TALLINN UNIVERSITY OF TECHNOLOGY
Department of Software Science

Ksenia Gulova 123681

AUTOMATION AND INTEGRATION OF A DOWNTIME NOTIFICATION SYSTEM

Bachelor's thesis

Supervisor: Deniss Kumlander
Doctor of Philosophy
in Engineering

Tallinn 2017

TALLINNA TEHNIKAÜLIKOOL
Tarkvarateaduse instituut

Ksenia Gulova 123681

SEISUAJA TEAVITUSSÜSTEEMI AUTOMATISEERIMINE JA INTEGREERIMINE

bakalaureusetöö

Juhendaja: Deniss Kumlander

Tehnikateaduste doktor

Tallinn 2017

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Ksenia Gulova

21.05.2017

Abstract

When an outage occurs with the system, it is essential to notify customers of this in a quick and timely manner. Status pages are used for this purpose, as they display the current status of the system, notify customers when an issue occurs and inform them when it is resolved.

The aim of this bachelor thesis is to select a simple and reliable status page service, according to the company’s requirements and develop a synchronizing application to automatically update statuses on the Status Page.

In order to simplify the status page management, this paper will outline the creation and design of a web application, which will monitor the state of the system and automatically submit the current system state to the status page.

The result of this thesis will be a working web application which will:

- Update the status page data with tests received from the company's monitoring system in specific time intervals
- Allow to easily set up a synchronization process
- Allow managing the system through the browser
- Show a graphical display of company's services availability

This thesis is written in English and is 47 pages long, including 5 chapters, 21 figures and 7 tables.

Annotatsioon

Seisuaja teavitussüsteemi automatiseerimine ja integreerimine

Kui süsteemis toimub katkestus siis on oluline teavitada kliente sellest kiirelt ja õigeaegselt. Selleks kasutatakse olekulehti, sest nad kajastavad süsteemi hetkeseisu ja informeerivad kliente kui on tekkinud probleem ja millal saab see lahendatud.

Selle bakalaureusetöö eesmärgis on valida lihtne ja usaldusväärne olekulehe teenus mis vastab kliendi määratud nõuetele, ning looda veebirakendus, mis automatiseerib seisakutest teavitamise süsteemi.

Käesolev töö annab ülevaate veebirakenduse loomisest ja disainimisest, mida kasutatakse et lihtsustada olekulehe juhtimine ja mis jälgib süsteemi ja automaatselt edastab olekulehele andmed süsteemi olekust.

Selle töö tulemusena on töötav veebirakendus mis:

- Uuendab olekulehe andmed mis on saadud ettevõtte seiresüsteemi testidest kindla aja tagant
- Võimaldab sünkroonimisprotsessi kergesti seadistada
- Võimaldab juhtida süsteemid brauseri kaudu
- Näitab ettevõtte teenuste kättesaadavust graafilisel kujul.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 47 leheküljel, 5 peatükki, 21 joonist, 7 tabelit.

List of abbreviations and terms

IDE	Integrated development environment
API	Application programming interface
AJAX	Asynchronous Javascript and XML.
CSS	Cascading Style Sheets
SQL	Structured Query Language
HTML	HyperText Markup Language
IoC	Inversion of Control
SOLID	single responsibility, open-closed, Liskov substitution, interface segregation and dependency inversion
Atlassian	Enterprise software company
JSON	JavaScript Object Notation, data-interchange format
MVC	Model–view–controller
UI	User interface
CPU	Central processing unit
REST	Representational state transfer
PDF	Portable Document Format
Framework	A reusable set of libraries or classes for a software system
AngularJS	JavaScript-based web application framework
Grid	Graphical control element that presents a tabular view of data
UML	Unified Modeling Language, a software modeling language
OWIN	Open Web Interface for .NET

Table of Contents

1 Introduction....11
2 Preparation....13
2.1 Overview of Status Page systems....13
2.1.1 Choosing the most suitable Status Page service....14
2.2 Overview of the monitoring system....15
2.3 Application requirements....15
2.3.1 Functional requirements....15
2.3.2 Non-functional requirements....17
2.4 Choice of tools....17
2.4.1 Framework....17
2.4.2 Libraries and other tools....19
2.4.3 Third party tools....21
3 Design....22
3.1 Base structure....22
3.1.1 Use case....23
3.1.2 User interface....23
3.1.3 User interface wireframes (Mockups)....24
3.1.4 Database....26
3.1.5 Business Logic....28
3.1.6 Login into system....29
4 Implementation....30
4.1 Base classes....30
4.2 Castle Windsor....32
4.3 Client-side....34
4.4 Synchronization....36
4.5 API....40
4.6 Authorization....40
4.7 Refactoring....42

4.8 Technical debt....43
4.9 Future plans....44
5 Summary....46
5 Kokkuvõte....47
References....48

List of Figures

Figure 1: Design diagram.....22
Figure 2: Use case diagram.....23
Figure 3: Mockup. List of status page components.....24
Figure 4: Mockup. Monitors settings.....25
Figure 5: Database tables structure.....26
Figure 6: Business logic structure.....29
Figure 7: Components subsystem structure.....30
Figure 8: ComponentService structure.....31
Figure 9: Creation of the IoC container.....32
Figure 10: Windsor Installer.....33
Figure 11: Custom controller factory.....34
Figure 12: Razor view example - Login page.....34
Figure 13: Custom validator example.....35
Figure 14: Grid subscribers.....35
Figure 15: Timer example.....36
Figure 16: Scheduling process.....37
Figure 17: Synchronization process for certain component.....39
Figure 18: Web request example - GetCurrentStatus.....40
Figure 19: OWIN configuration.....41
Figure 20 Log in process.....42
Figure 21: Replace Array with object example.....43

List of Tables

Table 1. Grid comparison.....20
Table 2. Synchronization general settings database table.....26
Table 3. Component database table to store data for downloaded components from status page.....27
Table 4. Component status database table to to store available statuses for components 27
Table 5. Monitor database table to store monitors downloaded from monitoring system 27
Table 6. MonitorSyncSettings table to store monitor settings for synchronization.....27
Table 7. ComponentMonitor table to connect monitors with component.....28

1 Introduction

Any system may be unavailable at some point, whether it is a planned maintenance or a major outage. Regardless, it is extremely important to inform users about downtime as quickly as possible, in which case customers are not going to panic, because they will already be informed. In most cases, downtime is unpredictable, and it may be difficult to choose the best way to let customers know what is happening. Instead of frightening the users, the company should organize customer communication during outages in such a way that every person using the system is aware that the issue is known and that a solution is already being anticipated.

The outage communication can be represented to customers in many ways. The main goal is to show customers a maintenance page. However, not each incident in the system should be disclosed to everyone. The goal is to display those that affect customers. Instead of wasting time figuring out if an incident is major or not at all times, a reliable system may be configured by the company to inform of an outage. Such systems are called status pages.

The market has a huge number of applications which enable the desired objectives, however they offer to change statuses manually which does not allow to notify users about the problem as soon as possible. It was decided to choose an existing service to make a status page, which allows easy automation through an API.

The objective of this work is to choose a status page service according to company's requirements and develop a synchronizing application to automatically update statuses on the Status Page.

This first part is dedicated to select a status page application among available alternatives, that would meet the following requirements, set by the company:

- Open and extensive API

- Friendly design
- Subscription possibility for email notifications

An automation web application will be created as part of this thesis, which will be connected to the selected status page via an API and will retrieve and analyse data from an external monitoring system in order to identify outages and update the statuses and create new incident messages on the status page automatically. This web application will include:

- The ability to set a synchronization period, which will check the status of the system at set time intervals
- Status identification system (Partial Outage, Performance Issues, Major, etc.). It should be able to launch a number of tests on each of the status page components to determine the exact efficiency of the system. Basing on the test results, the status is defined by the original identification algorithm. Each of the monitoring system components must also have a priority level assigned to them to identify the severity of the problem.
- Login system based on status page authorization service

The result of this thesis is a working web application which will automatically update the status page with new information quickly and efficiently.

2 Preparation

The author was given the task to select an existing status page service and automate it according to the results of tests, performed by a monitoring system. The synchronization application should be very simple and reliable, it should function as a bridge between the Status Page and the Monitoring application. The data requested from the monitoring system needs to be filtered and sent back to the status page.

2.1 Overview of Status Page systems

The author of this thesis decided to find an existing status page service that meets the requirements provided by the company. Below is a short overview of found services.

Sorryapp is a simple status page system for a reasonable price which supports Email notifications and has a Slack and Twitter integration. It allows to configure custom styling and domain [1]. The downside of this system is that it has a monthly limit on the amount of API calls that can be made to it.

Status.io is another status page system with a customizable design, which allows users to inject their own custom HTML and JavaScript code, as well as CSS. It also has the ability to split system statuses and monitors geographically, showing the present state of the system in a particular location [2]. Compared to other systems, Statuis.io is rather expensive, with the least expensive options being very limited in terms of functionality and number of users.

