Evaluation of Smart Home API with a developed application to a heating system
Abstract

Many companies are trying to be the central part of the smart home. By using public APIs you can integrate with other enterprise solutions and devices while you by yourself does not have to develop systems that are already on the market. This report consists of three parts in order to solve the integration of public APIs to an existing device in a heating system in smart home. The first part is a comparison of four different public APIs. The second part deals with the development of a couple system architectures for a general solution of how to add various public APIs in an easy and convenient way. The last part is an implementation of one system architecture to an existing system.

Keywords: Smart Homes, public API, comparison, architecture, MVC, SOA, C#.
Preface

I would like to thank NIBE AB for giving me this opportunity to help them with one of their problems. They have been very helpful by providing information, give me more insight into the heating system, the software development process and provide equipment for use and testing.

I would also want to thank the staff at LNU that have provided feedback and information in many different ways.

Lastly I want to thank my family for all support they have given.
## Contents

1 **Introduction** .......................................................... 1  
   1.1 Introduction / Background ....................................... 1  
   1.2 Problem definition .................................................. 1  
   1.3 Purpose and research questions .................................... 1  
   1.4 Scope / Limitation .................................................... 2  
   1.5 How this document is organized ..................................... 2  
   1.6 Technologies .......................................................... 2  
      1.6.1 OAuth2.0 ......................................................... 2  
      1.6.2 JSON ............................................................. 3  

2 **Method** ...................................................................... 4  
   2.1 Comparing APIs ......................................................... 4  
   2.2 System architectures ................................................... 4  
   2.3 Implementation ........................................................ 4  
   2.4 Analysis ................................................................. 4  
   2.5 Reliability ............................................................... 4  

3 **Public API comparison** .............................................. 6  
   3.1 Introduction ............................................................ 6  
      3.1.1 Background ......................................................... 6  
      3.1.2 Criteria .............................................................. 6  
      3.1.3 Selection of public APIs ......................................... 7  
   3.2 Comparison study ........................................................ 7  
      3.2.1 Authorization ....................................................... 9  
      3.2.2 CO₂ (Carbon Dioxide) .......................................... 9  
      3.2.3 Communication ..................................................... 9  
      3.2.4 Devices ............................................................. 10  
      3.2.5 History data ....................................................... 10  
      3.2.6 Humidity .......................................................... 10  
      3.2.7 Licensing agreements ........................................... 11  
      3.2.8 Limitations ....................................................... 11  
      3.2.9 Measurement interval ........................................... 11  
      3.2.10 Other functionality ............................................. 12  
      3.2.11 Security ......................................................... 14  
      3.2.12 Temperature ..................................................... 14  
      3.2.13 Temperature scales .......................................... 14  
      3.2.14 Time zone ........................................................ 14  
      3.2.15 Weather forecast ............................................... 15  

4 **System architecture** ................................................. 15  
   4.1 Introduction ............................................................ 15  
      4.1.1 Background ......................................................... 15  
      4.1.2 Purpose and Scope .............................................. 15  
      4.1.3 Stakeholder Representation .................................... 16  
      4.1.4 How each system architecture is organized ................. 16  
   4.2 SA 1: Service oriented architecture (SOA) pattern ........... 17  
      4.2.1 Overview ......................................................... 17  
      4.2.2 Elements ........................................................ 17
A.14 TemperatureMeasurement
A.15 Index.cshtml page for the HomeController
A.16 Login.cshtml page for the AuthenticateController
A.17 DisplayDevice.cshtml page for the DataController
A.18 DisplayMeasure.cshtml for the DataController
A.19 Netatmo.cshtml start page for the HomeController
Abbreviations and terminology used

API – Application Programming Interface is an interface with a set of routines, protocols, and other tools for building software applications. API defines functionalities that are independent of their implementations.

C# - is an object-oriented programming language.

CO - Carbon monoxide.

CO₂ - Carbon dioxide.

ESB - Enterprise Service Bus.

GMT - Greenwich Mean Time.

HTTP - HyperText Transfer Protocol is a structured text (messages) that is used for communicating between clients and servers.

HTTPS - HTTP Secure or HTTP over SSL.

JSON - JavaScript Object Notation is a data format that is easy for both humans and computers to read and understand.

MVC - Model View Controller - a software architectural pattern.

MQ - Message Queuing.

OAuth2.0 - Authorization process for multiplying platforms and devices.

PWS - Personal Weather Station.

REST - Representational State Transfer is a software architecture that consisting of different designs and best practices for creating scalable web services.

SAD - Software Architecture Document.

SOA - Service Oriented Architecture - a software architectural pattern.

SSL - Secure Sockets Layer is cryptographic security protocol designed for communication over a network.

