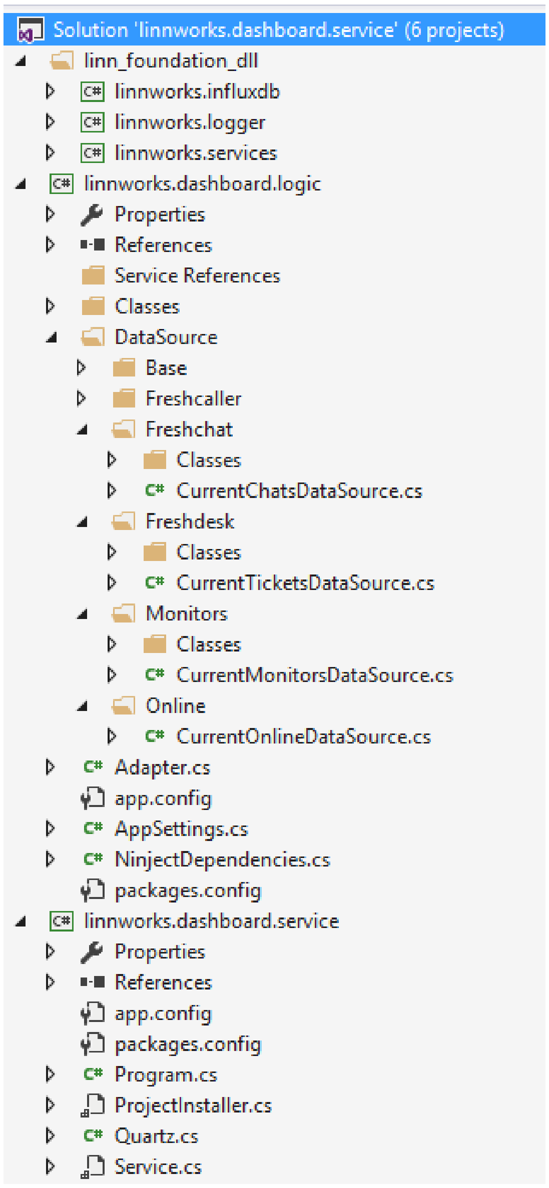
linnworks.dashboard.logic – all logic for jobs is being kept in separate subfolders in DataSource folder. Also Adapter is located in this project, which controls job execution process: collects available jobs, executes them and saves received data to appropriate database.

linnworks.dashboard.service – contains logic related to scheduling process and service startup configurations.

linnworks.influxdb – is an internal project that keeps settings and configurations to work with InfluxDB and contains data structure for logs. This project has been created by LinnSystems database administrator and has been updated by the author of this thesis.

linnworks.logger – is an internal project that is used to save logs and exceptions among all LinnSystems' products and applications. This project has been updated by the author of this thesis in order to work with new logs structure.

linnworks.services – is an internal LinnSystems' project that is used to provide clients for internal WCF services. This project has been updated by the author of this thesis in order to work with Dashboard service.



```
Solution 'linnworks.dashboard.service' (6 projects)
◢ linn_foundation_dll
    ▷ linnworks.influxdb
    ▷ linnworks.logger
    ▷ linnworks.services
◢ linnworks.dashboard.logic
    ▷ Properties
    ▷ References
      Service References
    ▷ Classes
    ◢ DataSource
        ▷ Base
        ▷ Freshcaller
        ◢ Freshchat
            ▷ Classes
            ▷ CurrentChatsDataSource.cs
        ◢ Freshdesk
            ▷ Classes
            ▷ CurrentTicketsDataSource.cs
        ◢ Monitors
            ▷ Classes
            ▷ CurrentMonitorsDataSource.cs
        ◢ Online
            ▷ CurrentOnlineDataSource.cs
    ▷ Adapter.cs
      app.config
    ▷ AppSettings.cs
    ▷ NinjectDependencies.cs
      packages.config
◢ linnworks.dashboard.service
    ▷ Properties
    ▷ References
      app.config
      packages.config
    ▷ Program.cs
    ▷ ProjectInstaller.cs
    ▷ Quartz.cs
    ▷ Service.cs
```

Figure 7. Project structure

4.1.2 Scheduling

A scheduling has to be implemented to set up service execution on a regular basis. It this project Quartz.NET [17] library was used. Quartz is a jobs scheduling library that is used to easily setup scheduling for different jobs. Although this tool does not have advanced settings for unusual tasks, nor any job history, alerts or error handling, this tool fully meets our requirements, therefore other options won't be reviewed. Windows service is configured using Quartz.NET to run scheduled tasks every predefined interval of time. When execution begins, adapter collects all available job types and starts to iterate through them one by one. When all jobs finish execution, process completes until next scheduled run.

4.1.3 Startup configuration

Startup configurations are located in *linnworks.dashboard.service* project. For example, in *Service.cs* class there is logic which determines if application runs locally or on windows server as a windows service. Based on startup option is chosen: to start as a service or as a console application. [18] In order to make decision the author relies on `Environment.UserInteractive` value as Windows Service by default is not allowed to be interactive (Figure 8).

```
static void Main()
{
      var service = new dashboard.service.Service();
      if (Environment.UserInteractive)
      {
            service.RunAsConsole(null);
            }
            else
            {
                  ServiceBase[] ServicesToRun;
                  ServicesToRun = new ServiceBase[]
                  {
                        new Service()
                  };

                  ServiceBase.Run(ServicesToRun);
            }
      }
```

Figure 8. Startup options

4.2 Dashboard interface

Dashboard interface elements' style mainly depends on type of information that should be presented. First prototype (Figure 9) has been discussed with manager and some of the team members. Initially it was hard to understand what do we need to achieve and how it should look like. As the author and the manager did not have a clear vision, some most interesting elements have been chosen without justification and development started.

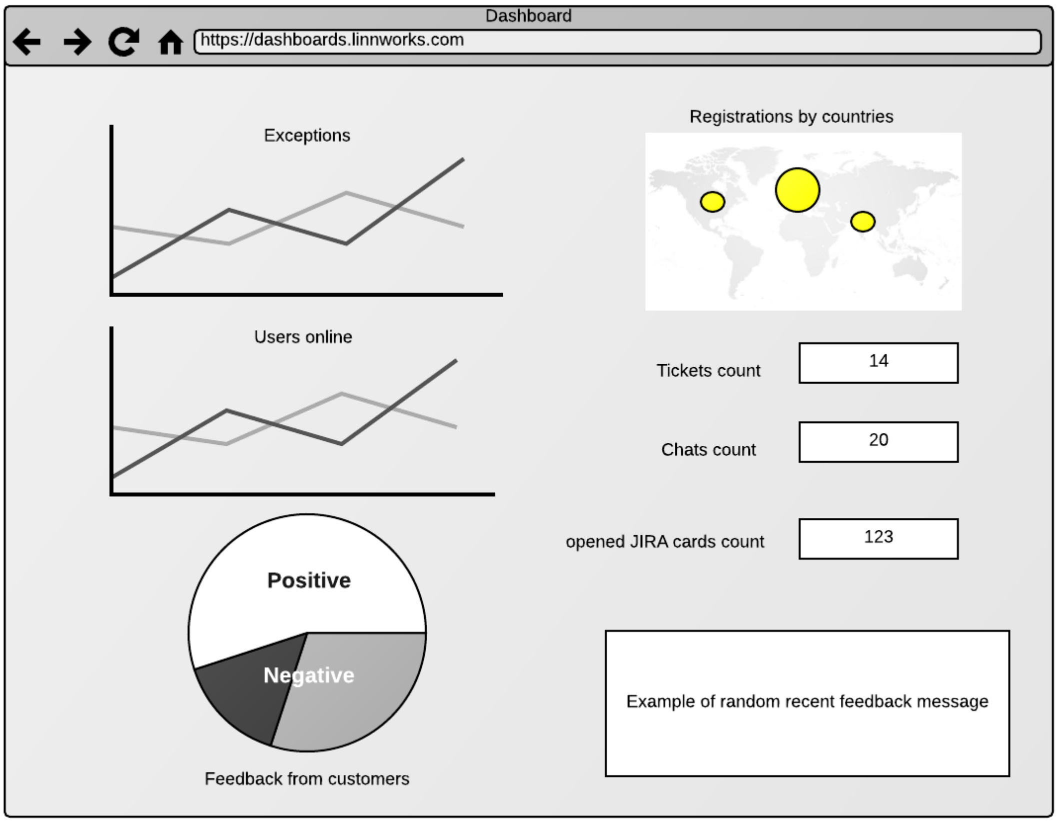


Figure 9. Dashboard 1st prototype

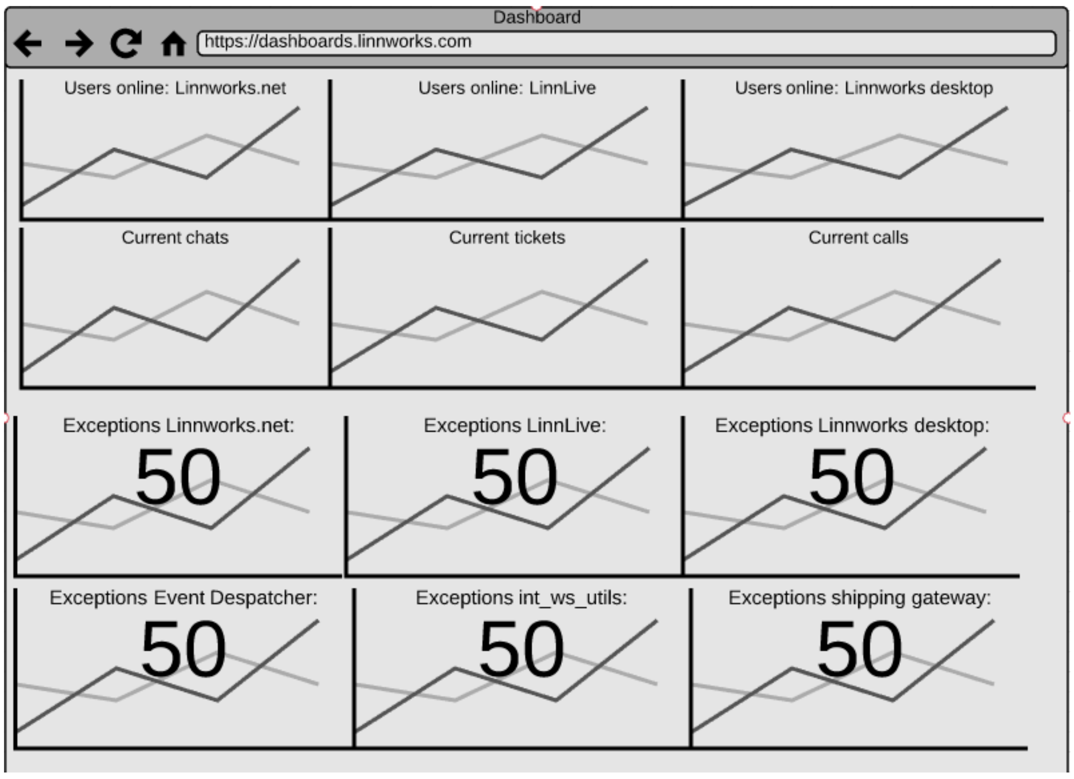
For visualizing exceptions level LineChart diagram would be the best choice as it allows us to track growth or decrease an amount of unhandled exceptions over time. Exceptions level will greatly help to notice problems in the system after weekly releases or after hotfixes.

In order to track pressure on the support team, it would be useful to show current amount of opened tickets, calls and chats from customers. If amount of them greatly increases in a short period of time, it may mean that there is a problem in the software.

World map element may represent amount of registered customers by their country. It is easier to perceive such type of information from world map rather than from table.

Customers feedback (positive and negative) about Linnworks product could be represented in pie chart with a randomly selected message that change periodically to provide more interactive feedback because there is not enough space to show all feedback at once. Amount of opened and closed JIRA cards might be shown in numbers or in a line chart in order to stimulate developers to increase quality of their work. There is also a line chart displaying amount of active users over time.