Statuspage.io is a product of Atlassian, which is aimed at large businesses and enterprises. It has a large number of third party integrations, as well as the ability to display the status of each one of these integrations, however none of these integrations are used by the client company. One more factor that makes this system stand out is the ability to subscribe to a particular component, which allows clients to only receive notifications regarding components that affect them directly [3]. The author decided

against using this system, as it is not suitable for smaller businesses and the functionality offered is not required by the client company.

Statushub.io has a very user-friendly setup process. It integrates with a number of monitoring systems, such as Pingdom, VictorOps, etc. [4], however, the monitoring system used by the client company is not among the list of such integrations and the service cost is high, which is why this would not be a suitable system.

Cachethq.io is an open-source status page system, which is very multifunctional. It supports localization into ten different languages, and this amount is constantly growing. It offers a simple, yet powerful JSON API, which allows to create custom integrations. It also has extensive documentation which makes the implementation stage very simple. Another beneficial feature is that it is supported on mobile devices, as well [5].

2.1.1 Choosing the most suitable Status Page service

The company had a list of requirements, which needed to be met when choosing a status page system:

- Low cost
- Email notifications
- Subscriptions
- Scheduled incidents
- Metrics
- A friendly design
- An open API
- Access for multiple team members
- Integration with the Monitis monitoring system

The author presented several services, which were described earlier, to the customer, provided an analysis and pointed out strengths and weaknesses of each suitable application.

As none of the systems supported an integration with the Monitis monitoring system, however, the remaining requirements were met by all of the reviewed systems, the

choice was made towards the least expensive application of the list. It was decided that the open-source CachetHQ system will be chosen and a custom integration with Monitis would be created manually by the author of this thesis.

2.2 Overview of the monitoring system

To determine the current status of a component, the automation program will reference the results of the tests of the external monitoring system. Monitis will be used as the monitoring system, since company is already using this application.

Monitis is a powerful monitoring platform, which allows to monitor all types of servers, to test CPU, disk space, http statuses, service statuses, full page load and transactions [6].

It has an easy-to-use dashboard and allows to get all critical information about a system. Monitis provides the tools for continuous monitoring of the whole system, has a dashboard section which shows all incidents and failed monitors. It has informative, clear API which allows to easily get information about system tests and their results.

2.3 Application requirements

This chapter describes the requirements that the application should meet. It is divided into two parts: functional and non-functional requirements. Since this project was developed by the order of the company, the author has composed this list in accordance with the request of the product owner.

2.3.1 Functional requirements

1. Component mapping
    1. One status page component may be connected to multiple monitors.
    2. If a component does not have any mapped monitors, it will be ignored during synchronization
2. Automation settings

1. It should be possible to enable and disable the synchronization for each component.
2. It should be possible to set up statuses for each mapped monitor, which will be connected to the component on the status page in case of test failures. (Performance issues, Partial Outage, Major Outage)
3. Each component should have a ‘max failed count’ setting. If the test fails once, it may be a short outage, which should not be shown on the status page. It should be possible to configure the count of failed tests, which, when reached, will update the status on the status page and create an incident report.
4. Add the possibility to set up incident messages for different outages and messages for resolved issues.
5. The synchronization settings may be applied to a single monitor, as well as a set of monitors, as the severity of the problem may differ based on the amount of failed monitors. This will also speed up and simplify the system setup process.

3. Synchronization
    1. In case of an outage:
        1. The status of the component should be automatically updated on the status page.
        2. The status is set from the monitor settings.
        3. If multiple monitors mapped with a component exceed the max failed count, a status with a higher priority will be chosen. (for example major instead of partial)
        4. New incident should be automatically created on the status page with an incident message set up in the settings.
    2. In case the existing issue is resolved:
        1. Update the status of the component to “OK”

2. Send a message that the issue is resolved
3. Reset the current failed count

4. Scheduling
    1. It should be possible to choose the synchronization frequency. (5, 10, 30, 60 minutes)
    2. It should be possible to start synchronization immediately by clicking a button.
5. Login system

2.3.2 Non-functional requirements

1. User interface language should be English because the main office of the company is located in the United Kingdom.
2. It must be a web application specifically, as it is not limited by hardware requirements.
3. Programming language – C#. As all apps in the company are written in C#, as such other languages are not suitable for the application.
4. Software Framework – .NET Framework 4.5, which is already used by the client.
5. Web browser support: Google Chrome, Mozilla Firefox, Internet Explorer, Microsoft Edge.
6. IDE - Visual Studio 2015, as all developers within the company work with this IDE.
7. Version control – Mercurial, as it is a standard version of control system, used by the company.

2.4 Choice of tools

2.4.1 Framework

The main reason for choosing .NET Framework 4.5 [7] was the requirement of the company. As all developers in the company have knowledge and experience of using this framework, they can easily understand the code and make changes if needed.

.NET is a huge framework with a lot of subsystems, such as ASP.NET, WPF, Silverlight etc. WPF and Silverlight both have an ability to make rich UI easily. However, WPF only allows to create a desktop application, as such, the company requires a web application. Silverlight is a web browser plugin, however, a lot of browsers do not support it anymore. (e.g. Chrome and Mozilla Firefox already stopped working with it) The reasonable solution is to use the ASP.NET web application framework, which gives the ability to build dynamic web sites. It supports four main technologies – Web Forms, MVC, Web API, and ASP.NET Core, which can be combined together or used separately [8]. All of them support adding third party libraries, separating user interface from application business logic and client-side scripting.

Web forms has a large amount of prepared user interface controls, which allows developers to build extensive UI very quickly. This technology implies the use of event-driven programming, however a huge number of event handlers makes application extremely difficult for unit testing [9]. Additional downsides of Web forms are shown when the project contains a number of server-side web controls on a page, this makes the View State very large. Web forms is the oldest technology out of the ones, researched by the author, and is not used commonly in new projects any longer.

ASP.NET Core is the newest Microsoft framework, it is fully open source, moreover it is a cross-platform technology, which allows developers to build their applications on Windows, Mac and even Linux. Due to the fact that multi-browser support will be sufficient for this project, such cross-platform technology is not required [10]. The author has decided against using this framework, as it is a rather new piece of technology, which will require more time to adapt within the company, and will require more skill to support in the future, and it also does not meet the time constraints, set by

the client. Furthermore, being quite new, this framework has not been thoroughly tested and may present further issues.

ASP.NET MVC is a framework which is completely based on MVC architecture, provides clean separation of modules, and it supports test driven development. Additionally, it enables full control over rendered HTML. This framework has a large amount of extensions and a wide variety of third party libraries. It also presents a higher level of stability, when compared to such frameworks, as ASP.NET Core [11].

Web API is an open source platform, that is ideal for building REST-ful applications over the .NET framework. It allows to build HTTP services, which may be consumed by a large number of clients, such as web-browsers, smartphones, and other devices. It is quite lightweight in architecture, which makes it a good choice for devices with limited bandwidth, such as smartphones. It also supports such MVC features as routing, controllers, action results and model binders. Additionally, it has the ability to accept and generate content, which may not be object-oriented, such as images and PDF files [12].

The author was choosing between two technologies: Web API and MVC. While the author agrees that using Web API, coupled with AngularJS would be considered good practice, as it would allow to decouple the back-end from the frontend for sustainable and easy to maintain development, the time constraints set by the client have not given the author enough time to research this technology. As a result, using this framework may have been more complex and time-consuming to configure this project, which is why the choice was made to use MVC, as it is fully testable, has a strong structure and low entry level.

2.4.2 Libraries and other tools

Newtonsoft.Json: [13] is a widespread and high performance JSON framework. It allows to serialize and deserialize any .NET object with JSON. It is a lightweight library, which is faster than other similar libraries.

In order to inject dependencies, the application uses the fully-functional open source inversion control tool **Castle Windsor** [14]. The .NET Framework also supports such

popular containers as StructureMap, Ninject, Autofac, Spring.NET, [15] however they are comparable and essentially aim at providing same services, so there was no need to make a deep investigation about when to use which one.

ADO.NET Entity Framework: is an object-relational mapper which supports the development of data-oriented .NET applications. The author has decided to use the framework as it allows to easily create data models and classes [16].

jQuery is a JavaScript library, which supports AJAX requests and event handling. It is extremely helpful for the front-end part of the application. The same functionality is provided by other frameworks, for example AngularJS or REST API. It was decided to use jQuery, as it is more lightweight than others and contains all the necessary functionality [17]. The application will have little interactivity on the page, so there is no need to use more complex frameworks. The usage of AngularJS and other alternative technologies is described in paragraph 4.8.

For data representation and editing, using a **data grid** was decided. As the market offers a lot of such tools, the choice was not simple. The author made a comparison of several popular grids: **Slickgrid [18], Kendo UI Grid [19], Datatables [20], Grid.Mvc [21], Mvcgrid [22].**

Table 1. Grid comparison

	Slickgrid	**Kendo UI**	**Datatables**	**Grid.Mvc**	**MVC-Grid.Net**
Open-source	Yes	No	No	Yes	Yes
Full documentation	Yes	Yes	Yes	No	Yes
Inline editing	Yes	Yes	Yes	No	No
Filtering	Yes	Yes	Yes	Yes	Yes
Sorting	Yes	Yes	Yes	Yes	Yes
Pagination	Yes	Yes	Yes	Yes	Yes
Cell formatting	Yes	Yes	Yes	Poor	Yes
Is supported by now	No	Yes	Yes	No	Yes
Razor support	No	Yes	No	Yes	Yes

The criteria are listed in the table above, based on their priority in a descending order. Performance was not taken into account in this comparison, as the application will not be working with large amounts of data.