UML - Unified Modelling Language

URL - Uniform resource locator is an reference to a resource on a network. Many different protocols like HTTP, FTP and JDBC use URL.

UTC - Universal Time Coordinated.

XML - Extensible Markup Language.
1 Introduction

This thesis project in computer science was written at Linnaeus University. The course was 15 credits, called 2DV00E and written at bachelor level. The work was made at NIBE AB who gave the project proposal and supervised the project.

1.1 Introduction / Background

Smart home is a popular concept in today’s technology-focused society. To connect all electronic appliances in a household to get the synergies is no longer a vision of the future, but something that is possible even for the ordinary consumers [1, 2]. Exactly how this is done varies between different manufacturers, many would love to see themselves as the brain of the smart home and use internal proprietary protocols to communicate. What they have in common is APIs (Application programming interfaces) that are made available on the Internet via HTTP [3, 4].

1.2 Problem definition

A future implementation of different public APIs in smart homes should be as general as possible, where the target is to integrate several different API vendors into an existing system. In the future other public APIs should be possible to integrate with small amount of effort.

For this to be possible, work should be based on a comparative study of available public APIs to get a stronger foundation to build from. Based on the lessons provided for this study a couple of system architectures can be developed where the focus is on developing a common platform for integration of public APIs. As a last step implementation of one of these system architectures can be done.

The expected results are the following:

- A comparison of different public APIs that can be used together with heating systems in smart homes.
- Several system architectures where the focus is on developing a common platform for integrating public APIs.
- An implementation that is based on one of the derived system architectures and integration of the selected API.

1.3 Purpose and research questions

Many companies try to be the center of the smart homes and connect their services to the Internet so they can be controlled and managed from distance [5]. Solving that problem can improve the customer benefit and create a link between products. This can make it easier for the customer to connect different services to each other if there are more common and modular platforms for the integration of public APIs [6].

The objective is to compare public APIs and create system architectures, that can be used with the heating system in smart homes, to be able to answer the research questions in Table 1.1.
RQ1 What differences are there between the selected APIs and what can they perform?

RQ2 How can system architectures, based on different design patterns, look like and what are the main advantages and disadvantages of each?

RQ3 How does one of the implemented system architectures cooperate with a heating system device in a smart home?

Table 1.1: Research questions for this project.

1.4 Scope / Limitation

To narrow down the field, the comparison will be of four different public APIs that can be used with heating system devices in smart homes. Choosing the different system architectures will be decided after the comparison is complete. The whole system with all different functionality requests will not be modelled in the system architecture due to time constraints. A general and common solution with temperature and humidity measurements will be created. The implementation of a chosen architecture will be decided and motivated when the system architectures are complete. Performance will not be a priority since latency, both from internet and the devices, will play a big part. No stress test or similarly can be tested due to the limitations of number of requests in the public APIs.

1.5 How this document is organized

Section 1 - Introduction presents the background, problem and describe things that affect the project.

Section 2 - Method gives an explanation of the various methods that will be used to solve the problem.

Section 3 - Public API comparison contains the result and analysis for comparison.

Section 4 - System architecture contains the result and analysis for system architectures.

Section 5 - Implementation contains the result and analysis for implementation.

Section 6 - Discussion regarding the results.

Section 7 - Conclusion from this project and future research.

1.6 Technologies

This section briefly describes different technologies and protocols that can be found in this document.

1.6.1 OAuth2.0

OAuth 2.0 are an open standard for authorization. The OAuth 2.0 framework was released in October 2012 and are not backwards compatible with OAuth 1. It provides a specific authorization flow for many types of applications. It can be used for many kinds of platforms like mobile phones, web applications or desktop applications. It works well with HTTP and allows creating access token for third party clients by using an authorization server. It is often used to login to third party web sites with more common accounts, like Google or Facebook accounts. It uses SSL for confidentiality and server authentication [7, 8].

2
1.6.2 JSON

JSON (JavaScript Object Notation) is a lightweight data format used for exchanging data objects. It is designed to both be easy for people to read and write. JSON is also easy for computers to generate and parse. JSON is language independent and can be used with many programming languages. The notations and conventions are often easy to understand for developers that have experience of programming in the C-languages, Java, Perl or Python. JSON has two structures and the first one consists of a collection of name and values (object in most languages). The second structure is an ordered list of values (array in most languages) [9].
2 Method

This section explains which methods that will be used for solving the research questions.

2.1 Comparing APIs

To investigate and compare various APIs, we will start by writing down a few areas that I believe is important to investigate. Investigate a couple of public APIs’ documentation, to be inspired by what they think is important, to include in the comparison. I will create a form and I will then consult a couple of people in the computer science field if they have comments on the points I have included for the comparison. When this is done, I will study various public APIs’ documentation and collecting information and enter it into the form previously created.