In the beginning this prototype has been accepted by team members and displayed information seemed useful. However, when development started and the first semi-working copy has been launched, it appeared that dashboard is not giving the team desired information. After further discussion with the team and manager, author came to a decision that a dashboard should mostly contain information related to system status rather than displaying customers' feedback, opened Jira cards and registration locations. Based on received from colleagues feedback about the dashboard new prototype has been presented to the workgroup (Figure 10):


Dashboard
https://dashboards.linnworks.com
Users online: Linnworks.net
Users online: LinnLive
Users online: Linnworks desktop
Current chats
Current tickets
Current calls
Exceptions Linnworks.net:
50
Exceptions LinnLive:
50
Exceptions Linnworks desktop:
50
Exceptions Event Despatcher:
50
Exceptions int_ws_utils:
50
Exceptions shipping gateway:
50
Figure 10. Dashboard 2nd prototype

In the second prototype it was decided to allocate more space for information that is more important to control in real time: amount of active users, amount of unhandled exceptions and load measurement on support department. After some time of development, author presented second working copy to team members and the manager. It was displayed on TV on the office wall. For a few days we liked new dashboard but then team members started to notice that there is information only about exceptions and not enough information regarding other aspects of the system. Often it is not enough to rely only on the amount of exceptions. Moreover, dashboard consists of many similar elements and it is hard to focus on information, hard to understand if amount of exceptions is high or everything is in its normal state. The idea of the dashboard was to allow workers in the company to check if system has problems or not by taking a quick look at the monitor when bypassing it.

After reviewing new feedback from team members and the manager, the next idea that the author came up with was to remove linear graphs and exception numbers when everything is running smoothly in order to get rid of unimportant information. Each project or important part of a system which requires monitoring will have its own "health box" (Figure 11). When everything is in normal condition, graph will have green background and no information displayed on it. Elements will have its own threshold values for 'Warning' and 'Critical' conditions. Those values are set based on average numbers of exceptions during a few weeks and they are separate for each "health box". When some metric reaches its 'Warning' or 'Critical' condition, graph will be marked in yellow or red color and critical metric name with value will be displayed on the graph. Color system makes it easy for eyes to focus only on important elements and small part of displayed information makes it easy to read and get snapshot of system status in a few seconds.

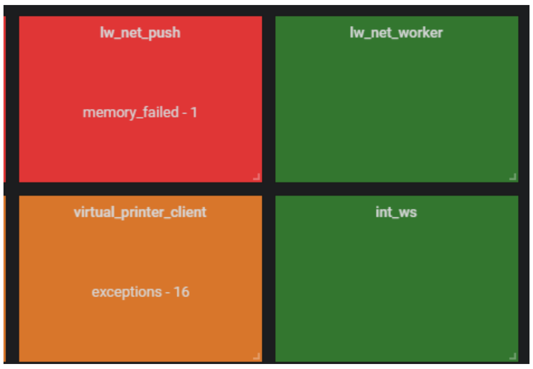


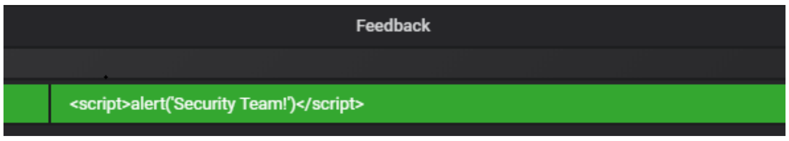
Figure 11. Health boxes

Initially it was planned to display only amount of exceptions on these boxes but after further discussion with colleagues, more interesting idea was born – the idea to monitor complex metrics. In order to implement this idea it was decided to collect monitoring logs from Monitis API. Monitis [19] is a tool that allows monitoring of web infrastructure. Monitis monitors include Memory, CPU and Drive load, bandwidth and availability metrics. As a result, each "health box" has been connected with 6 metrics:

- Exceptions
- Memory
- CPU
- Drive
- Bandwidth
- Availability

Monitoring of different metrics allows to control health status of different part of the system and if somewhere appears to be a problem it is easy to identify which projects or parts of the system are affected and what exactly is failing. This does not mean that previous version of dashboard was completely useless. Some elements have been taken from it and moved to a separate dashboard contaning only charts with exceptions. This page will be shown on TV display during slides rotation configured in RiseVision.

Feedback, which has been added on the final stage of development, is displayed as a table. Table has one column with text message. Element will change colour depending on type of feedback – green for positive and orange for negative (Figure 12).



Feedback

<script>alert('Security Team!')</script>

Figure 12. Feedback colouring

At the time of writing this project there are two working dashboards rotating on TV screen in LinnSystems office (Figure 13 and Figure 14):

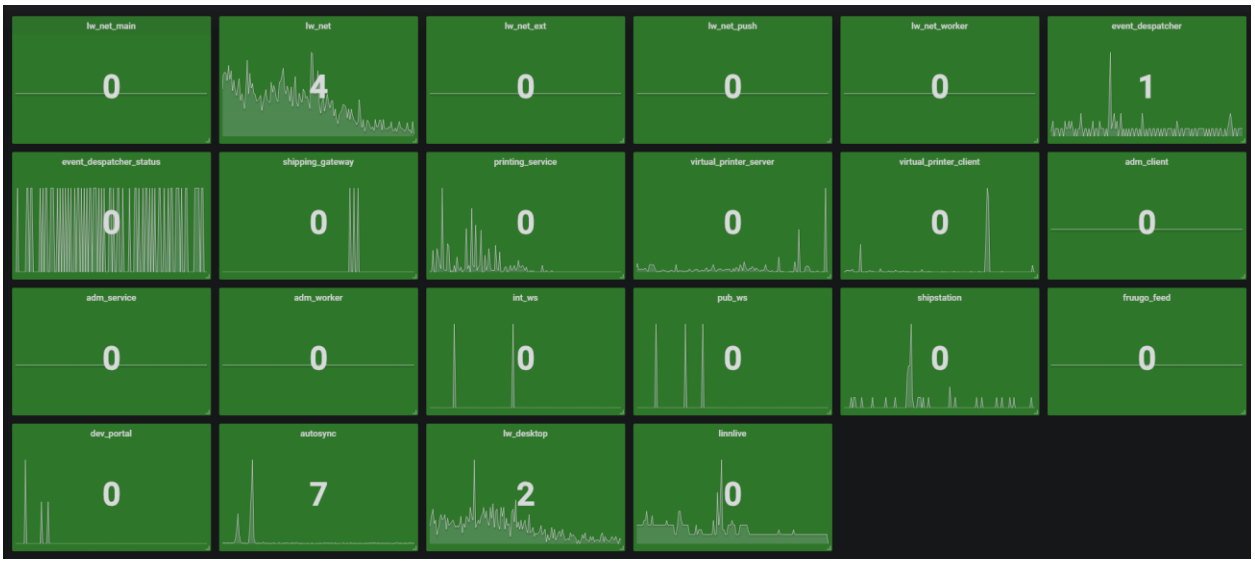


Figure 13. Dashboard 1

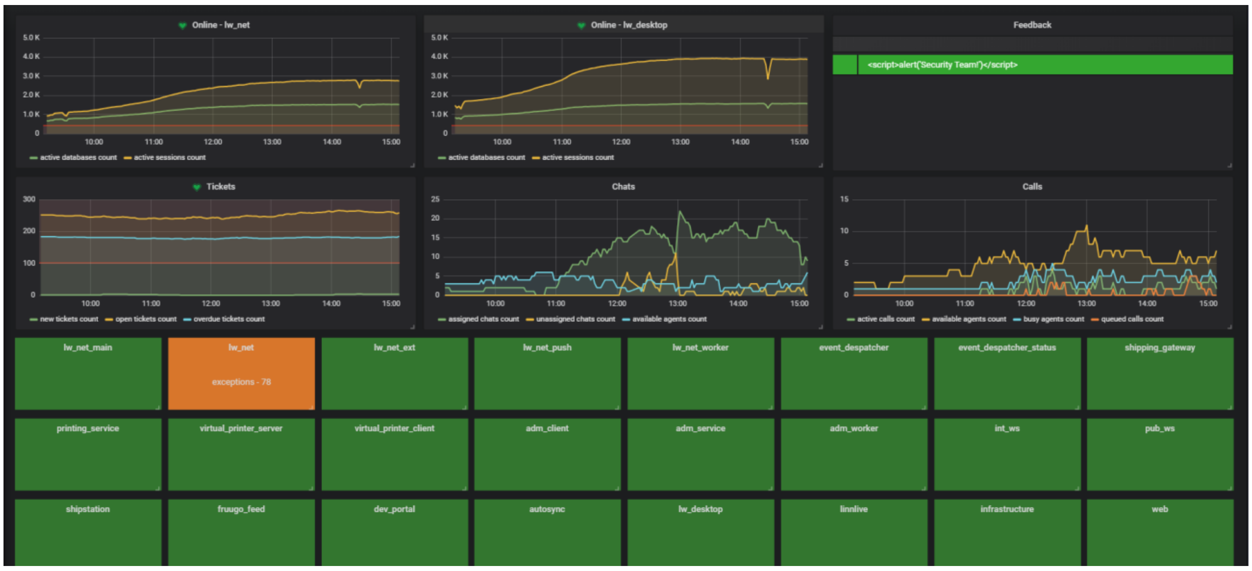


Figure 14. Dashboard 2

4.3 Data collecting

There are described data collecting jobs. They have different logic and different approaches for requesting data. All jobs have one common base class *BaseDataSource* which contains data source name and *appSettings* with authorization keys and connection strings. The main purpose of this class is to unite classes so that Adapter can collect all derived classes in order to get all jobs for execution. This idea allows us to easily add new jobs in the future – we only have to inherit *BaseDataSource* class and job collector will be able to execute them.

4.3.1 Current online

Amount of currently active users among LinnSystems' products is stored in SQL database with restricted access. Company's internal int_ws_utils service will be used to get this data. A new method has been added to this service as a part or this project in order to receive structured information about online users. New stored procedure and table have been added to SQL database. Code will be described in details below as most logic for retrieving amount of online users is located in internal projects and therefore is not included in archive with project's code. Information about active sessions is being kept in *online.sessions_current* table (Table 1).

Column name	**Column type**	**Description**
UserDatabase	Nvarchar(255)	User's database name
Time	Datetime	Time of activity
ServiceName	Nvarchar(255)	Name of application
Username	Varchar (255)	Customer's username

Table 1. sessions_current structure

Method *GetAppData* (Figure 15) was added to *int_ws_utils* project in *Dashboard.svc*. This method executes *[online].[get_total]* (Figure 16) stored procedure and converts received information to *List<AppData>*.

```
public static List<AppData> GetAppData() {
      var result = new List<AppData>();

      using (var conn = new SqlConnection(
      Helpers.SettingHelper.linn_logs_connectionString)) {
            conn.Open();

            using (var cmd = new SqlCommand(@"online.get_total",
      conn)) {
                  cmd.Parameters.Add(
                  new                             SqlParameter("@time",
                  DateTime.UtcNow.AddMinutes(-7)));
```

Figure 15. Call of the procedure

Procedure online.get_total (Figure 16) is executed with parameter *@time* set as current time minus 7 minutes in order to get an amount of active sessions and databases for the last 7 minutes.

```
IF NOT EXISTS (SELECT * FROM sys.objects WHERE object_id =
OBJECT_ID(N'[online].[get_total]') AND type in (N'P', N'PC'))
BEGIN
EXEC dbo.sp_executesql @statement = N'CREATE PROCEDURE
[online].[get_total] AS RAISERROR(''[online].[get_total] is not yet
defined'', 16, 1);
                         '
END
GO
ALTER PROCEDURE [online].[get_total]
    @time datetime
AS
BEGIN
      /* online sessions */
      SELECT
            ServiceName,
            COUNT(*) as SessionsCount
      FROM [linn_logs].[online].[sessions_current]
      WHERE [Time] > @time AND Username IS NOT NULL
      GROUP BY ServiceName

      /* online databases */
      SELECT
            ServiceName,
            Count(*) as DatabasesCount
      FROM
      (
            SELECT
                  ServiceName,
                  UserDatabase,
                  COUNT(*) as OnlineCount
            FROM [linn_logs].[online].[sessions_current]
            WHERE [Time] > @time AND Username IS NOT NULL
            GROUP BY ServiceName, UserDatabase
      ) AS t
      GROUP BY t.ServiceName
END
GO
```

Figure 16. Stored procedure get_total

Received information is being converted to list of *AppData* classes (Figure 17):

```
public class AppData
{
        public string AppName;

        public int DatabasesOnline;
        public int SessionsOnline;
}
```

Figure 17. AppData class structure

Class contains application name and amount of active users. On windows service side *linnworks.services* project is used to communicate with WCF service, as one of the manager's desires was to follow company's standards. This is an already existing project implemented by LinnSystems which contains service references to different internal services and has a service factory which provides service clients with correct urls and keys. New service reference has been added to *linnworks.services* project (Figure 18).