Kendo UI and Datatables are powerful and flexible tools with a wide variety of options, however their use is not free of charge, therefore, the author was choosing between Slickgrid, Grid.Mvc and MVCGrid.Net. The main advantage of MVCGrid.Net and Grid.Mvc is that they were developed for ASP.NET MVC and support Razor technology. It was decided to abandon the use of Grid.Mvc, as it has poor documentation and has limited functionality. MVCGrid.Net is a simple good-looking grid, which supports basic data operations, however it is not a client-side tool, all grid configurations should be placed in the root of an application. SlickGrid is a JavaScript spreadsheet with a lot of functions. It has a poor user interface, but is fully customisable. Despite the fact that SlickGrid is no longer supported by its author, it is open source and is being developed further by other people. It was decided to use this data grid as it is very configurable, fast and lightweight.

2.4.3 Third party tools

In order to create class diagrams, the author has chosen the open source tool, called **NClass** [23], as it allows to easily create UML class diagrams and has C# language support. This program has simple, but powerful functionality for creating sophisticated, professional class diagrams. It allows to generate code from a model and supports language specific elements, such as properties, enums, delegates, etc.

Microsoft Visual Studio [25] was chosen as the IDE, as it supports .NET framework and the author is familiar with it, due to using it in day-to-day tasks at work.

The author used the source control tool, called **Tortoise HG** [24] to manage the working repository. In deciding to use this tool, the author’s decision was based on the fact that this tool has originally been created for Mercurial distributed revision-control systems, using which was required by the company. Also, in the author's own personal opinion, it is more convenient of a tool to use, compared to other similar programs, such as Source tree or VisualHG.

In order to create workflow diagrams and wireframes, the author has decided to use **LucidChart** [26], as it has straightforward usability, and an integration with Google Docs. Another key factor was that LucidChart is the main application that is used in the client company. The author also has experience working with this application.

3 Design

3.1 Base structure

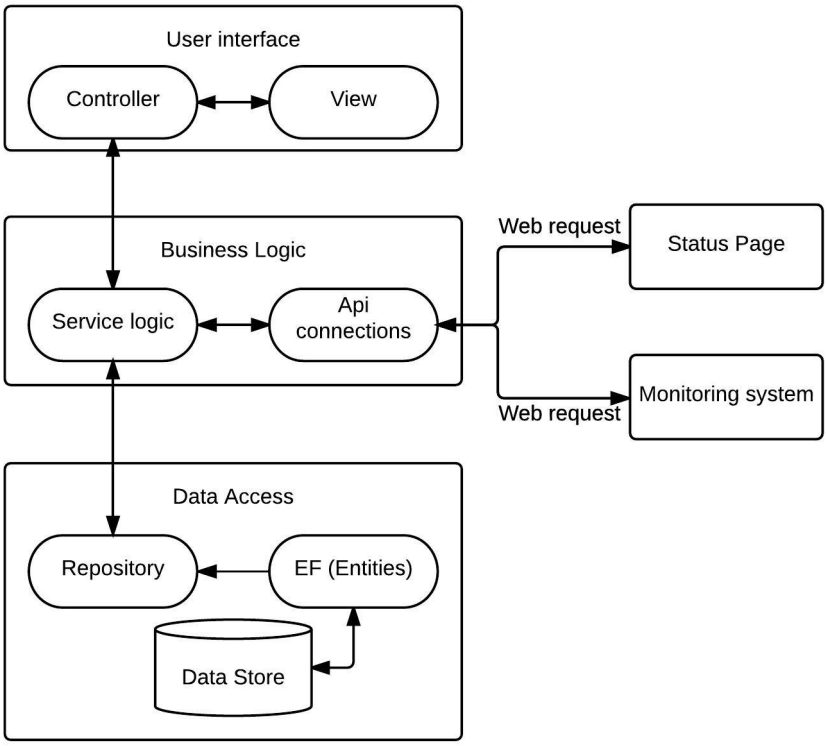

Figure 1: Design diagram

The application is separated into 3 layers: User interface (Front-end), Business logic (Back-end) and Data Access. The front-end layer consists of user views and controllers. A view displays the application's user interface and sends user actions to a controller, in it's turn, the controller then handles retrieved user interaction and selects the appropriate view to render. In order to retrieve data from a view, the controller refers a business logic layer, which contains API connection classes to communicate with external sites and service logic classes to manipulate the database and API data. This layer is aimed at getting data from sources, operate on it and return it back to the controller. The Data

Access layer contains the Repository classes for data operation using the Entity framework context, which allows to query, insert, update and delete data, using common language runtime objects (known as entities). The entity framework materializes data returned from the database as entity objects, tracks changes and sends them back to the data storage.

3.1.1 Use case

The developed automation system has one main actor – the team member, whose responsibilities are to configure and manage the automation process.

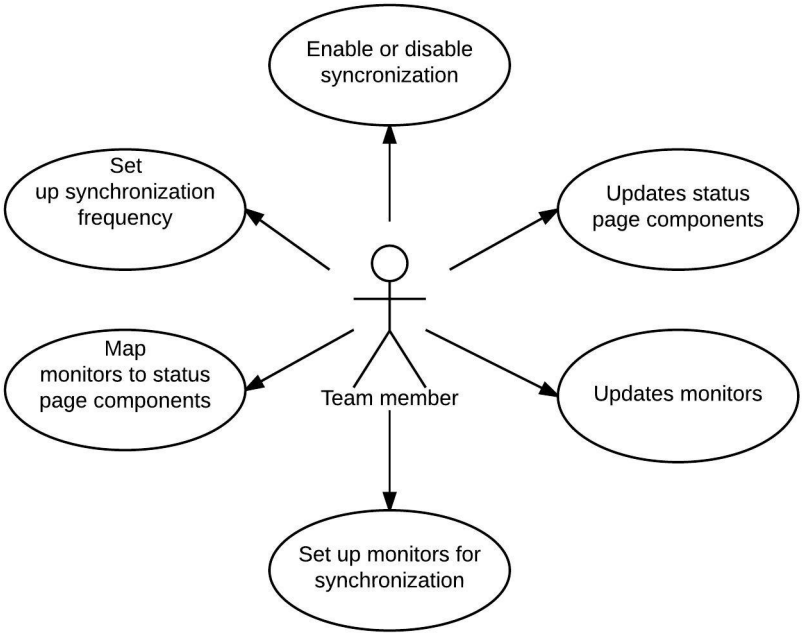

Figure 2: Use case diagram

3.1.2 User interface

User interface is written using HTML, JavaScript and Razor programming syntax and consists of views, which are responsible for generating and rendering HTML from the data of a model. It represents the file extension called **cshtml [27]** that refers to the razor view engine and consists of the HTML code and C# code, which is compiled on the server before the page is being loaded into the browser. It is also connected to java script files to communicate with a controller through AJAX requests and CSS files for

describing the presentation of the page. Systems will contain three main views – the Login page, the Components page, the Monitors page and one additional view – the layout page to create reusable blocks of content, such as common scripts, the header and the footer.

3.1.3 User interface wireframes (Mockups)

A mockup is a representation of the product's appearance. It helps developers better understand what exactly is it that the customer wishes to see in the software. The main reason of using mockups is time and cost efficiency. Changing project functionality and design at a later stage of development may cause financial losses. Usually, these changes can not be avoided, but may be reduced.

The author used the **LucidChart** application to create wireframes.