2.2 System architectures

One of the goals is to develop a general system architecture which should be easy to integrate into an existing system. This is difficult to achieve with only one system architecture so therefore three architecture proposals will be produced that may be suitable for different systems. OAuth 2.0 are used by many different public APIs so the different architectures will be based on it. Moreover the system architectures will differ since they will be based on different architecture patterns. Research will be needed to see which architectures pattern that can be suitable to build from. I will spend time on research on what technologies that can be used. I will create UML diagrams and write documentation for the different system architectures. I will also give short descriptions of patterns and technologies used. These descriptions will not be complete but are meant for the reader to gain some understanding of what the pattern or technology is about.

2.3 Implementation

One system architecture and one public API will be selected. I will give a motivation why those two got selected. They will then be implemented and tested against hardware. As a last step an analysis and conclusion of the project will be given and the project will come to a closure.

2.4 Analysis

The API comparison will be given to NIBE AB, and a couple of other persons in the computer science field. I will analyse the feedback and responses I get from them. This will be mentioned in the discussion part. I will analyse the architecture by implementing one of them. The implementation will be tested against hardware and some exploratory testing and code review will be done.

2.5 Reliability

The comparison will be based on the documentation that is provided by the company behind the public API. Since all of the selected APIs are well known and tested the reliability should be very high. What might go wrong is my interpretation about details in new technologies that I have not yet read about, which my thought is to do before API comparison is complete. The architectures will be based on well known patterns and will
be tailored to meet the requirements for this project. In the implementation I will use well
tested libraries in the programming language I choose. I will also test the implementation
in real devices and present the result in the report.
3 Public API comparison

This section contains an introduction, background and the API comparison.

3.1 Introduction

This is a comparison study of different public APIs that can be used together with heating system devices in smart homes.

3.1.1 Background

The comparison is based on four public APIs and below is a short description of them. The Weather Underground [10] API differs from the other APIs but I have decided to look at it as well in this comparison. The main difference is that the other APIs are more used inside the homes and have different devices like thermostats while Weather Underground is more a weather collection API where different reports can be extracted.

- **Ecobee** [11] provides HTTP-based interfaces for access to and control Ecobee thermostats, where the users can read, update and get information about their system. The manufacturer’s aims are to be backwards compatible with previous versions and that their API is self-documenting and easy to use and understand.

- **Nest** [12] is a real-time data API that through subscription offer access to data shared between Nest devices. It is mostly developed for the American market where the thermostats in different rooms control the heating system. One of their biggest selling point is that Nest can lower the energy costs. It can be configured in many ways, like lowering the temperature when no one is at home to save energy.

- **Netatmo** [13] has sold weather stations all over the world and collect weather data that can be accessed by public APIs. Users can control and get data by connecting their weather station and thermostats in their homes. With their special camera, with face recognition, you can for example get real-time updates in your application when your kids come home. Other events like when doors or windows open can easily be notified about in your smart phone.

- **Weather Underground** presents weather forecasts in various ways. Combined with a *Personal Weather Station* you can collect and view weather data from your own station. You can collect history from the complete timespan since the weather stations was initially started. Weather Underground is widely deployed and used all over the world.

3.1.2 Criteria

The list below shows the different things that are included in the comparison and a brief description of them. More information is provided in their respective sections.

- Authorization - Which protocols are used?
- CO₂ (Carbon Dioxide) - Can it be measured?
- Communication - Which protocols are used?
- Devices - Needed or optional?
• History data - For how long is the data stored?
• Humidity - Can it be measured?
• Licensing agreements - For using the API.
• Limitations - What limitations are there?
• Location and language settings - How is it decided?
• Measurement interval - How often is data measured?
• Other functionality - Some of the other functionality that is supported.
• Security - Is it secure?
• Temperature - How is temperature stored and which data can be requested?
• Temperature scales - Which are supported?
• Time zone - How is it stored?
• Weather forecast - Is it supported?

3.1.3 Selection of public APIs

The process of selecting the APIs includes searching the internet for different APIs and companies that offer this kind of service. NIBE has come with suggestions of companies that they have come in contact with. We have selected the first three APIs (Ecobee, Nest, Netatmo) because they seem to have a well developed API (at a glance) and their focus lies on thermostats in smart homes. The last public API (Weather Underground) is of a different kind since it focus is on the weather outside and using different weather stations. We thought it can be good to see some of the differences and similarities even though we cannot make an entirely fair comparison.