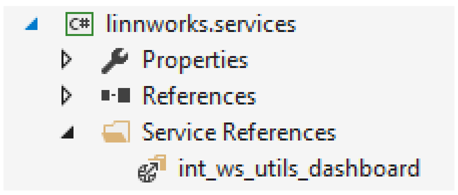


Figure 18. Service reference

Then in *ServicesFactory* creation of client is defined (Figure 19).

```
public IDashboardClient GetDashboardClient()
{
        var client = new int_ws_utils_dashboard.DashboardClient();
        client.Endpoint.Address = new
System.ServiceModel.EndpointAddress(
                settings.int_ws_utils_dashboard_url);
         var binding = new System.ServiceModel.BasicHttpBinding();
         binding.Security.Mode =
System.ServiceModel.BasicHttpSecurityMode.Transport;
         client.Endpoint.Binding = binding;

         return client;
}
```

Figure 19. Define client creation

Then data is requested and received information is returned in appropriate format (Figure 20).

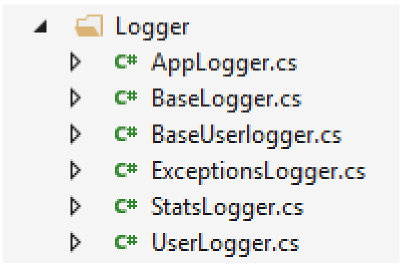
```
public List<InfluxDB.Net.Base.BaseDataStructure> GetData(DateTime
time)
{
      var list = new List<InfluxDB.Net.Base.BaseDataStructure>();
      var appData = new services.int_ws_utils_dashboard.AppData[] { };
      using (var client = servicesFactory.GetDashboardClient())
      {
            appData = client.GetAppData();
      }
      foreach (var app in appData)
      {
            list.Add(new
      InfluxDB.Net.DataStructure.CurrentOnline(time, app.AppName,
      app.DatabasesOnline, app.SessionsOnline));
      }

      return list;
}
```

Figure 20. GetData method

4.3.2 Exceptions

In windows service there is no need to create a separate job for collecting exceptions as they are already saved in Elasticsearch database. Grafana supports direct integration with Elasticsearch. Queries for displaying exceptions will be described in Visualizing section.

Almost all of the company's applications and related services (linnworks.net, linnworks desktop, event despatcher, shipping gateway, int_ws_utils, int_ws_channels and others) have global exceptions handler that passes all occurred exceptions using *linnworks.logger* directly to Elasticsearch. Elasticsearch is used as a source for exceptions on Grafana dashboard. Logger separately handles different exception and log types (Figure 21).



```
Logger
    AppLogger.cs
    BaseLogger.cs
    BaseUserlogger.cs
    ExceptionsLogger.cs
    StatsLogger.cs
    UserLogger.cs
```

Figure 21. Logger types

ExceptionsLogger is used for unhandled exceptions. Exception message and stack trace are passed to Elasticsearch over TCP.

4.3.3 Chat system for monitoring

When customers experience problems in their everyday use of Linnworks they may address their issues to support department using support chat system. Company uses Freshchat as a chat system. It would be useful to show on the dashboard amount of available support representatives, assigned chats and unassigned (queued) chats. This information may reflect problems with software if amount of opened chats greatly increase in a short period of time. Unfortunately, Freshchat’s service has a very poor API possibilities at the moment and it is impossible to get all needed information using this method.

Freshdesk support team provided some examples. It is possible to send requests to this endpoint:

https://web.freshchat.com/app/dashboard/basic/conversation

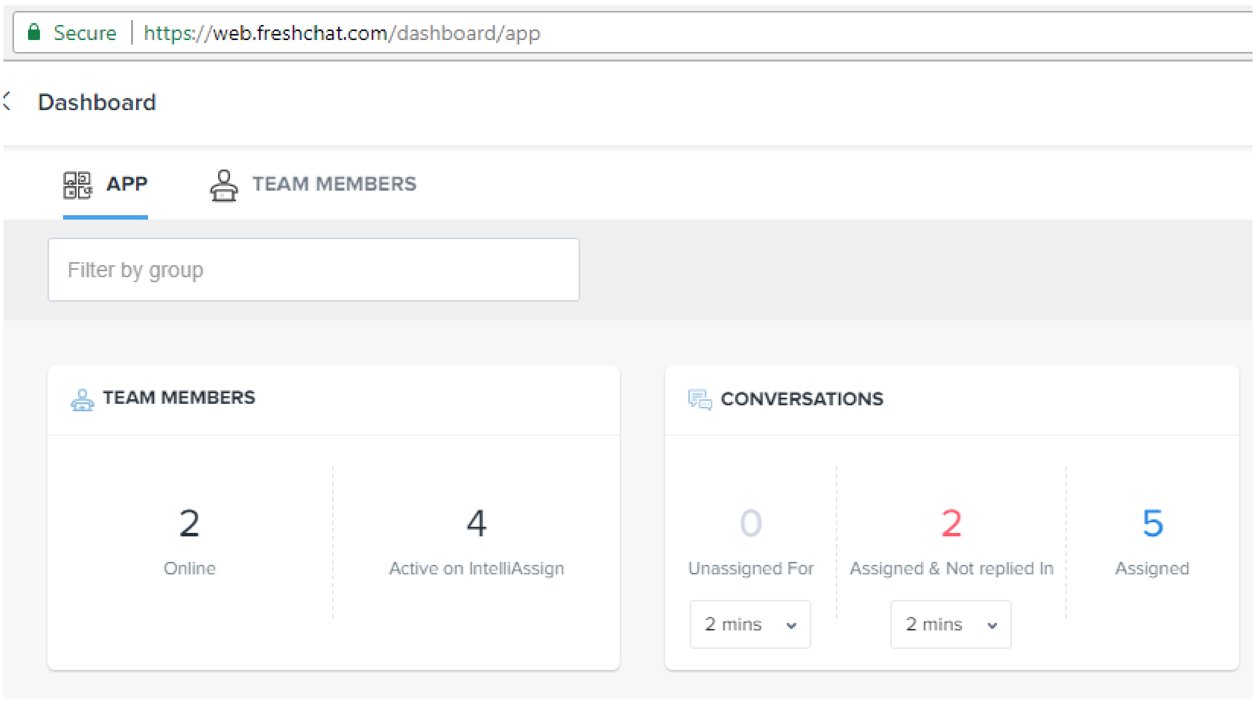
using X-HL-AUTH-TOKEN and response in the following format will be received:

{"conversationUnassigned":[{"time":2,"count":23}],"conversationNotReplied":[{"time":2,"count":1}],"activeAgent":[42994679111689,43009522368512,42993793548289,43009643958279,42994502606856,43011007320064]}

The problem is that this endpoint expects not more than 1 request every 5 minutes. Moreover, response does not contain amount of assigned chats. It only contains amount of available agents and unassigned or not replied chats. For this project it is important to see how many chats are already assigned as it reflects actual load on support department.

As a solution, author decided to use Selenium WebDriver and Chrome WebDriver libraries. They are often used by LinnSystems testing team to write automated UI-tests. In this project those tools will be used to login into freshchat from browser using username and password, locate needed elements on web page and fetch values from them.

On picture (Figure 22) there is displayed a part of Freschat page with displayed information about current conversations and amount of available support agents in chat system. This information is taken from page and is saved into database.


Secure | https://web.freshchat.com/dashboard/app

Dashboard

APP TEAM MEMBERS

Filter by group

TEAM MEMBERS

2 Online

4 Active on IntelliAssign

CONVERSATIONS

0 Unassigned For 2 mins

2 Assigned & Not replied In 2 mins

5 Assigned

Figure 22. Freshchat main dashboard

To get information from web page we will use Chrome driver (Figure 23).

```
//Options
ChromeOptions options = new ChromeOptions();
options.AddArguments("--disable-extensions");
options.AddArguments("--start-maximized");
options.ToCapabilities();

ChromeDriverService service =
ChromeDriverService.CreateDefaultService(appSettings.ChromeDriverResou
rcesPath);

IWebDriver driver = new ChromeDriver(service, options);
```

Figure 23. Chrome driver settings

ChromeDriver is placed in /*resources* folder. When deployed, this path will be different so has to be replaced by Jenkins during deployment. Search of elements will be performed on the page by element ids, class names or by text (Figure 24).

```
var ele = driver.FindElement(By.ClassName("select2-input"));
var js = (IJavaScriptExecutor)driver;

js.ExecuteScript("arguments[0].click();", ele);
driver.FindElements(By.ClassName("select2-result-label")).
      FirstOrDefault(x => x.Text.Equals("tech support",
      StringComparison.InvariantCultureIgnoreCase)).Click();
```

Figure 24. Locate elements on page

In the end we get the following information about chats:

- Amount of active agents
- Amount of assigned chats
- Amount of unassigned chats

4.3.4 Current calls

Freshcaller system is used for calls management in support department. Requests to web page are performed to retrieve information about current calls. First call will be used for authorization only. We will call https://help.linnworks.com/support/login and provide data with authorization keys in the following format:

```
utf8=%E2%9C%93&authenticity_token={0}&user_session%5Bemail%5D={1}&user_session%5Bpassword%5D={2}
```

Request will be encoded (Figure 25).

```
var data = Encoding.ASCII.GetBytes(postData);
```

Figure 25. Encode data

Authorization token is taken from the response header:

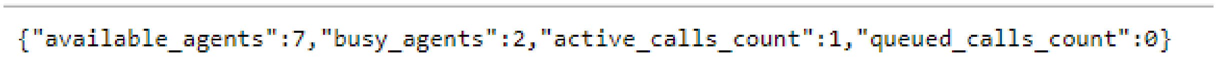
```
response.Headers["Set-Cookie"]
```

This token is used in the next request to */phone/dashboard/dashboard_stats* (Figure 26).

```
private DashboardStatsResponse GetDashboardStats(string cookie) {
      var request = (HttpWebRequest)WebRequest.

      Create("https://help.linnworks.com/phone/dashboard/dashboard_sta
      ts");

      request.Method = "GET";
      request.ContentType = "application/json";

      request.Headers[HttpRequestHeader.Cookie] = cookie;

      var response = (HttpWebResponse)request.GetResponse();

      var responseString = new
StreamReader(response.GetResponseStream()).ReadToEnd();

      return
JsonConvert.DeserializeObject<DashboardStatsResponse>(responseString);
}
```

Figure 26. Request information about calls

Response will be in the following format (Figure 27):



```
{"available_agents":7,"busy_agents":2,"active_calls_count":1,"queued_calls_count":0}
```

Figure 27. Freshdesk response

SecurityProtocolType.Tls2 should be used for both requests.

4.3.5 Current tickets

Freshdesk is used as a ticket system by support department. In order to get information about tickets, API calls to ‘*..helpdesk/tickets/summary.xml?view_name=open’* will be made. Basic authorization in request header has the following format: „apiKey:X“. Endpoint has a limit of 1000 calls per hour what is more than enough for our purpose. Service will make approximately 30 requests per hour. Security protocol has to be set to Tls1.2 to successfully perform requests. That setting is placed in service’s *OnStart* method (Figure 28).