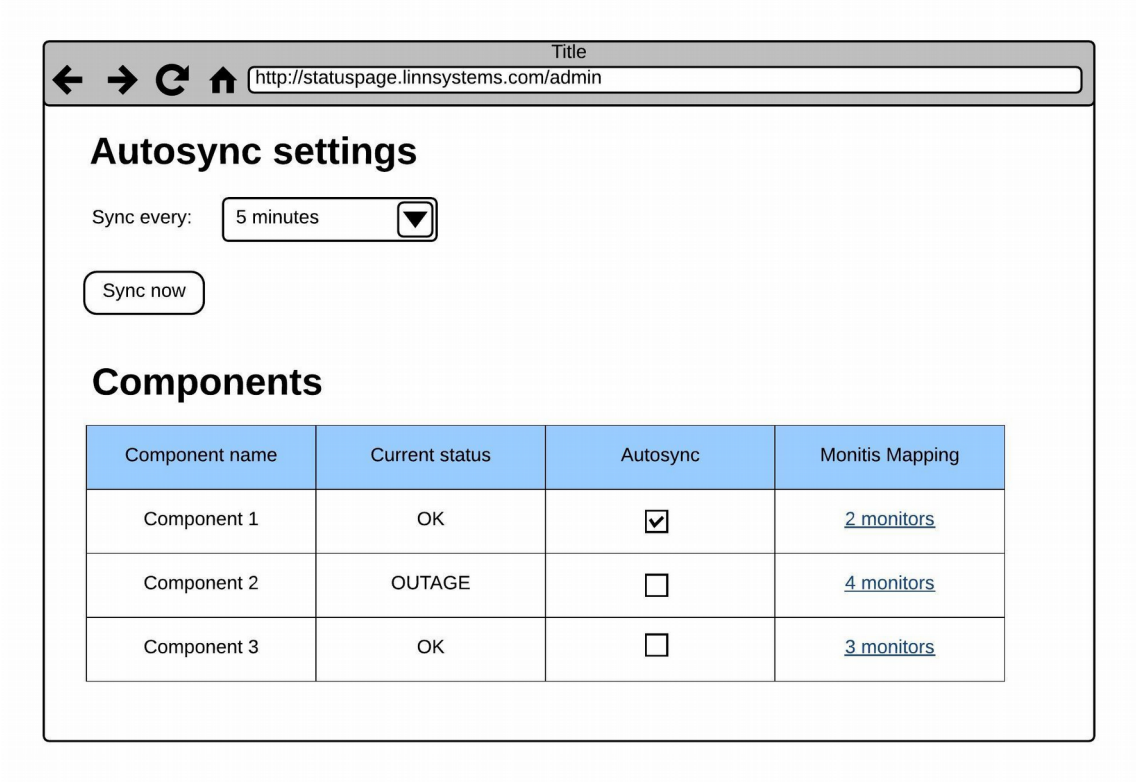

Figure 3: Mockup. List of status page components

This is the view that will be presented to the user after entering the system. General automation settings, where a user can choose the synchronization frequency and start the synchronization process immediately by pressing the 'Sync now' button, will be

presented in this view. The view also contains a list of status page components, where a user can enable or disable synchronization of each component separately. The ‘Current status’ column will show the status of the component, received during the last synchronization. The ‘Monitis mapping’ column will show the count of mapped monitors, clicking which will redirect the user to the monitors settings page.

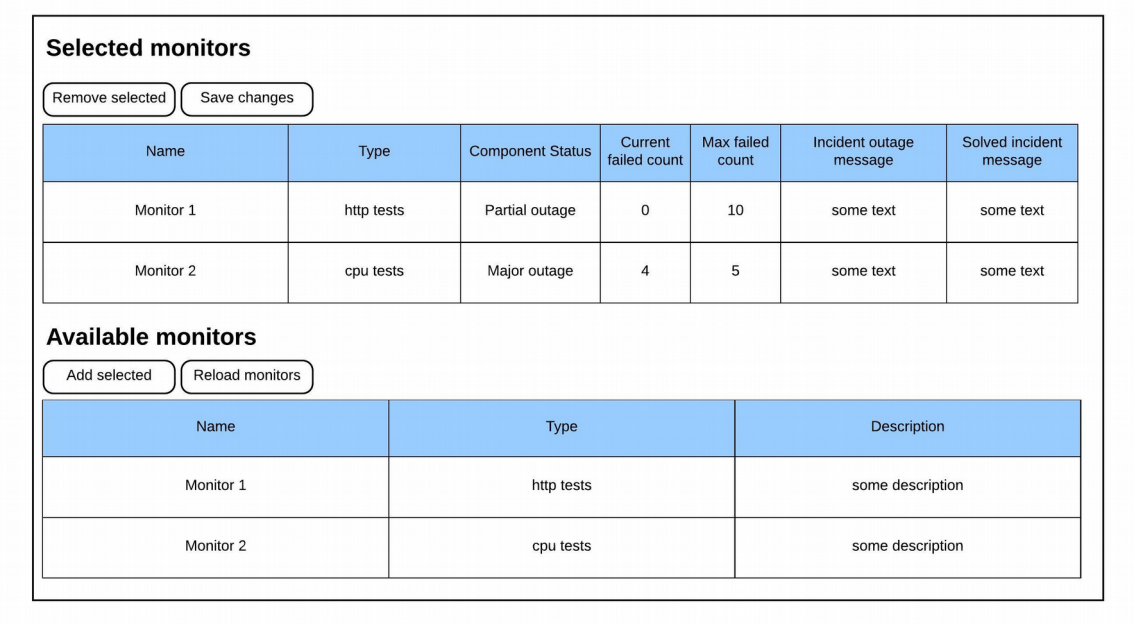

Figure 4: Mockup. Monitors settings

This view will display the following two lists: ’Selected monitors’, which are mapped to a certain component; and ’Available monitors’, which display the monitoring system tests. The user may select multiple rows from the list of available monitors and add them to the selected list by clicking on a button called ’Add selected’. In the same way, the user will be able to remove selected monitors by selecting needed rows and pressing the ‘Remove selected’ button. The selected monitor has the following editable columns:

- Component status – indicates a status, which will be sent to the status page in case of an outage. It is a drop down list of available status page statuses.
- Incident outage message – in this column, the user can specify the text of the incident message, which will be sent to the status page in case of an outage
- Solved incident message - a message which will be sent to the status page when the incident has been resolved

3.1.4 Database

To store all necessary data, the system uses the **Microsoft SQL Server** relational database management system and the data access technology: **ADO .NET** (ActiveX Data Object for .NET). Access to the database can be obtained through the connection string, which is stored in the application configuration file called web.config. The Database consists of six tables: *AutoSyncSettings, Component, ComponentStatus, Monitor, ComponentMonitor* and *MonitorSyncSettings.* Below is a diagram of the database where the table relationships are displayed, and the description of each individual table.

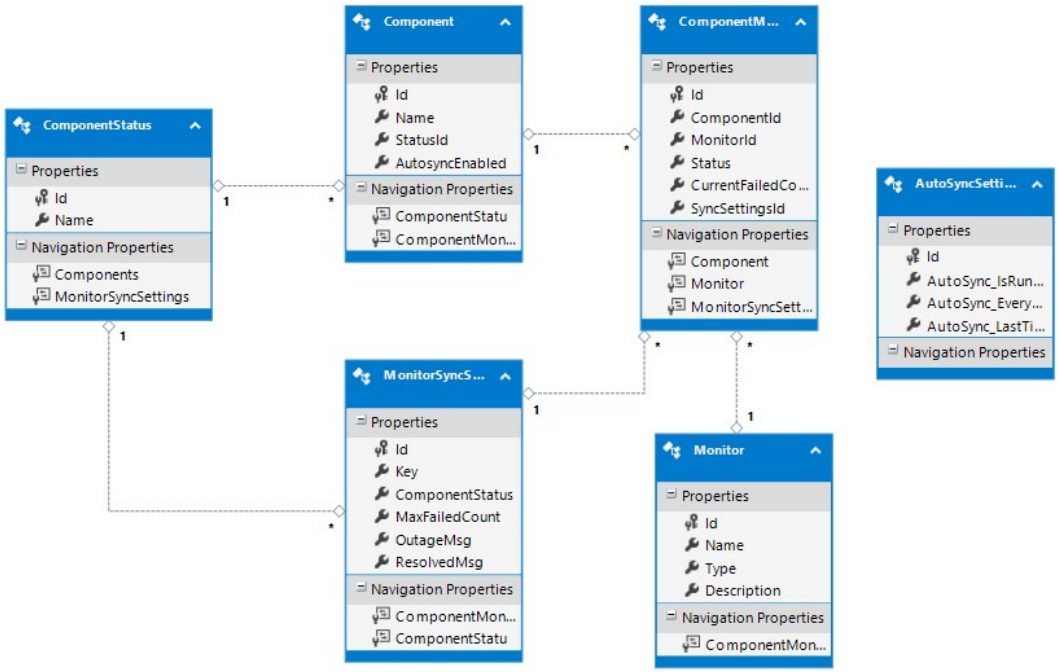

Figure 5: Database tables structure

Table 2. Synchronization general settings database table

AutoSyncSettings		
Column Name	**Type**	**Description**
Id	Int	Primary key
AutoSync_IsRunning	Bit	Specifies if synchronization is running at the moment, not null
AutoSync_EveryMinutes	Int	Synchronization frequency, not null

AutoSyncSettings		
Autosync_LastTime	Datetime	Datetime when last synchronization process ended, may be null

Table 3. Component database table to store data for downloaded components from status page

Component		
Column Name	**Type**	**Description**
Id	Int	Primary key
Name	Nvarchar(200)	Name of a component, not null
StatusId	Int	Foreign key to Component-Status primary key
AutosyncEnabled	Bit	Specifies is synchronization turned on for this component. Default value = 0

Table 4. Component status database table to to store available statuses for components

ComponentStatus		
Column Name	**Type**	**Description**
Id	Int	Primary key
Name	Nvarchar(50)	Status name, not null

Table 5. Monitor database table to store monitors downloaded from monitoring system

Monitor		
Column Name	**Type**	**Description**
Id	Int	Primary key
Name	Nvarchar(200)	Monitor name, not null
Type	Nvarchar(50)	Monitor type, not null
Description	Nvarchar(500)	Additional monitor info, may be null

Table 6. MonitorSyncSettings table to store monitor settings for synchronization

MonitorSyncSettings		
Column Name	**Type**	**Description**
Id	Int	Primary key

Name	Nvarchar(200)	Monitor name, not null
ComponentStatus	Int	Foreign key to Component-Status primary key. It shows which status should be sent to status page component in case of monitor failure
MaxFailedCount	Int	Failed tests count after which component will be updated on a status page. Default = 0
OutageMsg	Nvarchar(500)	Outage message which will be sent to a status page in case of failure. May be null
ResolvedMsg	Nvarchar(500)	Resolved message which will be sent to status page when incident was resolved. May be null

Table 7. ComponentMonitor table to connect monitors with component

ComponentMonitor		
Column Name	**Type**	**Description**
Id	Int	Primary key
ComponentId	Int	Foreign key to Component primary key
MonitorId	Int	Foreign key to Monitor primary key
MaxFailedCount	Int	Failed tests count after which component will be updated on a status page. Default = 0
Status	Bit	Specifies current status. Failed = 0, OK=1, not null
CurrentFailedCount	Int	Specifies how many times monitor result was failed in a row. Default = 0
SyncSettingsId	Int	Foreign key to MonitorSync-Settings primary key

3.1.5 Business Logic

This layer is divided into three logical parts:

1. Components logic: includes logic for getting, updating and configuring status page components.

2. Monitors logic: includes logic for getting, updating and configuring monitors

3. Synchronization: uses the monitor part to analyze the monitor's test results and the components part to send investigated data to the Status Page. The synchronization starts at configured intervals.

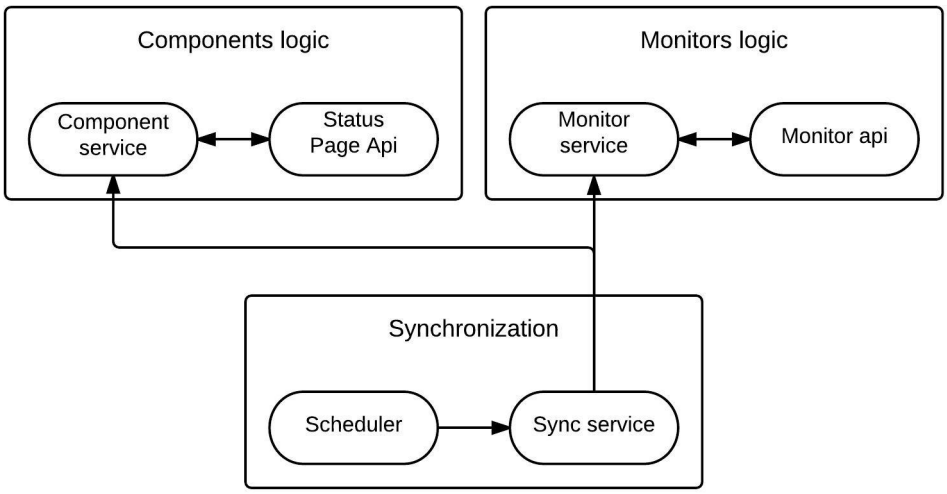

Figure 6: Business logic structure

3.1.6 Login into system

In accordance with the client's request, access to the application will only be granted to a certain number of people, who will be handling the setup and the configuration of the synchronization process. This makes it necessary to create a login page. After discussing this with the client, it was decided that user data would not be stored in the application, rather the status page user authentication is going to be used instead, as the application will be hosted on a secure server, which will only be accessible from the working machines within the company. This decision will eliminate the possibility of unauthorized access from outside of the company. For authorization will be used the OWIN Identity. The usage of the OWIN identity is described in paragraph 4.6.

4 Implementation

4.1 Base classes

The author has tried separating the application modules from each other and building a system which follows the SOLID [29] principles by using abstractions. Base classes of the component subsystem and their relationship to each other are described below. The same structure was used to build the Monitor subsystem.

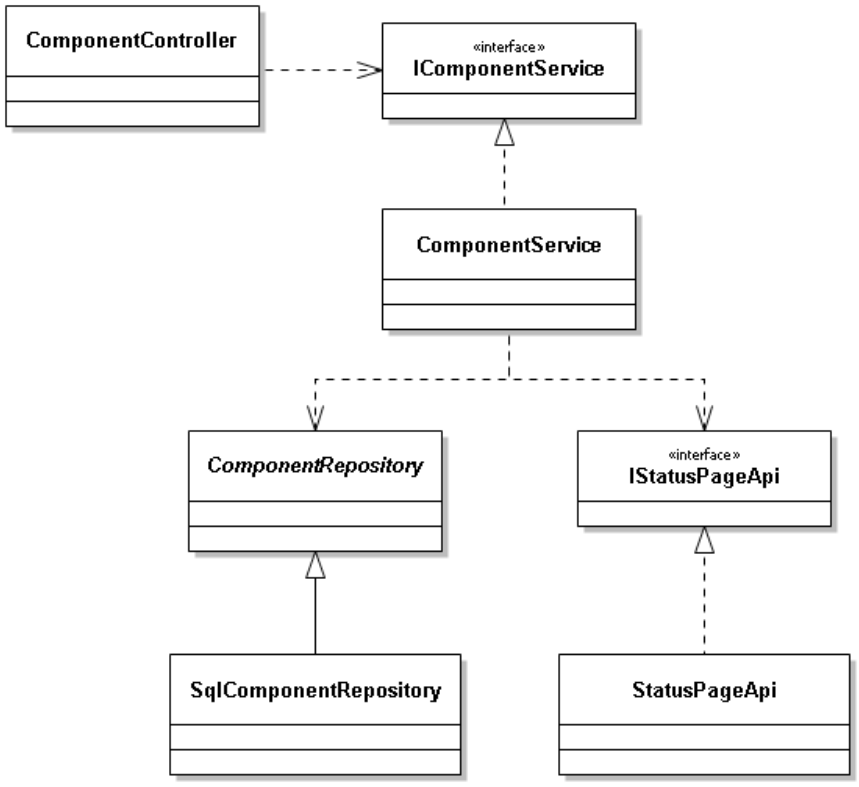

Figure 7: Components subsystem structure

- SqlComponentRepository is a part of the data access layer, it implements methods from ComponentRepository.

- ComponentRepository is a simple abstraction used for data access. It allows to separate the business logic layer from a specific data access implementation.

- StatusPageApi is a class which implements IStatusPageApi methods for Cachet status page service

- IStatusPageApi is an interface which specifies methods for getting and updating components from Status Page.

- ComponentService is a class which includes methods for components manipulations. It uses the IStatusPageApi interface to download and update components data from a StatusPage and ComponentRepository class to get component data from database, update and save it back. ComponentService uses the Constructor injection pattern to specify dependencies. This way, the component service delegates dependency management to the ComponentController.

```
public class ComponentService : IComponentService
{
    private readonly ComponentRepository _repository;
    private readonly IStatusPageApi _api;
    0 references
    public ComponentService(ComponentRepository repository, IStatusPageApi api)
    {
        if (repository == null)
        {
            throw new ArgumentNullException(nameof(repository));
        }
        if (api == null)
        {
            throw new ArgumentNullException(nameof(api));
        }
        _repository = repository;
        _api = api;
    }
```

Figure 8: ComponentService structure

The Component service implements a constructor injection pattern which allows to pass the needed dependencies into it externally. Since dependent fields are read only and are initialized and checked in the constructor, the developer who will use it can be sure that all references were made correctly and there is no need to check the fields for null in each separate method.

- ComponentController uses IComponentService and delegates dependency management to a composition root. As a result of such structure, controller does not know anything about data manipulations and can only use methods from the service interface, which means that it is fully separated from business logic and data access layers. Low coupling between modules allows to easily expand their functionality without affecting other modules.

4.2 Castle Windsor

The author used the Castle Windsor inversion of control container to inject the application dependencies. The container's goal is to convert and control the graphs of objects [33]. The container is registered in the composition root of the application (Global.asax – Application_Start() method)

```
var container = new WindsorContainer();
container.Install(new StatusSynInstaller());
var controllerFactory = new StatusSyncControllerFactory(container);
```

Figure 9: Creation of the IoC container

In order for the container to know how to compile the required types, including all of the dependencies, it is necessary to pass the information on the registration of abstractions with their specific types to it. The registration is written in the StatusSyncInstaller class, which is the implementation of the Castle Windsor interface, named IWindsorInstaller. It was used instead of writing the registration code in a static helper method to package the configuration in reusable modules.

```
public class StatusSynInstaller : IWindsorInstaller
{
    0 references
    public void Install(IWindsorContainer container, IConfigurationStore store)
    {
        container.Register(Component.For<ILoginManager>().ImplementedBy<LoginManager>());

        container.Register(Component.For<IStatusPageApi>().ImplementedBy<StatusPageApi>());
        container.Register(Component.For<ComponentRepository>().ImplementedBy<SqlComponentRepository>());
        container.Register(Component.For<IComponentService>().ImplementedBy<ComponentService>());

        container.Register(Component.For<IMonitoringApi>().ImplementedBy<MonitoringApi>());
        container.Register(Component.For<MonitorRepository>().ImplementedBy<SqlMonitorRepository>());
        container.Register(Component.For<IMonitorService>().ImplementedBy<MonitorService>());

        container.Register(Component.For<SyncRepository>().ImplementedBy<SqlSyncRepository>());
        container.Register(Component.For<ISyncService>().ImplementedBy<SyncService>());

        var controllers = Assembly.GetExecutingAssembly()
            .GetTypes().Where(x => x.BaseType == typeof(Controller)).ToList();
        foreach (var controller in controllers)
        {
            container.Register(Component.For(controller).LifestylePerWebRequest());
        }
    }
}
```