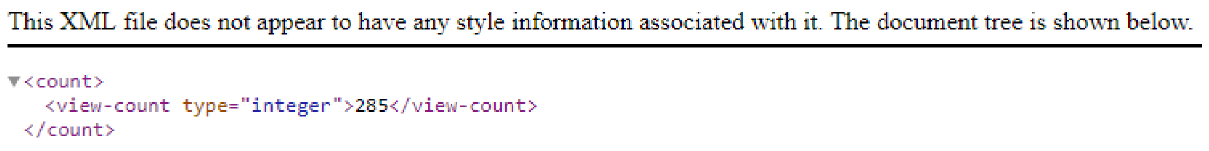
```
protected override void OnStart(string[] args) {
      System.Net.ServicePointManager.SecurityProtocol =
            System.Net.SecurityProtocolType.Tls12;
       Quartz.Register(120);
}
```

Figure 28. Set Tls2 protocol

Tickets with the following statuses will be requested:

- Opened
- New
- Overdue

Each call requests tickets only for requested type one by one. In total three calls are be performed. Response will be in the following format (Figure 29):



```
This XML file does not appear to have any style information associated with it. The document tree is shown below.

▼<count>
   <view-count type="integer">285</view-count>
 </count>
```

Figure 29. Response - tickets count

When tickets from all three categories are collected, they are transformed to *CurrentTickets* class (Figure 30) before being forwarded to database.

```
var openCount = int.Parse(GetSummary(OPEN_COUNT_URL).Count.Text);
var newCount = int.Parse(GetSummary(NEW_COUNT_URL).Count.Text);
var overdueCount =
int.Parse(GetSummary(OVERDUE_COUNT_URL).Count.Text);

list.Add(
new InfluxDB.Net.DataStructure.CurrentTickets(time, openCount,
newCount, overdueCount)
);
```

Figure 30. CurrentTickets class

4.3.6 Monitis monitors

Monitis is a tool for website and servers performance monitoring. It is capable of monitoring server’s load on CPU, RAM, drive, website availability etc. Monitis is already configured to monitor LinnSystems servers’ infrastructure. Monitis API is used [20] to get information about monitors. The problem is that when requesting data about monitors we do not get the information about current state and it is impossible to determine if the monitor is failed or is working flawlessly. Failed monitors should be retrieved using a separate call.

First of all, failed monitors among all groups are requested. Then total amount of monitors is requested for each type:

- Internal monitors
- External monitors
- Other monitors
- Transaction monitors

In total there are 799 monitors. Each monitor belongs to one or many groups or has a tag, which determines which application this monitor is related to. For example, one drive may have two groups if two applications are located on the same drive. At the moment of writing this project, there are the following monitor groups in Monitis: admservice, adm_client, adm_worker, autosync, backup_service, docstore, event_despatcher, event_despatcher_status, fruugo_feed, infrastructure, int_ws, linnlive, lw_net, lw_net_main, lw_net_ext, lw_net_push, meanrepricer, old_acc, old_desktop_webservice, printing_service, pub_ws, shipping_gateway, shipstation, sql, virtual_printer_server, web.

When all monitors are recived, iteration through all failed monitors is performed where status of matching monitor in the list of all monitors is set (Figure 31).

```
allMonitors.ForEach((monitor) =>
      {
            if (failedMonitors.Any(failedMonitor => failedMonitor.Id
      == monitor.Id))
            {
                  monitor.Status = MonitorStatus.Failed;
            }
      });
```

Figure 31. Set failed monitors

Total amount of monitors and amount of failed monitors grouped by application they belong to and by *type* will be saved to database. *Type* field may contain one of the following values:

- Drive
- CPU
- Memory
- Bandwidth,
- Availability

Originally there are more types in monitis but in order to group them somehow for dashboard, they will be converted to listed types (Figure 32).

```
private MonitorType ConvertMonitorType(MonitorTypeMonitis
monitisMonitorType)
{
        switch (monitisMonitorType)
        {
                case MonitorTypeMonitis.Drive:
                case MonitorTypeMonitis.DiskIO:
                        return MonitorType.Drive;

                case MonitorTypeMonitis.CPU:
                        return MonitorType.CPU;

                case MonitorTypeMonitis.Memory:
                        return MonitorType.Memory;

                case MonitorTypeMonitis.Bandwidth:
                        return MonitorType.Bandwidth;

                case MonitorTypeMonitis.HTPPHTTPS:
                case MonitorTypeMonitis.Uptime:
                case MonitorTypeMonitis.TCP:
                case MonitorTypeMonitis.Transaction:
                case MonitorTypeMonitis.Tomcat:
                case MonitorTypeMonitis.NodeJS:
                case MonitorTypeMonitis.Oracle:
                case MonitorTypeMonitis.FullPageLoad:
                case MonitorTypeMonitis.WindowsService:
                case MonitorTypeMonitis.RUM:
                case MonitorTypeMonitis.PING:
                case MonitorTypeMonitis.AdvancedPing:
                case MonitorTypeMonitis.Process:
                        return MonitorType.Availability;

                default:
                        return MonitorType.Availability;
        }
}
```

Figure 32. Monitor type conversion

4.3.7 Customers' feedback

Initially we had three graphs showing amount of active users in Linnworks.net, Linnworks desktop and LinnLive applications. However, as LinnLive product will be closed in near future, the manager suggested to remove that graph from dahboard and use that free space for something else. As all highly required by team infomation is already presented, after some discussion with the team it was decided to display a table with customers feedback which may be a good replacement. Although customers feedback does not give us any information about current system condition, it would be interesting for the team members to read it. This information will point weak places of Linnworks and at the same time reading feedback may be entertaining. The problem is that we can't easily show feedback directly from database as Grafana is unable to show unindexed text fields such as feedback message. Fortunately, it is possible to use information in JSON format and display it as a table. However, there appears another problem: we have to transform information taken from SQL database to JSON format and pass it to Grafana. In order to pass JSON to Grafana a datasource plugin called SimpleJSON has been installed on Grafana instance. It allows integration with JSON datasources. Then a web page with api has been created to provide feedback in a JSON format for Grafana. Its purpose is to get feedback from SQL database between requested dates which have been selected in Grafana UI.Project called *linnworks.dasboard.webapi* has been created (Figure 33).

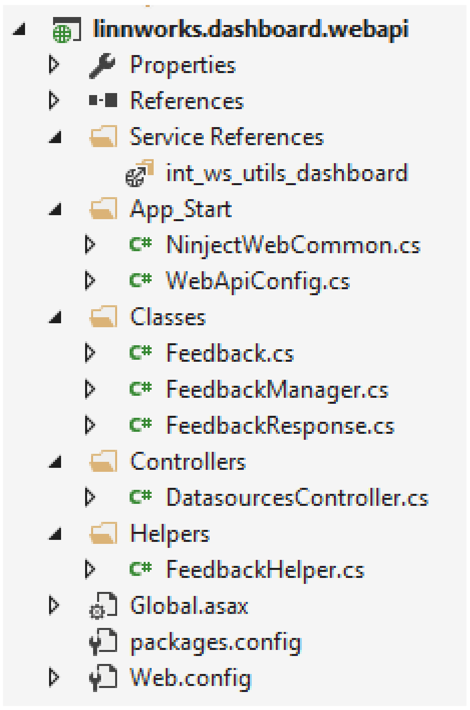

```
◢ linnworks.dashboard.webapi
  ▷ Properties
  ▷ References
  ◢ Service References
      int_ws_utils_dashboard
  ◢ App_Start
    ▷ NinjectWebCommon.cs
    ▷ WebApiConfig.cs
  ◢ Classes
    ▷ Feedback.cs
    ▷ FeedbackManager.cs
    ▷ FeedbackResponse.cs
  ◢ Controllers
    ▷ DatasourcesController.cs
  ◢ Helpers
    ▷ FeedbackHelper.cs
  ▷ Global.asax
    packages.config
  ▷ Web.config
```

Figure 33. linnworks.dashboard.webapi structure

DatasourcesController has three methods:

- Index() – method returns dummy data and is used by Grafana test method during datasource integration to ensure that datasource is valid
- Search() – method returns possible metrics to be selected in Grafana. In our case we do not need metrics and this method returns empty string
- Query() – main method which requests JSON data for the requested time period

When request is received, we take start time and end time from the query (Figure 34) convert it into datetime format and pass it to *int_ws_utils_dashboard* service.

```
HttpContext.Current.Request.UrlReferrer.Query
```

Figure 34 Query

Dashboard.svc service has already been created for collecting amount of active users, therefore just a new method will be added to an existing service. Method *GetFeedback* (Figure 35) which uses following procedure to fetch data from database (Figure 36) has been added to the service.

```
List<Feedback> GetFeedback(DateTime startTime, DateTime endTime)
```

Figure 35. GetFeedback

```
IF NOT EXISTS (SELECT * FROM sys.objects WHERE object_id =
OBJECT_ID(N'[feedback].[customer_feedback_register_get]') AND type in
(N'P', N'PC'))
BEGIN
EXEC dbo.sp_executesql @statement = N'CREATE PROCEDURE
[feedback].[customer_feedback_register_get] AS
RAISERROR(''[feedback].[customer_feedback_register_get] is not yet
defined'', 16, 1);                     '
END
GO
ALTER PROCEDURE [feedback].[customer_feedback_register_get] (
      @startTime DATETIME,
      @endTime DATETIME
) as
BEGIN
      SELECT
            cfr.[Time],
            cfrt.[pkFeedbackTypeId],
            cfrt.[Name],
            cfr.[Username],
            cfr.[Message]
      FROM [feedback].[customer_feedback_register] as cfr
      INNER JOIN feedback.customer_feedback_register_type as cfrt on
cfrt.pkFeedbackTypeId = crf.fkFeedbackTypeId
      WHERE [Time] BETWEEN @startTime AND @endTime
END
GO
```

Figure 36. Feedback - stored procedure

Recived from the service data is converted to appropriate for dashboard format (Figure 37) and returned to Grafana where it is displayed in a table element.

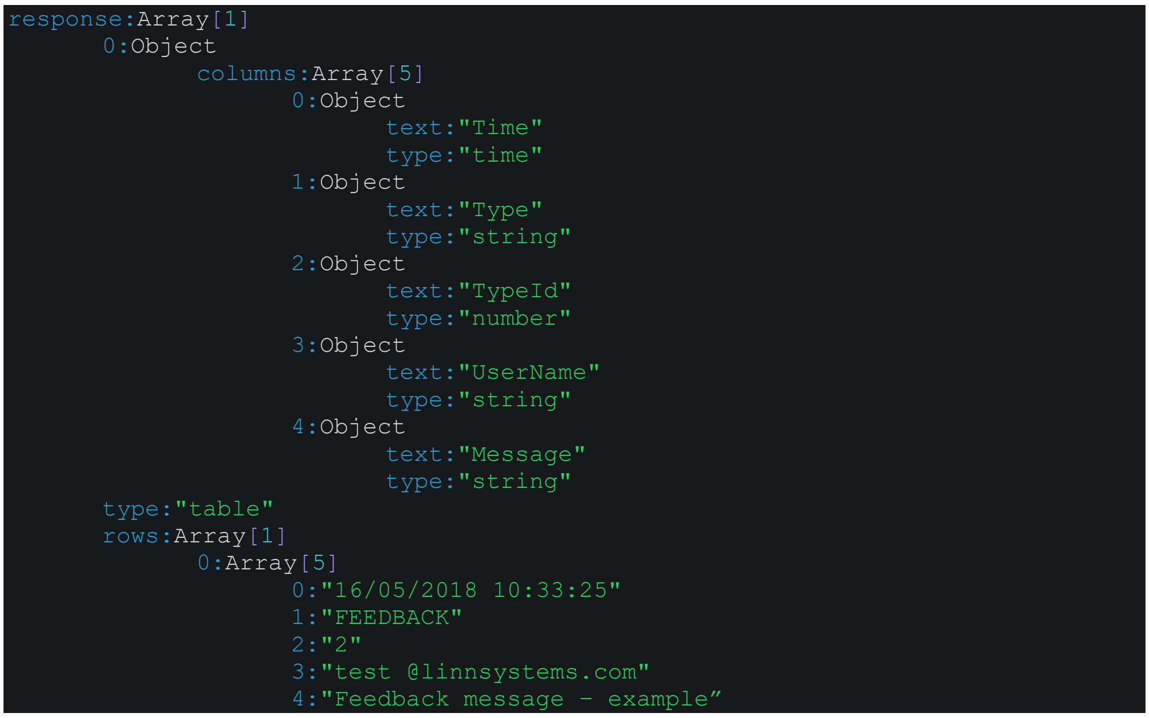

```
response:Array[1]
        0:Object
                columns:Array[5]
                        0:Object
                                text:"Time"
                                type:"time"
                        1:Object
                                text:"Type"
                                type:"string"
                        2:Object
                                text:"TypeId"
                                type:"number"
                        3:Object
                                text:"UserName"
                                type:"string"
                        4:Object
                                text:"Message"
                                type:"string"
        type:"table"
        rows:Array[1]
                0:Array[5]
                        0:"16/05/2018 10:33:25"
                        1:"FEEDBACK"
                        2:"2"
                        3:"test @linnsystems.com"
                        4:"Feedback message – example"
```

Figure 37. Grafana - required format

4.4 Data saving and storing

Company is already using Elasticsearch and InfluxDB database to store different types of information. As there are many new objects with different data structures that have to be stored as a part of this dashboard project, structure of saved entities should be reworked.