Figure 10: Windsor Installer

Since ASP.NET MVC automatically creates controller instances only for controllers with parameterless constructors, in order to implement the constructor injection in controllers and allow the container to connect the controllers to the application, it is necessary to create a custom controller factory, which inherits the System.Web.Mvc.DefaultControllerFactory class and overrides the GetControllerInstance method. The application will call the GetControllerInstance() method for each incoming HTTP-request and delegate the work to the container. The Resolve() method builds a complete graph, which should be used to manage this particular request, and returns it [31].

```
2 references
public class StatusSyncControllerFactory : DefaultControllerFactory
{
    3 references
    public IWindsorContainer Container { get; protected set; }

    1 reference
    public StatusSyncControllerFactory(IWindsorContainer container)
    {
        if (container == null)
        {
            throw new ArgumentNullException(nameof(container));
        }

        this.Container = container;
    }

    0 references
    protected override IController GetControllerInstance(RequestContext requestContext, Type controllerType)
    {
        if (controllerType == null)
        {
            return null;
        }

        return Container.Resolve(controllerType) as IController;
    }

    0 references
    public override void ReleaseController(IController controller)
    {
        var disposableController = controller as IDisposable;
        disposableController?.Dispose();
        Container.Release(controller);
    }
}
```

Figure 11: Custom controller factory

4.3 Client-side

The visual elements of the application were constructed using Razor and javascript libraries. The Razor view uses a model to represent, bind and validate data.

```
@Html.ValidationMessage("CredentialsError", new { @class = "text-danger" })
<div class="form-group">
    @Html.LabelFor(m => m.UserName, new { @class = "col-md-2 control-label" })
    <div class="col-md-10">
        @Html.TextBoxFor(m => m.UserName, new { @class = "form-control" })
        @Html.ValidationMessageFor(m => m.UserName, "", new { @class = "text-danger" })
    </div>
</div>
<div class="form-group">
    @Html.LabelFor(m => m.Password, new { @class = "col-md-2 control-label" })
    <div class="col-md-10">
        @Html.PasswordFor(m => m.Password, new { @class = "form-control" })
        @Html.ValidationMessageFor(m => m.Password, "", new { @class = "text-danger" })
    </div>
</div>
```

Figure 12: Razor view example - Login page

The application uses SlickGrid as the core visualization component. As this library does not support Razor, JavaSciprt files using AJAX requests were added to allow it to communicate with the controller. In order to configure the grid, the columns and such options as whether the grid is editable and whether it supports the addition of new rows, must be set up. SlickGrid uses editor functions, which are responsible for the cell state, to change the data within the cells. The library has a number of existing editors, however, a customized one may be applied if needed. Formatters are used in order to change the layout of the cell and to display data, whereas validators are used to check the input data. These may also be chosen from the list of existing ones, or may be personalised according to one's needs, which makes this grid very customisable.

```
function PositiveNumberValidator(value) {
    if (value <= 0) {
        return { valid: false, msg: "Value should be a positive number" };
    } else {
        return { valid: true, msg: null };
    }
}
```

Figure 13: Custom validator example

The library also has a large list of event handlers, which may be subscribed to and the grid may be set up to respond to a user's actions in a particular way.

```
function setSubscribers(gridData) {
    gridData.dataView.onRowCountChanged.subscribe(function (e, args) {
        gridData.grid.updateRowCount();
        gridData.grid.render();
    });

    gridData.dataView.onRowsChanged.subscribe(function (e, args) {
        gridData.grid.invalidateRows(args.rows);
        gridData.grid.render();
        gridData.grid.getSelectionModel().setSelectedRanges([]);
    });
}
```

Figure 14: Grid subscribers

4.4 Synchronization

The synchronization process occurs automatically in time intervals, set within the application settings. When the application is launched, the Scheduler class is created, which is responsible for running the synchronization process at the correct time by using the Timer system class. During the course of program execution, the synchronization frequency may be changed by the user, therefore, the Timer event occurs every minute and, based on the last settings, decides whether a new synchronization process should be started already. The synchronization process will not be launched until the time interval is passed, or if the previous synchronization process is still running.

```
timer = new System.Timers.Timer(60000);
timer.Elapsed += (e, v) =>
{
    if (HasSyncIntervalElapsed(_syncService.GetSyncSettings()))
    {
        _syncService.Synchronize();
    }
};
timer.AutoReset = true;
timer.Enabled = true;
timer.Start();
```

Figure 15: Timer example

The Timer.Elapsed event occurs when the timer interval elapses. The timer interval property is specified in milliseconds. The timer AutoReset property indicates whether the timer should raise the event each time the specified interval elapsed or only the first time after it elapses.

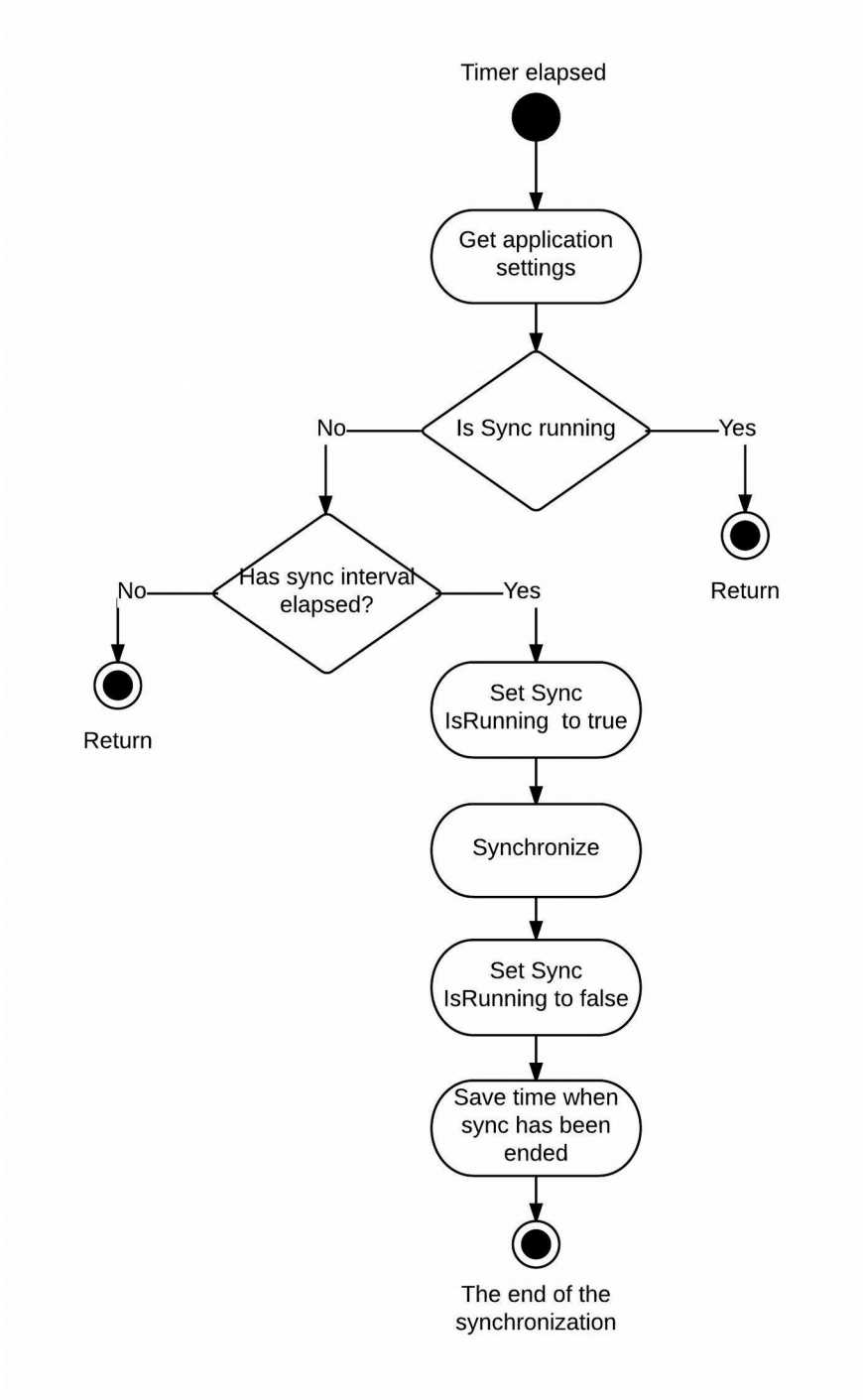

Figure 16: Scheduling process

The synchronization process is run for every status page compnent, which has the AutoSyncEnabled field set to True. During the synchronization process, the program requests the current status of the component from the status page, as well as the latest results of the component tests from the monitoring system. In cases where the current status, received from the status page is operational, the program updates the CurrentFailedCount value for all monitors, based on the test results. If the test is passed successfully, the CurrentFailedCount value is reset to zero, otherwise, it is increased by 1 and a failed monitor is returned. When there are multiple failed monitors, the program will pick the monitor with the highest priority. In this thesis, the priority is defined by the ComponentStatus monitor setting, which also decides, what status will be submitted to the status page in case of an outage. Afterwards, the component status is updated on the status page and an incident is created if the OutageMessage setting has been configured. If a component is already experiencing a problem and the result of the test has shown that it is now operational, the component status will be changed to operational on the status page, as well, and a new incident will be created, notifying that an issue has been resolved.