4.4.1 Elasticsearch log structure

Elasticsearch is used to store exceptions and various system logs from Linnworks application. In order to prepare data related to exceptions and current online for database, already existing *linnworks.logger* project will be used. However, it has to be modified and improved to handle requirements of this project. Each type of log or entity, except 'stats' should correspond to this template and contain the following fields (Table 2):

Field name	**Possible value (example)**
app	lw_net / lw_net_push / ..
type	exception / IIS / user_logs / system / stats / ...
severity	error / info / warning / ..

Table 2. Log entity structure

For dashboard project ‘exceptions’ type is used. It is not planned to use other types in the scope of this project. Structure has been changed in order to come to one standard solution of separation logs by their type.

4.4.2 InfluxDB data structure

In InfluxDB information entities are separated by Measurements. Measurement can be compared to a table in SQL-based database. It this project every job saves information under its own measurement. For example, measurement for amount of active users shown on Figure 38.

```
public class CurrentOnline : BaseDataStructure
{
      public override string Measurement
      {
            get
            {
                  return "current_online";
             }
        }
```

Figure 38. Measurement - current online

An adapter which works with InfluxDB already exists in *linn_foundation* solution. It has been added and modified to work with our database instance by LinnSystems’ database administrator. Author of this thesis added the following classes to *linnworks.influxdb* project which represents structure of measurements for different types of information (Figure 39):

- CurrentCalls
- CurrentChats
- CurrentMonitors

- CurrentOnline
- CurrentTickets

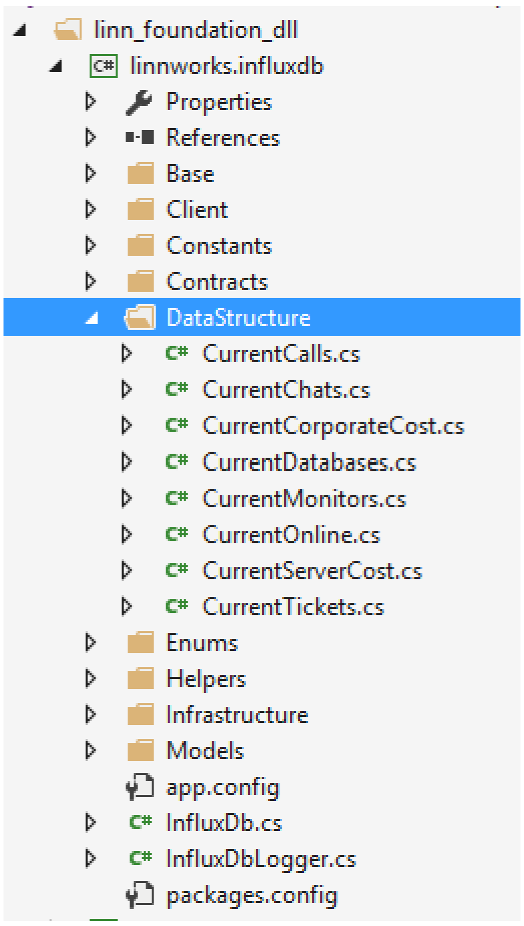


Figure 39. InfluxDB - data structure

Each structure class contains constructor with parameters which have to be saved to one document (Figure 40).

```
public CurrentCalls(DateTime time, int activeCallsCount, int
queuedCallsCount, int availableAgentsCount, int busyAgentsCount) :
            base(time)
{
      this.ActiveCallsCount = activeCallsCount;
      this.QueuedCallsCount = queuedCallsCount;
      this.AvailableAgentsCount = availableAgentsCount;
      this.BusyAgentsCount = busyAgentsCount;
}
```

Figure 40. Current calls constructor

Also class has method *GetFeilds*() or/and *GetTags*(). These methods create dictionaries with property name and value (Figure 41).

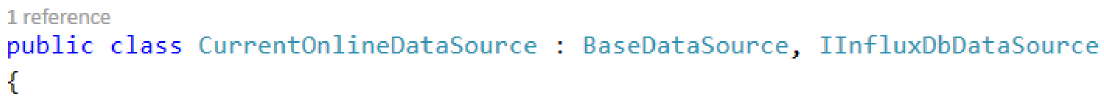
```
public override Dictionary<string, object> GetFields()
{
      var fields = base.GetFields();

      fields.Add("active_calls", ActiveCallsCount);
      fields.Add("queued_calls", QueuedCallsCount);
      fields.Add("available_agents", AvailableAgentsCount);
      fields.Add("busy_agents", BusyAgentsCount);

      return fields;
}
```

Figure 41. GetFields example

Then this data is being saved to InfluxDB. Saving process has been implemented by another person, it will not be described in this thesis.

4.4.3 Determine database

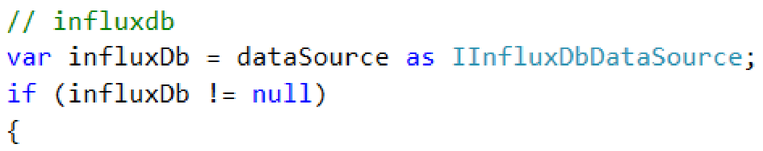
Service's adapter has been designed in such way that it has ability to determine where data provided by job has to be saved: to Elasticsearch or to InfluxDB. It depends on inherited interface (Figure 42):



```
1 reference
public class CurrentOnlineDataSource : BaseDataSource, IInfluxDbDataSource
{
```

Figure 42. Datasource interface

In adapter we check if current job is supposed to use *KibanaDataSource* or *InfluxDbDataSource* (Figure 43).



```
// influxdb
var influxDb = dataSource as IInfluxDbDataSource;
if (influxDb != null)
{
```

Figure 43. Determine data source

However, it the scope of this project our service will save data only to InfluxDB. Elasticsearch data source has been added as an option for the future in case there appears a need to save something to Elasticsearch.

4.4.4 Monitoring

In order to keep track of unhandled exceptions occurred in windows service, they will be saved to Elasticsearch and will be available in Kibana. To achieve that, jobs execution code part will be covered by try-catch and common logger will be used (Figure 44). This will help the author and team members to identify problems with service.

```
} catch (Exception ex) {
	var logger =
	new linnworks.logger.ExceptionsLogger(
	linnworks.logger.BaseLogger.AppName.dashboard_service);

	logger.SaveLog(DateTime.UtcNow, ex);
}
```

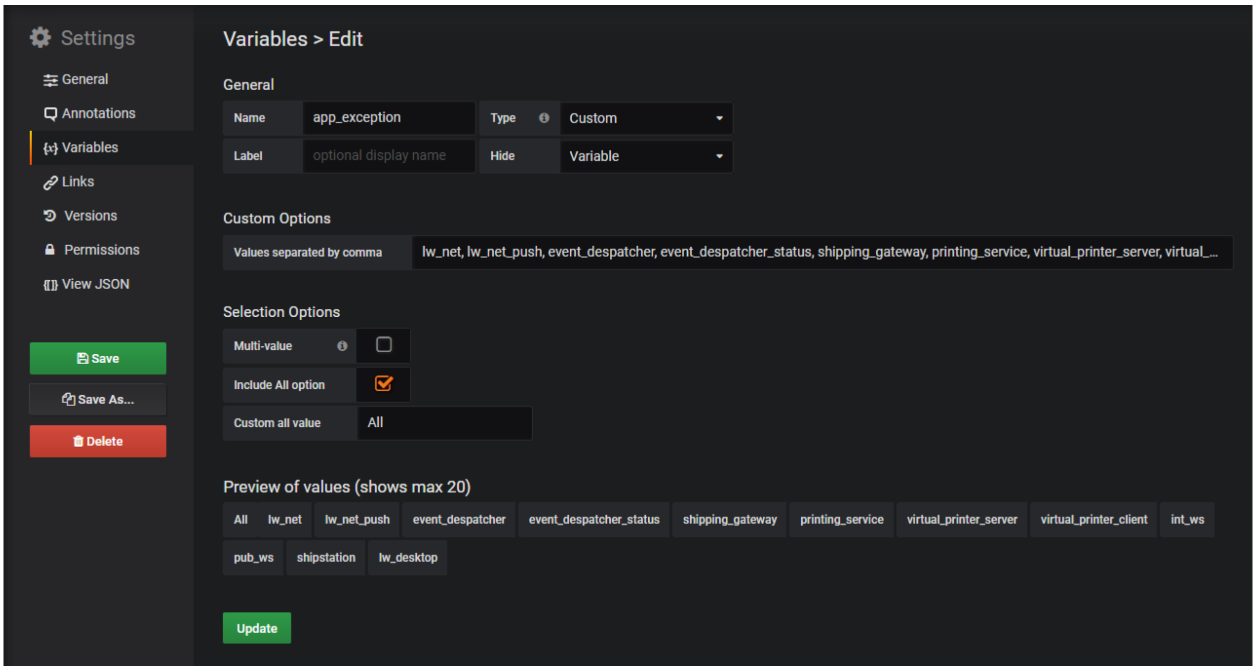
Figure 44. Exceptions logger

4.5 Visualizing data on dashboard

For vizualizing we will use Grafana 5. In order to display information from databases on visual elements we have to write queries. Queries will vary depending on datasource selected for the element. For example, for InfluxDB we have to write Lucene queries or compose queries in the UI. Almost all diagrams contain time axis, as it is important for the team to monitor system status over time.

4.5.1 Grafana templating

Grafana supports creating variables so there is no need to hardcode application types. Moreover, ’templating’ option allows us to easily manage many elements at the same time. In other words, when we edit one element, changes are applied to all elements related to variable (Figure 45).



Variables > Edit

General

Name	app_exception	Type	Custom
Label	optional display name	Hide	Variable

Custom Options

Values separated by comma: lw_net, lw_net_push, event_despatcher, event_despatcher_status, shipping_gateway, printing_service, virtual_printer_server, virtual_...

Selection Options

Multi-value	☐
Include All option	✓
Custom all value	All

Preview of values (shows max 20)

All, lw_net, lw_net_push, event_despatcher, event_despatcher_status, shipping_gateway, printing_service, virtual_printer_server, virtual_printer_client, int_ws, pub_ws, shipstation, lw_desktop

Figure 45. Grafana variables

4.5.2 Grafana 'health boxes'

Below we will look closer at the 'health boxes' elements. Let's see how one element in this combination of multiple boxes (Figure 46) is configured to display data.

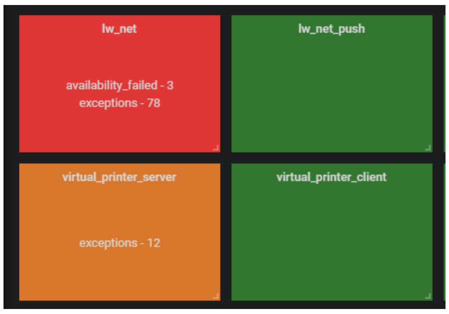
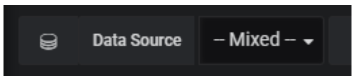


Figure 46. Grafana "Health box" element

Source type has to be set as 'Mixed' (Figure 47) as these 'health boxes' require data both from Kibana and from InfluxDB.



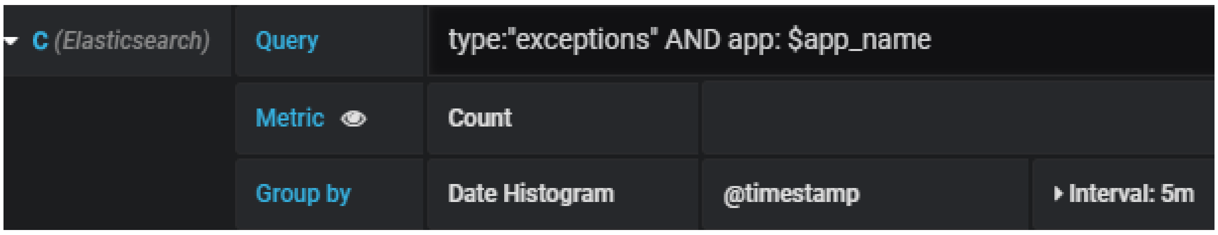
Data Source: – Mixed –

Figure 47. Mixed data source

For each metric there is a separate query. They are listed below.

- Exceptions (Figure 48):

Query that gets all records for the last 5 minutes with type ‘exceptions’ for *$app_name* (for example, “lw_net”) from Elasticsearch database is used.



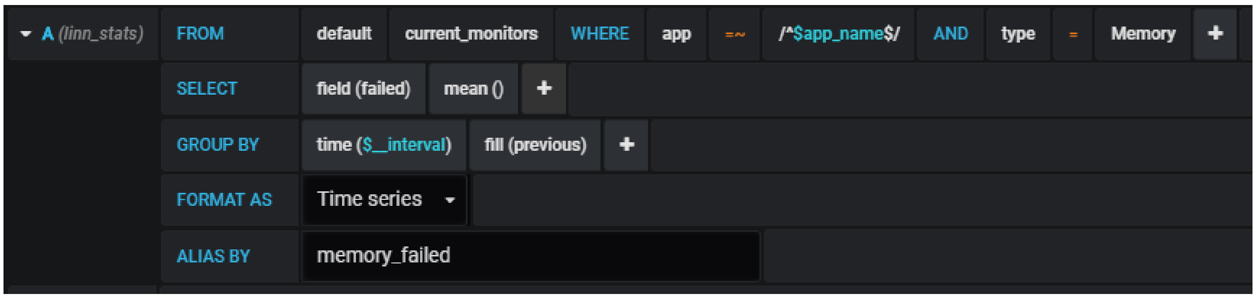
```
C (Elasticsearch)  Query     type:"exceptions" AND app: $app_name
                   Metric    Count
                   Group by  Date Histogram  @timestamp  Interval: 5m
```

Figure 48. Query - Exceptions

All information described in section below will be taken from InfluxDB called linn_stats.

- Memory (Figure 49):

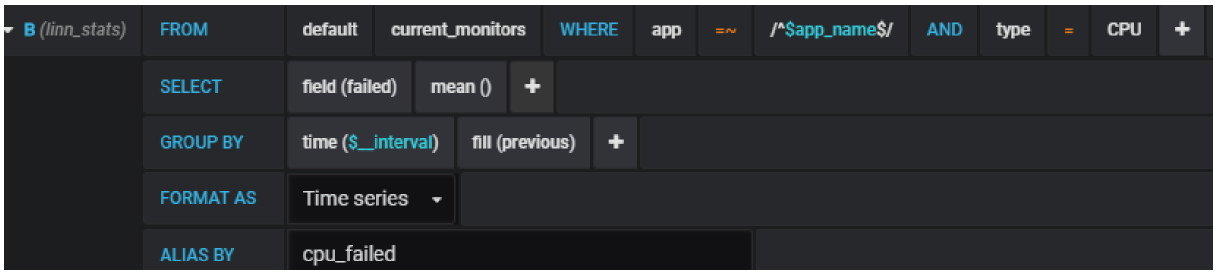
Information from InfluxDB with measurement ‘current_monitors’ where type is ‘Memory’ and app is $app_name is requested in order to get information about memory. Field ‘failed’ is selected. Results are grouped by time interval. Option ‘fill(previous)’ will take last value.



```
A (linn_stats)  FROM       default  current_monitors  WHERE  app  =~  /^$app_name$/  AND  type  =  Memory  +
                SELECT     field (failed)  mean ()  +
                GROUP BY   time ($__interval)  fill (previous)  +
                FORMAT AS  Time series
                ALIAS BY   memory_failed
```

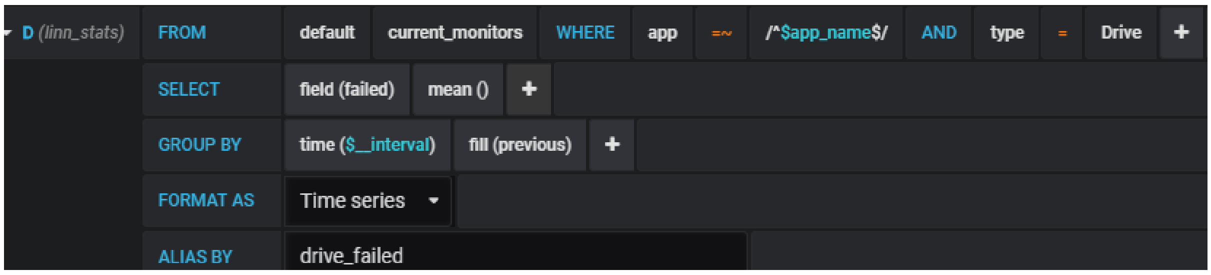
Figure 49. Query - Memory

- CPU (Figure 50):



```
B (linn_stats)  FROM       default  current_monitors  WHERE  app  =~  /^$app_name$/  AND  type  =  CPU  +
                SELECT     field (failed)  mean ()  +
                GROUP BY   time ($__interval)  fill (previous)  +
                FORMAT AS  Time series
                ALIAS BY   cpu_failed
```

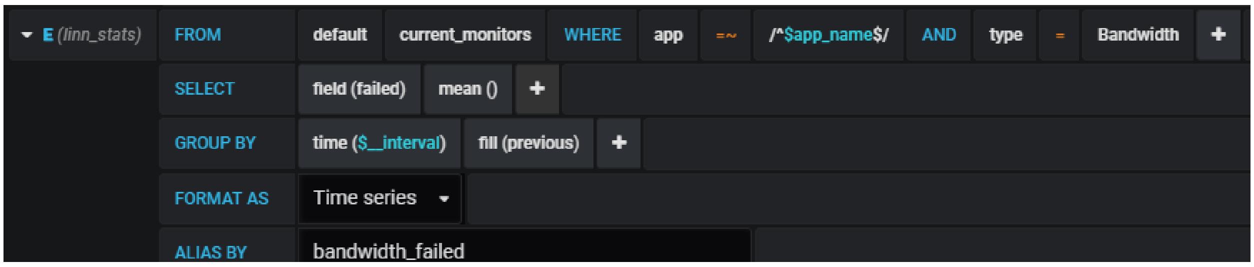
- Drive (Figure 51):



D (linn_stats) FROM default current_monitors WHERE app =~ /^app_name/ AND type = Drive +
SELECT field (failed) mean () +
GROUP BY time ($__interval) fill (previous) +
FORMAT AS Time series
ALIAS BY drive_failed

Figure 51. Query - Drive

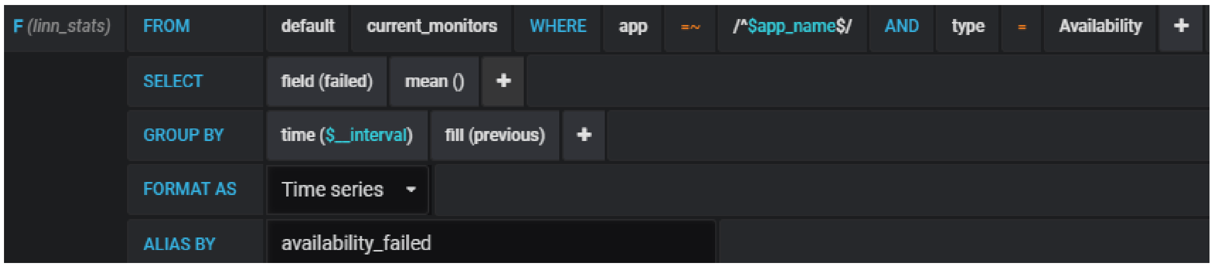
- Bandwidth (Figure 52):



E (linn_stats) FROM default current_monitors WHERE app =~ /^app_name/ AND type = Bandwidth +
SELECT field (failed) mean () +
GROUP BY time ($__interval) fill (previous) +
FORMAT AS Time series
ALIAS BY bandwidth_failed

Figure 52. Query - bandwidth

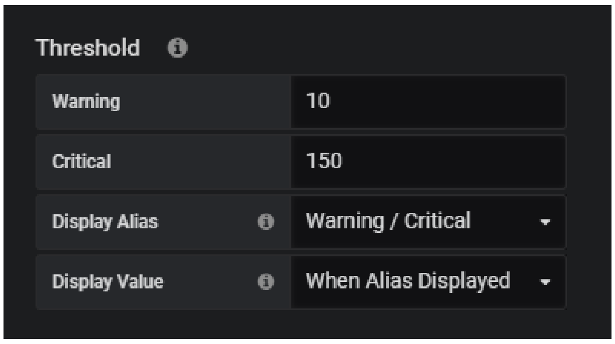
- Availability (Figure 53):



F (linn_stats) FROM default current_monitors WHERE app =~ /^app_name/ AND type = Availability +
SELECT field (failed) mean () +
GROUP BY time ($__interval) fill (previous) +
FORMAT AS Time series
ALIAS BY availability_failed

Figure 53. Query - Availability

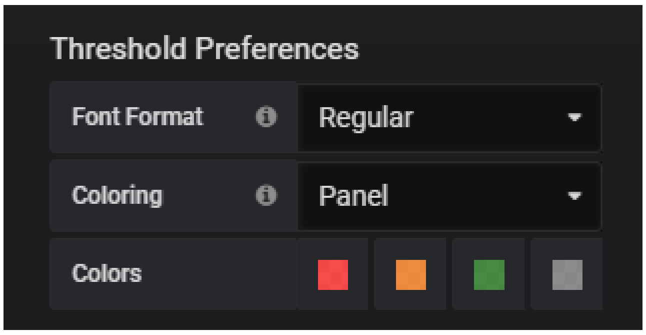
When any of these metrics reach ‘Warning’ or ‘Critical’ condition (Figure 54), metric name and value will be visible on dashboard. This is configured by option ‘Display alias’ set to ‘Warning/Critical’.


Threshold

Warning	10
Critical	150
Display Alias	Warning / Critical
Display Value	When Alias Displayed

Figure 54. Warning threshold

When everything is in its normal state, all information will be hidden and box will have green background (Figure 55).


Threshold Preferences

Font Format	Regular
Coloring	Panel
Colors	🟥 🟧 🟩 ⬜

Figure 55. Background colouring

4.6 Service deployment

Deployment is performed using Jenkins [7]. Jenkins is configured to replace authorization keys and connection strings for security purposes. Jenkins greatly helps to deploy new versions of application as there is no need to manually interact with the server.

Deployment has 3 stages:

1. Preparation – loads last stable branch from repository and builds it
2. Deployment – deploys code on server in a separate new folder
3. Switch version – switches working service to work from the new folder

4.7 Display dashboard on display

There is a need to somehow show dashboard on a TV screen. RiseVision [21] will be used to manage dashboard on the screen. Once display is connected to RiseVision account, it is possible to easily edit dashboard without manipulations with TV where dashboard is installed.

Maintenance and configuration of displayed information is performed from website https://apps.risevision.com. There is no need to install additional software. (Figure 56).

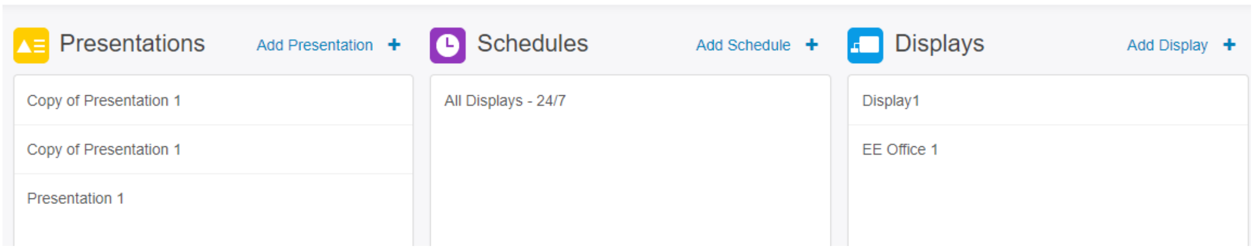


Figure 56. RiseVision control menu

As we have two dashboards (main board and one only with exceptions), RiseVision has been configured to rotate two images every 15 seconds. To achieve that, two pages have been added to the playlist (Figure 57).