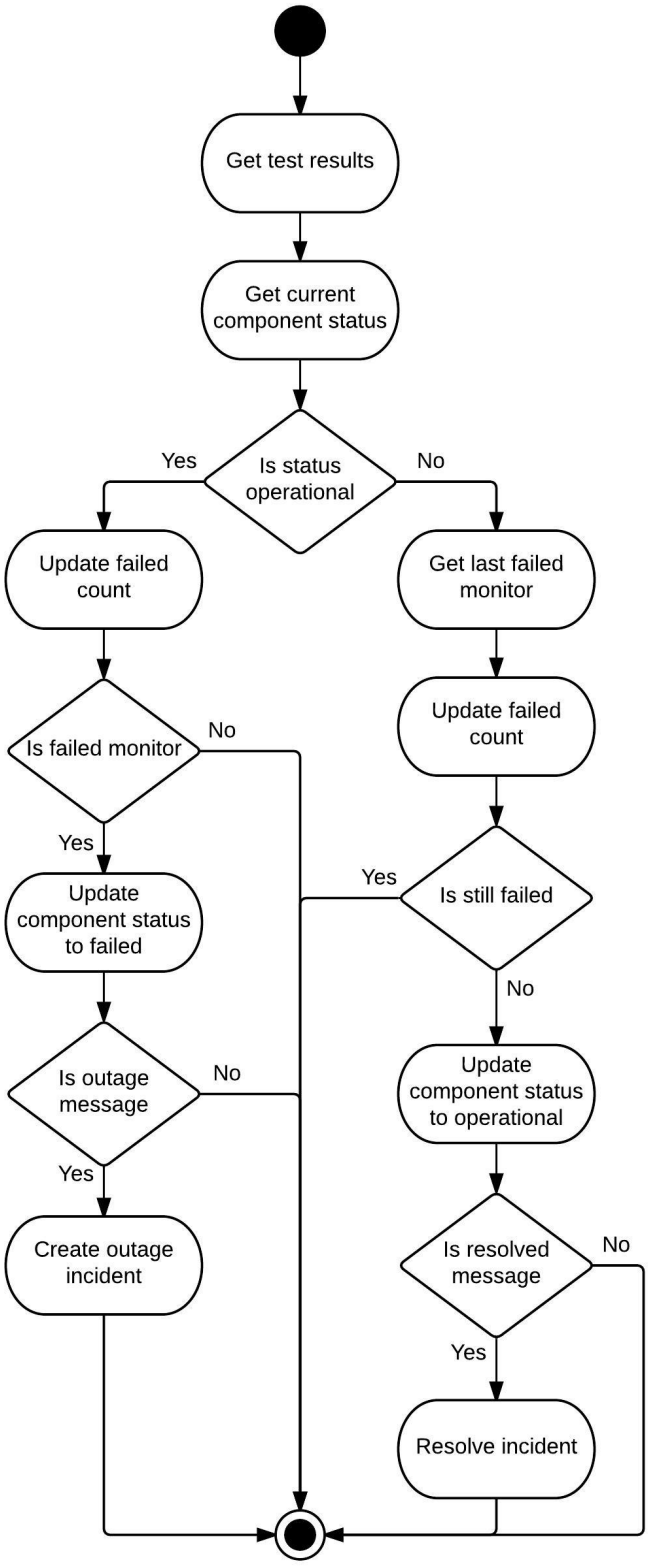

Figure 17: Synchronization process for certain component

4.5 API

To communicate with the status page and the monitoring system, classes that send web requests to external sites, receive a reply, parse it and return the formatted result, were created.

```
HttpWebRequest request = (HttpWebRequest)WebRequest.Create(url);
request.Headers.Add("X-Cachet-Token", token);

using (HttpWebResponse response = (HttpWebResponse)request.GetResponse())
using (Stream stream = response.GetResponseStream())
using (StreamReader reader = new StreamReader(stream))
{
    dynamic result = JObject.Parse(reader.ReadToEnd());

    if (result.data != null && result.data.status!=null)
    {
        return result.data.status;
    }
    else
    {
        throw new HttpListenerException((int)response.StatusCode, response.StatusDescription);
    }
}
```

Figure 18: Web request example - GetCurrentStatus

The classes are accessed only through their interfaces, which allows to easily replace the implementation of the API methods without affecting the rest of the application.

```
public interface IMonitoringApi
{
    2 references
    IEnumerable<Monitor> GetMonitors();
    3 references
    string GetStatusByMonitorId(int id, string type);
}
```

Access to the API methods can be obtained through the token. Though the Cachet API allows to send GET requests using only the username and the password, it was not used because sending authentication details in plain text is not secure.

4.6 Authorization

The first thing that a user sees when he runs the application is the login page, which is also the only accessible page for anonymous users. To ensure secure access to other pages of the application, the Microsoft.AspNet.Identity.Owin authorization system was used. When a user logs into the system, the application sends a request to the status page to make sure that the user has access to the system and calls the OWIN Cookie Authentication middleware to generate a cookie, so when a valid cookie is returned, the application will set the logged in user principal to HttpContext.User, so the rest of ASP.NET pipeline will know what user is authenticated, then it automatically redirects to the main page with the status page components. To initialize the OWIN identity components, a Startup class was added, which will automatically be located and initialized by the OWIN host [32].

```
public class Startup
{
    0 references
    public void Configuration(IAppBuilder app)
    {
        ConfigureAuthentication(app);
    }
    1 reference
    public void ConfigureAuthentication(IAppBuilder app)
    {
        app.UseCookieAuthentication(new CookieAuthenticationOptions
        {
            AuthenticationType = DefaultAuthenticationTypes.ApplicationCookie,
            LoginPath = new PathString("/Account/Login")
        });
        AntiForgeryConfig.UniqueClaimTypeIdentifier = ClaimsIdentity.DefaultNameClaimType;
    }
}
```

Figure 19: OWIN configuration

When a user enters the username and password, the login page sends the POST request to the AccountContoller, which validates the provided credentials and logs the user in. To check that the user actually has access to the system, the controller sends request to the login manager class, which is connected to the status page API. If the credentials are valid, the ClaimsIdentity object is created with information about the current user, then the controller registers the identity in the OWIN authentication manager. This sets the authentication cookie on the client. Finally, the application redirects the user to the main page.

```
if (_loginManager.SignIn(model.UserName, model.Password))
{
    var claims = new List<Claim>();
    claims.Add(new Claim(ClaimTypes.NameIdentifier, model.UserName));
    claims.Add(new Claim(ClaimTypes.Name, model.UserName));
    var identity = new ClaimsIdentity(claims, DefaultAuthenticationTypes.ApplicationCookie);

    AuthenticationManager.SignIn(
        new AuthenticationProperties {
            IsPersistent = true,
            AllowRefresh = false,
            ExpiresUtc = DateTime.UtcNow.AddMinutes(20)
        }, identity);

    return RedirectToAction("Index", "Home");
}
```

Figure 20 Log in process

4.7 Refactoring

During the development stages of the project, it may prove difficult to write well-structured code straight away, without the necessary experience. Therefore, the author has spent part of the time on the refactoring of the code, which resolved the following issues:

1. In development, long methods have appeared in the code, which impaired it's readability. The author had improved upon the code in a way that each method would only be responsible for performing a single action by using the Extract method to achieve this.

2. Instead of small objects, primitive fields have been used in certain parts of the code. In order to organize code better, the author used the Replace Data Value with Object and Replace Array with Object methods. For example, the Monitors View used two separate collections in order to display data - SelectedMonitors and AvailableMonitors. In order to avoid using Primitive Obsession code, the author unified these two collections into one.

```
3 references
public class MonitorViewModel
{
    1 reference
    public IEnumerable<MonitorModel> SelectedMonitors { get; set; }
    1 reference
    public IEnumerable<Monitor> AvailableMonitors { get; set; }
}
```

Figure 21: Replace Array with object example

3. Data Clumps and code duplication. In the client side of the application, two different views were using similar grids, which had lead duplicated code to appear in early stages of development. This was resolved by moving common methods into a separate class GridHelper.js.

4.8 Technical debt

1. **Angular JS** - The author had only recently started studying AngularJS and agrees that it's use is more beneficial in the sense that it has the ability to make the application more structured, and that it has virtually replaced such libraries as jQuery. After becoming more proficient with this technology, the author hopes to implement this knowledge and improve the application by changing it's structure using AngularJS, instead of jQuery. jQuery code is hard to maintain and in using AngularJs, which is more component-oriented, the author hopes that it will lead to a more maintainable application, which will make implementing new features less time-consuming [35].