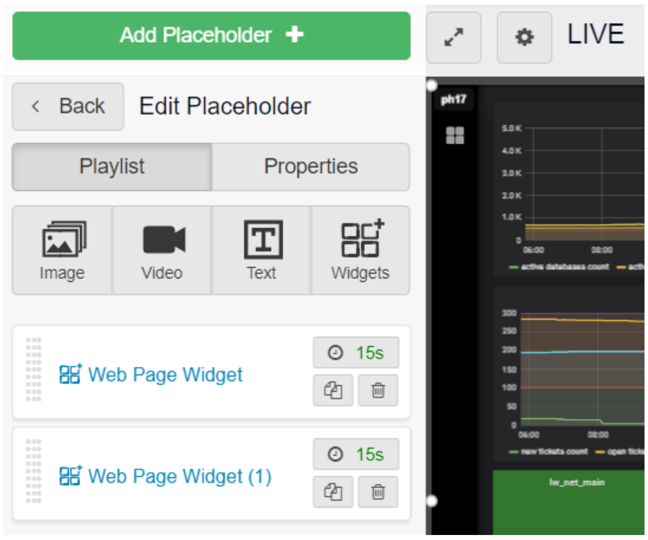
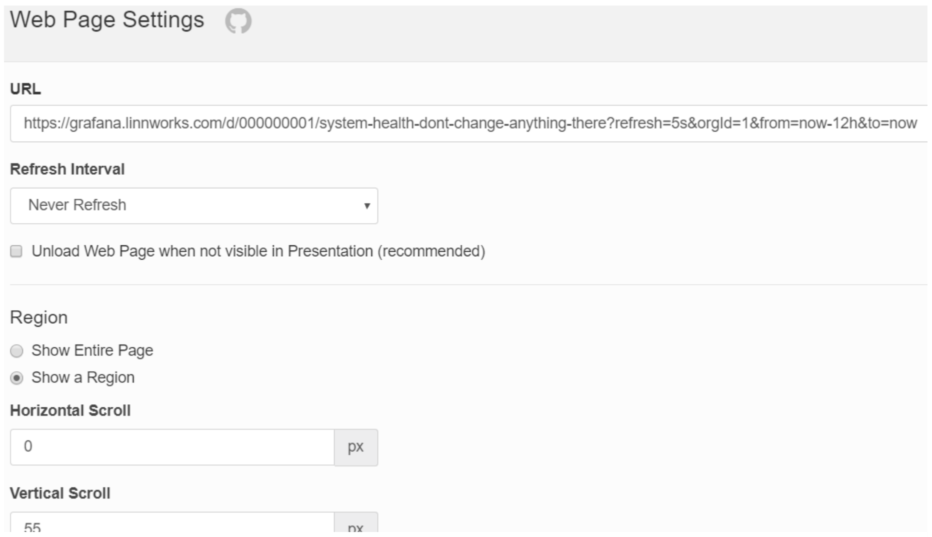


Figure 57. RiseVision playlist

Each web page widget has an url reference (Figure 58).


Web Page Settings

URL

https://grafana.linnworks.com/d/000000001/system-health-dont-change-anything-there?refresh=5s&orgId=1&from=now-12h&to=now

Refresh Interval

Never Refresh

☐ Unload Web Page when not visible in Presentation (recommended)

Region

○ Show Entire Page

● Show a Region

Horizontal Scroll

0 px

Vertical Scroll

55 px

Figure 58. Widget settings

4.7.1 Prepare TV

Intel Compute Stick – small computer for media applications is used to launch RiseVision on TV. Intel stick with installed windows 10 has been connected to TV using HDMI port. Rise Player software is needed to show presentation from RiseVision on TV display. After installation, the key generated on RiseVision website has to be entered in order to connect that TV with our main account. After that, the presentation which will be shown on TV should be selected. As a result, we can control our dashboard from any other computer by changing presentation settings.

4.8 Testing

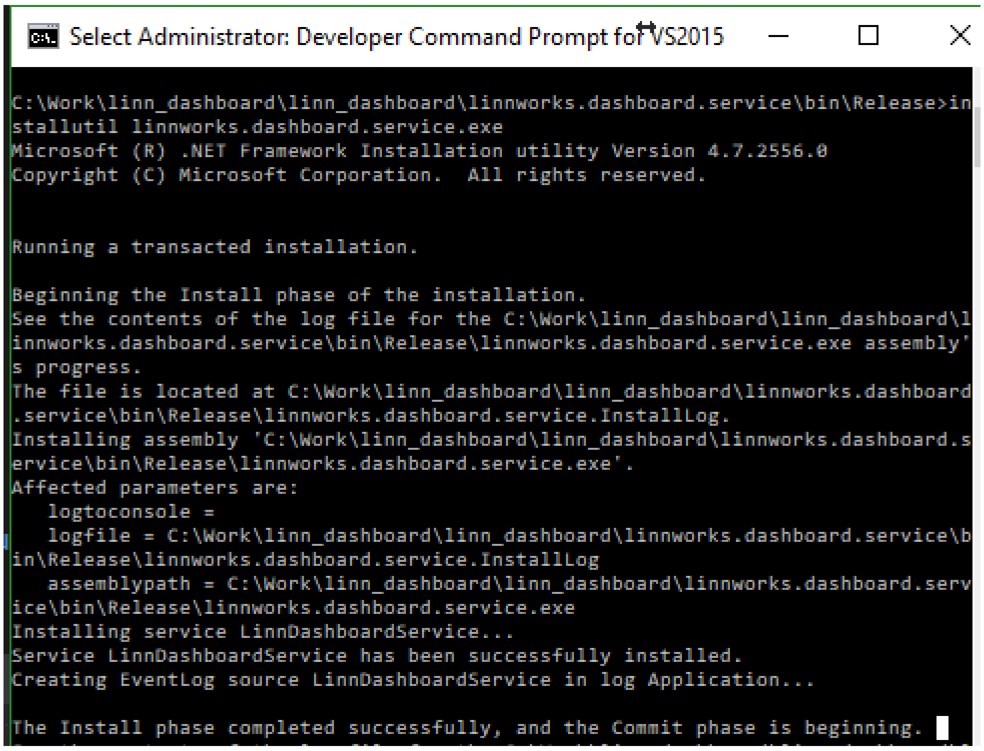
Elements of solution have been tested manually using smoke testing. Code inspection has been performed by team members. There is no need to perform performance testing because requirements do not have strict limitation for job execution time.

4.8.1 Scheduling

Scheduling has been tested on early development stages when running service in debug mode in Visual Studio 2015.

On final development stage, windows service has been installed on local machine and started as a service [22]. Developer Command prompt for VS2015 was used to install a service using the following command (Figure 59):

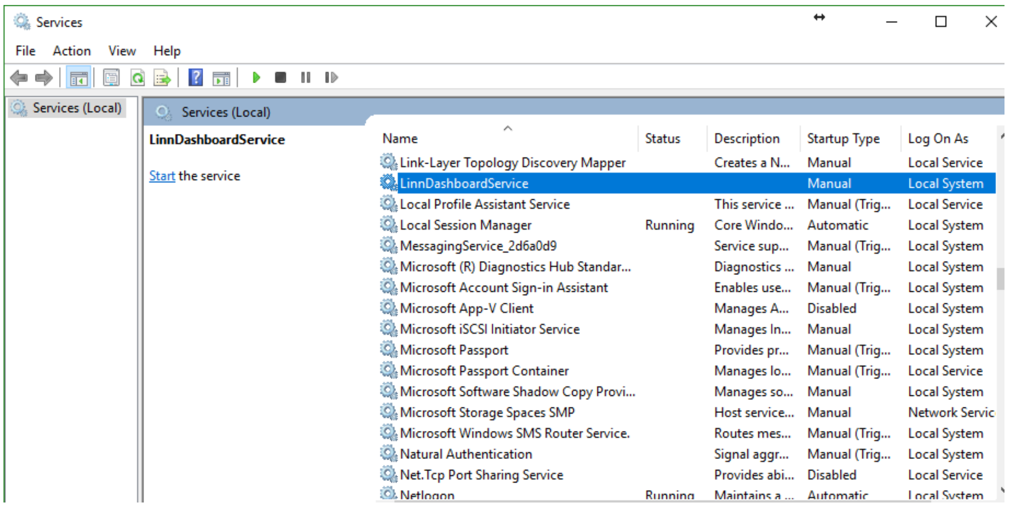
\linn_dashboard\linn_dashboard\linnworks.dashboard.service\bin\Release> installutil linnworks.dashboard.service.exe



```
C:\Work\linn_dashboard\linn_dashboard\linnworks.dashboard.service\bin\Release>in
stallutil linnworks.dashboard.service.exe
Microsoft (R) .NET Framework Installation utility Version 4.7.2556.0
Copyright (C) Microsoft Corporation.  All rights reserved.


Running a transacted installation.

Beginning the Install phase of the installation.
See the contents of the log file for the C:\Work\linn_dashboard\linn_dashboard\l
innworks.dashboard.service\bin\Release\linnworks.dashboard.service.exe assembly'
s progress.
The file is located at C:\Work\linn_dashboard\linn_dashboard\linnworks.dashboard
.service\bin\Release\linnworks.dashboard.service.InstallLog.
Installing assembly 'C:\Work\linn_dashboard\linn_dashboard\linnworks.dashboard.s
ervice\bin\Release\linnworks.dashboard.service.exe'.
Affected parameters are:
   logtoconsole =
   logfile = C:\Work\linn_dashboard\linn_dashboard\linnworks.dashboard.service\b
in\Release\linnworks.dashboard.service.InstallLog
   assemblypath = C:\Work\linn_dashboard\linn_dashboard\linnworks.dashboard.serv
ice\bin\Release\linnworks.dashboard.service.exe
Installing service LinnDashboardService...
Service LinnDashboardService has been successfully installed.
Creating EventLog source LinnDashboardService in log Application...

The Install phase completed successfully, and the Commit phase is beginning.
```

Figure 59. Service installation

After installation is complete, service may be started from Services management window (Figure 60):



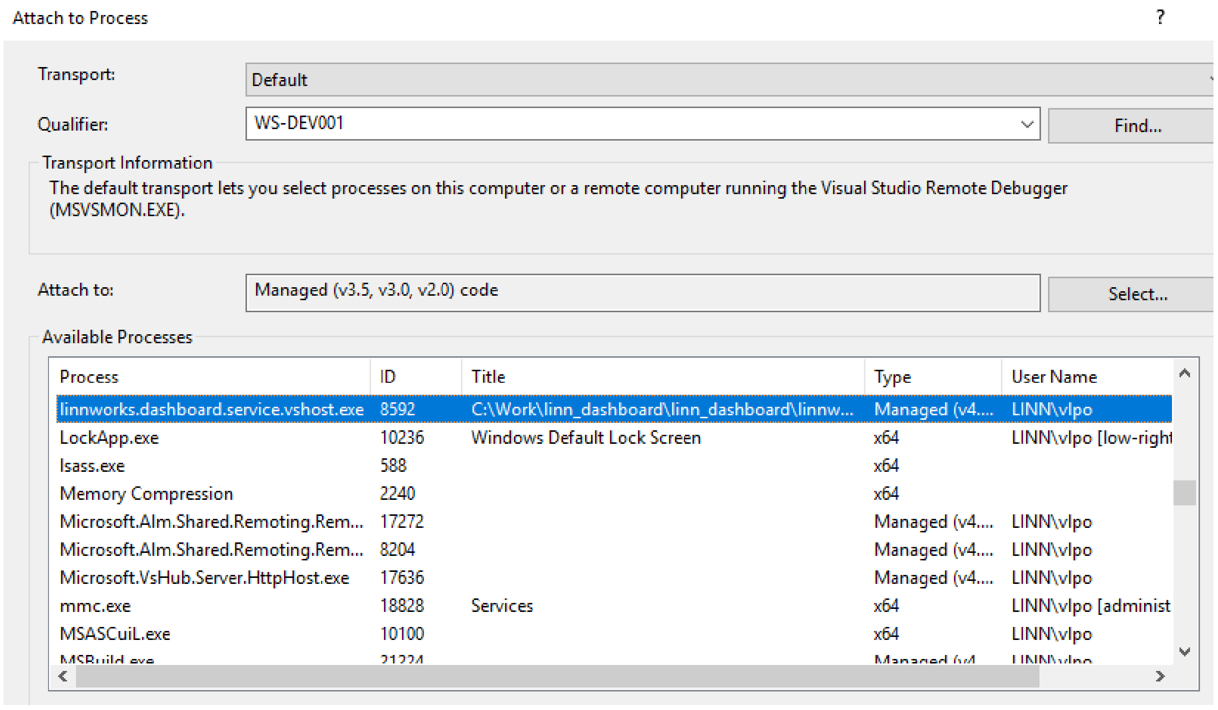
LinnDashboardService

Start the service

Name	Status	Description	Startup Type	Log On As
Link-Layer Topology Discovery Mapper		Creates a N...	Manual	Local Service
LinnDashboardService			Manual	Local System
Local Profile Assistant Service		This service ...	Manual (Trig...	Local Service
Local Session Manager	Running	Core Windo...	Automatic	Local System
MessagingService_2d6a0d9		Service sup...	Manual (Trig...	Local System
Microsoft (R) Diagnostics Hub Standar...		Diagnostics ...	Manual	Local System
Microsoft Account Sign-in Assistant		Enables use...	Manual (Trig...	Local System
Microsoft App-V Client		Manages A...	Disabled	Local System
Microsoft iSCSI Initiator Service		Manages In...	Manual	Local System
Microsoft Passport		Provides pr...	Manual (Trig...	Local System
Microsoft Passport Container		Manages lo...	Manual (Trig...	Local Service
Microsoft Software Shadow Copy Provi...		Manages so...	Manual	Local System
Microsoft Storage Spaces SMP		Host service...	Manual	Network Servic
Microsoft Windows SMS Router Service.		Routes mes...	Manual (Trig...	Local System
Natural Authentication		Signal aggr...	Manual (Trig...	Local System
Net.Tcp Port Sharing Service		Provides abi...	Disabled	Local Service
Netlogon	Running	Maintains a ...	Automatic	Local System

Figure 60. Starting service

Then Visual Studio debugger can be attached to a running process (Figure 61) and appropriate breakpoints could be set to ensure that service executes itself as intended.


Attach to Process ?

Transport: Default

Qualifier: WS-DEV001 Find...

Transport Information

The default transport lets you select processes on this computer or a remote computer running the Visual Studio Remote Debugger (MSVSMON.EXE).

Attach to: Managed (v3.5, v3.0, v2.0) code Select...

Available Processes

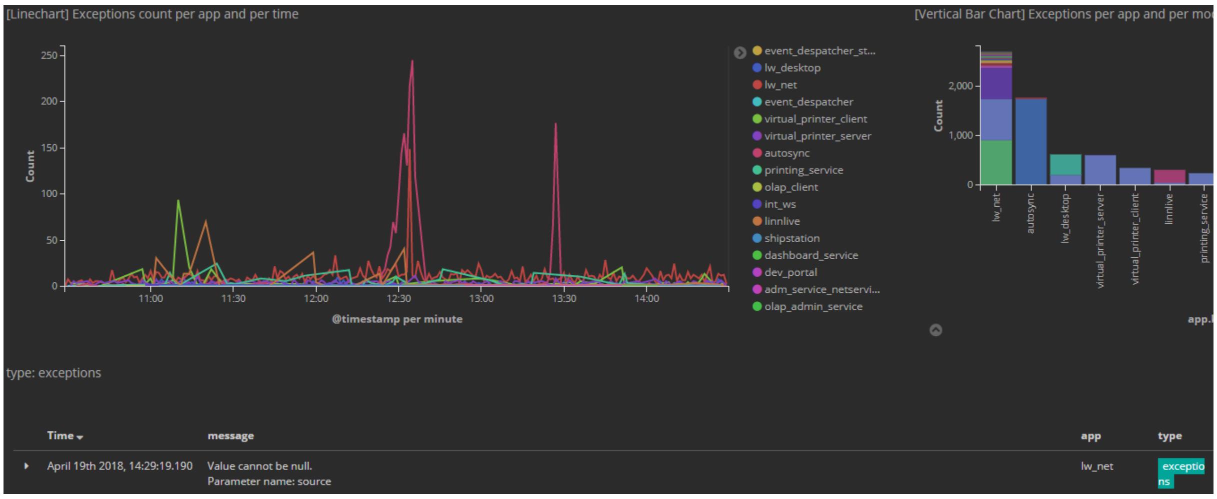
Process	ID	Title	Type	User Name
linnworks.dashboard.service.vshost.exe	8592	C:\Work\linn_dashboard\linn_dashboard\linnw...	Managed (v4....	LINN\vlpo
LockApp.exe	10236	Windows Default Lock Screen	x64	LINN\vlpo [low-right
lsass.exe	588		x64	
Memory Compression	2240		x64	
Microsoft.Alm.Shared.Remoting.Rem...	17272		Managed (v4....	LINN\vlpo
Microsoft.Alm.Shared.Remoting.Rem...	8204		Managed (v4....	LINN\vlpo
Microsoft.VsHub.Server.HttpHost.exe	17636		Managed (v4....	LINN\vlpo
mmc.exe	18828	Services	x64	LINN\vlpo [administ
MSASCuiL.exe	10100		x64	LINN\vlpo
MSBuild.exe	21224		Managed (v4	LINN\vlpo

Figure 61. Visual Studio debugger

However, this process of attaching debugger to a process is too complicated. There is an easier way to debug running service – we just have to add *System.Diagnostics.Debugger.Launch();* in the code we want to debug.

4.8.2 Data collecting

Results returned by data-collecting jobs have been manually verified by comparing them to the results in the sources. For example, amount of chats and tickets has been checked on the freshdesk website. Exceptions have been compared to ones in Kibana (Figure 62).



type: exceptions

Time	message	app	type
April 19th 2018, 14:29:19.190	Value cannot be null. Parameter name: source	lw_net	exceptions

Figure 62. Kibana dashboard

4.8.3 Grafana board

Dashboard's user interface does not need special testing as it is build using existing Grafana set of tools. Graphs are updated over defined period of time as intended (Figure 63).

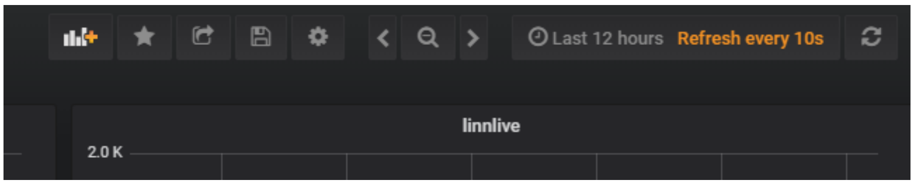


Figure 63. Grafana refresh rate

Summary

The aim of this thesis was to provide a complete solution for monitoring system status for LinnSystems. It involved choice of different dashboard tools, databases and their internal structure, design of dashboard elements, configuration of TV display and implementation of service including related development.

During the project development, author has proposed several prototypes which have been tested and discussed. After launching first versions on production new requirements for dashboard have been made based on team members' feedback.

As a result of this thesis, two of the presented prototypes has been chosen and are used in the company allowing company workers to easily get information about overall system condition. Current dashboard meets the requirements set by team members, manager and satisfies their needs. System has been set up in such way that it is easy to develop and deploy improvements for this project in the future.

The following tools and technologies have been used in this project: Windows Service, task scheduling, NoSQL databases, dashboard visualization tools, RiseVision, Selenium and much more.

As this software is being evolved, in final version of the dashboard some elements or information might be slightly different from the one described in this document.

Kokkuvõte

Käesoleva lõputöö eesmärgiks oli välja töötada rakendus, mis teostaks Linnsystems süsteemi staatuste seiret. Rakenduse väljatöötamine koosnes järgnevatest etappidest: sobiva infopaneeli, andmebaasi ja selle struktuuri valikust, teenuse loomisest, infopaneeli elementide disainist ja TV ekraani konfigureerimisest.

Töö käigus lõputöö autor esitas mitu prototüüpi, mis olid testitud ja arutletud koos LinnSystems meeskonnaga. Prototüüpide kasutusajal meeskond pakkus välja uued nõuded infopaneelile.

Antud lõputöö tulemuseks sai kaks prototüüpi, mis on praegu kasutusel ettevõttes. Infopaneelide abil töötajad saavad informatsiooni süsteemi seisundi kohta. Rakendus vastab püstitatud nõuetele ja oli disainitud nii, et tulevikus oleks lihtne seda projekti edasi arendada ja täiustada.

Lõputöös kasutati järgmiseid tehnoloogiad: Window Service, NoSQL andmebaasid, erinevad infopaneeli tööristad, RiseVision, Selenium ning palju muud.

Rakendus on veel arenemisel, mistõttu mõned dokumendis esitatud infopaneeli elemendid võivad erineda.

Table of literature

[1] “E-commerce,” [Online]. Available: https://en.wikipedia.org/wiki/E-commerce. [Accessed 02 04 2018].

[2] “Elasticsearch,” [Online]. Available: https://en.wikipedia.org/wiki/Elasticsearch. [Accessed 28 03 2018].

[3] “Logstash,” [Online]. Available: https://wikitech.wikimedia.org/wiki/Logstash. [Accessed 02 04 2018].

[4] “InfluxDB,” [Online]. Available: https://en.wikipedia.org/wiki/InfluxDB. [Accessed 25 03 2018].

[5] “Kibana,” [Online]. Available: https://en.wikipedia.org/wiki/Kibana. [Accessed 15 01 2018].

[6] “Grafana,” [Online]. Available: https://grafana.com/. [Accessed 02 04 2018].

[7] “Jenkins,” [Online]. Available: https://jenkins.io. [Accessed 07 05 2018].

[8] “nosql,” [Online]. Available: https://www.infoworld.com/article/3240644/nosql/what-is-nosql-nosql-databases-explained.html. [Accessed 07 05 2018].

[9] “quartzNet,” [Online]. Available: https://www.quartz-scheduler.net/. [Accessed 07 05 2018].

[10] “AWS CloudWatch,” [Online]. Available: https://aws.amazon.com/cloudwatch/?nc1=h_ls. [Accessed 19 05 2018].

[11] “Benchmark. InfluxDb - Elasticsearch,” [Online]. Available: https://www.influxdata.com/blog/influxdb-markedly-elasticsearch-in-time-series-data-metrics-benchmark/. [Accessed 16 05 2018].

[12] “DB-Engines ranking,” [Online]. Available: https://db-engines.com/en/ranking/time+series+dbms. [Accessed 16 05 2018].

[13] “Benchmark. Influx - OpenTSDB,” [Online]. Available: https://www.influxdata.com/blog/influxdb-markedly-outperforms-opentsdb-in-time-series-data-metrics-benchmark/. [Accessed 16 05 2018].

[14] “NewtonsoftJSON,” [Online]. Available: https://www.newtonsoft.com/json. [Accessed 02 04 2018].

[15] “Selenium,” [Online]. Available: https://www.seleniumhq.org/about/. [Accessed 24 03 2018].

[16] “Ninject,” [Online]. Available: http://www.ninject.org/. [Accessed 29 03 2018].

[17] “Quartz.NET,” [Online]. Available: https://www.quartz-scheduler.net/. [Accessed 24 01 2018].

[18] “Windows service as a console app,” [Online]. Available: https://alastaircrabtree.com/how-to-run-a-dotnet-windows-service-as-a-console-app/. [Accessed 16 05 2018].

[19] “monitis,” [Online]. Available: https://www.pcmag.com/business/directory/application-performance-

management/1770-monitis. [Accessed 08 05 2018].

[20] "Monitis API," [Online]. Available: http://www.monitis.com/docs/apiActions.html. [Accessed 16 05 2018].

[21] "RiseVision," [Online]. Available: https://help.risevision.com/hc/en-us/articles/115002868706-The-overview-how-does-this-work-. [Accessed 13 05 2018].

[22] "Install Windows service," [Online]. Available: https://docs.microsoft.com/en-us/dotnet/framework/windows-services/how-to-install-and-uninstall-services. [Accessed 16 05 2018].

[23] "WCFservice," [Online]. Available: https://docs.microsoft.com/en-us/dotnet/framework/wcf/whats-wcf. [Accessed 06 05 2018].

[24] "Ninject to web form," [Online]. Available: https://stackoverflow.com/questions/25046162/how-to-inject-dependencies-using-ninject-in-asp-net-webform?utm_medium=organic&utm_source=google_rich_qa&utm_campaign=google_rich_qa. [Accessed 27 05 2018].

Appendix 1 – Result

Photos below show that dashboard created during this project is in actual use by LinnSystems.