2. The author understands that using web services has a lot of benefits, the most important of which being it's ability to work outside of private networks, which offers developers an unpatented route to their solutions. As a result, the developed services are longer lasting. Thanks to the use of standards-based communication methods, web services are independent of the platform that they are launched from, and they let the developers use whichever programming language they prefer. Their deployability also make them an important asset, as they are deployed over standard Internet technologies, making it possible to deploy them to servers even over their firewall. They are also very secure, as

they include built-in underlying security, such as SSL. Using web services has not been done in this project, as the author has little experience in working with them. In the future, however, the author plans to migrate the API part of the application to using web services [36].

3. Improve user interface. Due to the fact that the client side of the application will only be used by a small number of people within the company, the design of the web application has not been a priority and the author has not paid due regard to the design of the application. The layout of the application must still be improved upon, however, in order to make the application easier to use.

4. The system was tested manually, however unit, automated and integration tests were not written in this release, due to the time constraints provided by the product owner. The author understands that tests are essential to simplify code refactoring and to increase maintainability of the application, so they have developed the system in such a way that in the future it would be easy to add tests. It will be done by the next release of the application.

4.9 Future plans

The current project is only the first version of the automated application. The project is going to be expanded upon with new functionality by the author. Throughout the development stages, the author has come up with a number of new ideas on how to improve the project. As this was outside of the boundaries of this project, as well as due to time constraints, it was not possible to implement these ideas at this state of the project. The author hopes, however, to add the following features in the near future:

1. Logging. As the application will be constantly running on the server, it is necessary to create a logging system, which will allow the user to control what is happening during the synchronization process. In case of an error, it will be easier to identify it's cause and eliminate the problem.

2. Expand the synchronization abilities. Make it possible to assign a priority level to each monitor. In the current state of the project, only a small number of

monitors has been added and at present, it is enough to be able to identify the severity of the problem based on statuses received from the status page. In the future, however, the amount of such tests may be increased drastically and such a synchronization system may not be sufficient.

3. Add new settings to speed up and simplify the system setup process:

    1. Add the possibility to apply synchronization settings to a set of monitors, as the severity of the problem may differ based on the amount of failed monitors.

    2. Add the support of incident templates. This is necessary in order to make the system setup process easier, as the incident message may apply to a number of different tests.

    3. Add the ability to enable and disable the customer notification system in the case of an outage

5 Summary

In the course of this thesis, a number of popular status page systems were analysed and one of them, **CachetHQ**, has been selected from the list to be used by the company. In order to automate the selected downtime notification system, a separate web application has also been created.

The current version of the application allows the team members to configure the automation process, which will effectively reduce the workload placed on the technical support department, as agents will not have to manage the downtime notification system manually. Also, owing to the developed web application, users will be notified timely in case of system outages. The resulting web application meets all of the functional and non-functional requirements and is developed in such a way that each developer in the company will be able to understand the system logic and seamlessly make changes to it's code if necessary.

The current version of the application is not the last release, throughout the development process the author got acquainted with new technologies and will apply them in the future to expand the capabilities of the application.

5 Kokkuvõte

Selle töö käigus olid analüüsitud mitmed populaarsed olekulehe süsteemid ja üks nendest, **CachetHQ**, oli valitud kasutamiseks ettevõttes. Selleks et automatiseerida seisakutest teavitamise süsteemi oli loodud eraldi veebirakendus.

Praegune rakendus versioon võimaldab meeskonna liikmetele seadistada automaatiseeritud protsessi, mis tõhusalt vähendab tehnilise toe osakonna töökoormust, sest töötajad ei pea haldama seisakutest teavitamise süsteemi käsitsi. Samuti tänu arendatud veebirakendusele kasutajad teavitatakse seisakutest õigeaegselt. Loodud veebirakendus vastab kõigile funktsionaalsetele ja mittefunktsionaalsetele nõuetele ja on välja töötatud nii, et iga arendaja ettevõttes suudab mõista süsteemi loogikat ja vajadusel koodi sujuvalt muuta.

Praegune rakenduse versioon ei ole viimane väljalask, kogu arendusprotsessi käigus on autor tutvunud uute tehnoloogiatega ja kohaldab neid tulevikus et laiendada rakenduse võimalusi.

References

[1] Overview of the Sorryapp status page. [WWW] https://www.sorryapp.com/product/status-page.html (01.05.2017)

[2] The Status.io application features. [WWW] https://status.io/features (01.05.2017)

[3] Overview of the Statuspage.io. [WWW] https://www.statuspage.io/tour (01.05.2017)

[4] The Statushub application features, [WWW] https://statushub.com/features (01.05.2017)

[5] The documentation of the Cachet application. [WWW] https://docs.cachethq.io/docs (01.05.2017)

[6] Overview of the Monitis application. [WWW] http://www.monitis.com/why-monitis (01.05.2017)

[7] Overview of the .NET Framework. [WWW] https://msdn.microsoft.com/en-us/library/zw4w595w(v=vs.110).aspx (05.05.2017)

[8] ASP.NET Overview, 2012 [WWW] https://docs.microsoft.com/en-us/aspnet/overview (05.05.2017)

[9] Introduction to ASP.NET and Web Forms, 2001. [WWW] https://msdn.microsoft.com/en-us/library/ms973868.aspx (08.05.2017)

[10] Introduction to ASP.NET Core, 2016. [WWW] https://docs.microsoft.com/en-us/aspnet/core/ (08.05.2017)

[11] ASP.NET MVC 5, 2014. [WWW] https://docs.microsoft.com/en-us/aspnet/mvc/mvc5 (08.05.2017)

[12] ASP.NET Web API, 2012. [WWW] https://docs.microsoft.com/en-us/aspnet/mvc/mvc5 (08.05.2017)

[13] Inroduction to Json.NET. [WWW] http://www.newtonsoft.com/json/help/html/Introduction.htm (10.05.2017)

[14] The documentation of the Castle Windsor. [WWW] https://github.com/castleproject/Windsor/blob/master/docs/README.md (10.05.2017)

[15] S. Hanselman. List of .NET Dependency Injection Containers (IOC), 2014. [WWW] https://www.hanselman.com/blog/ListOfNETDependencyInjectionContainersIOC.aspx (12.05.2017)

[16] Introduction to Entity Framework, October 2016, [WWW] https://msdn.microsoft.com/en-us/library/aa937723(v=vs.113).aspx (12.05.2017)

[17] jQuery library. [WWW] http://jquery.com (12.05.2017)

[18] SlickGrid. [WWW] https://github.com/mleibman/SlickGrid/wiki (15.05.2017)

[19] Kendo UI grid. [WWW] http://demos.telerik.com/kendo-ui/grid/index (15.05.2017)

[20] DataTables. [WWW] https://datatables.net/ (15.05.2017)

[21] Grid.Mvc. [WWW] https://gridmvc.codeplex.com/ (15.05.2017)

[22] MVCGRid.Net. [WWW] http://mvcgrid.net/ (15.05.2017)

[23] NClass application. [WWW] http://nclass.sourceforge.net/ (16.05.2017)

[24] TortoiseHG application. [WWW] https://tortoisehg.bitbucket.io/ (16.05.2017)

[25] Visual Studio IDE User's Guide. [WWW] Available: https://msdn.microsoft.com/en-us/library/dd831853.aspx (16.05.2017)

[26] Overview of the LucidChart application. [WWW] https://www.lucidchart.com/pages/tour (16.05.2017)

[27] Introduction to ASP.NET Web Programming Using the Razor Syntax (C#), 2014. [WWW] https://docs.microsoft.com/en-us/aspnet/web-pages/overview/getting-started/introducing-razor-syntax-c (16.05.2017)

[28] Shivprasad koirala. SOLID architecture principles using simple C# examples, 2016 [WWW] https://www.codeproject.com/Articles/703634/SOLID-architecture-principles-using-simple-Csharp (17.05.2017)

[29] Entity Framework working with DbContext, 2016. [WWW] (17.05.2017) https://msdn.microsoft.com/en-us/library/jj729737(v=vs.113).aspx

[30] Hrusikesh Panda, Jess Chadwick, Todd Snyder, Programming ASP.NET MVC 4, October 2012, O'Reilly Media (20.04.2017)

[31] M. Seemann. Dependency Injection in .NET, 2011. (20.04.2017)

[32] P. Thiagarajan, R. Anderson. OWIN Middleware in the IIS integrated pipeline, 2013. [WWW] https://docs.microsoft.com/en-us/aspnet/aspnet/overview/owin-and-katana/owin-middleware-in-the-iis-integrated-pipeline (17.05.2017)

[33] M. Fowler. Inversion of Control Containers and the Dependency Injection pattern, 2004. [WWW] https://martinfowler.com/articles/injection.html (05.04.2017)

[34] J. Miller. Patterns in Practice: The Unit Of Work Pattern And Persistence Ignorance, MSDN Magazine, 2009 (25.04.2017)

[35] J. Danylko. Advantges of using AngularJS over jQuery, 2016. [WWW] https://www.quora.com/What-are-the-advantages-of-using-AngularJS-over-JQuery (18.05.2017)

[36] Using Web Services. [WWW] https://www.tutorialspoint.com/webservices/why_web_services.htm (18.05.2017)