3.2 Comparison study

This section contains the comparison study and various features that have been compared between the selected APIs. Table 3.2 shows a summary of the API comparison.
<table>
<thead>
<tr>
<th></th>
<th>Ecobee</th>
<th>Nest</th>
<th>Netatmo</th>
<th>Weather Underground</th>
</tr>
</thead>
<tbody>
<tr>
<td>Authorization</td>
<td>OAuth 2.0</td>
<td>OAuth 2.0</td>
<td>OAuth 2.0</td>
<td>Subscription</td>
</tr>
<tr>
<td>Celsius</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>CO₂ (Carbon Dioxide)</td>
<td>No</td>
<td>No, CO</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Communication</td>
<td>JSON over HTTPS</td>
<td>JSON over REST or Firebase</td>
<td>JSON over REST</td>
<td>JSON or XML over HTTP</td>
</tr>
<tr>
<td>Devices needed</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Fahrenheit</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>History data</td>
<td>15 months</td>
<td>Not defined</td>
<td>No limit</td>
<td>No limit</td>
</tr>
<tr>
<td>Humidity</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Limitations for free</td>
<td>85 000 queries per month</td>
<td>1000 users</td>
<td>20 request for each user per hour</td>
<td>500 calls per day</td>
</tr>
<tr>
<td>Measurement interval</td>
<td>5 minute</td>
<td>Not defined</td>
<td>5 minute</td>
<td>6 hours</td>
</tr>
<tr>
<td>No one home temp.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Target temp.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Temperature decimals</td>
<td>One</td>
<td>Half(C)/whole(F)</td>
<td>One</td>
<td>One</td>
</tr>
<tr>
<td>Time zone</td>
<td>UTC</td>
<td>Not defined</td>
<td>UTC</td>
<td>UTC, mostly</td>
</tr>
<tr>
<td>Weather forecast</td>
<td>7 days</td>
<td>No</td>
<td>7 days</td>
<td>10 days</td>
</tr>
</tbody>
</table>

Table 3.2: Summary table of the API comparison.
3.2.1 Authorization

This section describes which technologies and protocols that are used for the authorization between a client application and the API.

- **Ecobee** authorization is based on the OAuth 2.0 protocol and their implementation uses three different methods: Authorization Code, PIN Authorization and Utility Authorization.

- **Nest** authorization is based on the OAuth 2.0 protocol.

- **Netatmo** uses OAuth 2.0 protocol for authorization between a client application and a Netatmo server. The implementation they use for OAuth 2.0 has two different methods. The first method (Client Credentials) allows you to retrieve an access token in one step. The second method (Authorization code) takes four steps to get the access token.

- **Weather Underground** lets the client application subscribe at their website and then get a generated key that is used in the request calls.

3.2.2 CO₂ (Carbon Dioxide)

This section describes if the API with its devices can measure CO₂.

- **Ecobee** can not measure CO₂.

- **Nest** can measure CO (carbon monoxide) with the *Nest Protect Smoke + CO Alarm*.

- **Netatmo** can measure CO₂ with the *Netatmo Weather station* and the indoor modules.

- **Weather Underground** can not measure CO₂.

3.2.3 Communication

This chapter describes how the communication is handled when requesting information from the various APIs.

- **Ecobee** uses JSON notation for requests/response objects sent to and from the Ecobee API. All data is sent over HTTPS. The URL needs to be encoded, see the HTML URL Encoding Reference [14], otherwise the server may not recognize the request and will send back an error message.

- **Nest** communication is handled with either a REST call or Firebase client libraries [15] to subscribe to different data values from the Nest API Server. When an application’s subscribed data value is changed on the Nest Server, it will with Firebase be updated in real-time.

- **Netatmo** uses REST to send request and responses over HTTPS. Responses are JSON encoded.

- **Weather Underground** make all requests are made over HTTP and the data is returned in either JSON or XML format.
<table>
<thead>
<tr>
<th>API</th>
<th>Required</th>
<th>Name</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ecobee</td>
<td>Yes</td>
<td>Smarter Bundle (Thermostat + 2 Sensors)</td>
<td>$313</td>
</tr>
<tr>
<td>Ecobee</td>
<td>No</td>
<td>Remote Sensors 2 pack</td>
<td>$79</td>
</tr>
<tr>
<td>Ecobee</td>
<td>No</td>
<td>Ecobee Thermostat</td>
<td>$249</td>
</tr>
<tr>
<td>Ecobee</td>
<td>No</td>
<td>Smart Si</td>
<td>$179</td>
</tr>
<tr>
<td>Nest</td>
<td>Yes</td>
<td>Nest Learning Thermostat</td>
<td>$249</td>
</tr>
<tr>
<td>Nest</td>
<td>No</td>
<td>Nest Protect</td>
<td>$99</td>
</tr>
<tr>
<td>Nest</td>
<td>No</td>
<td>Dropcam Pro</td>
<td>$199</td>
</tr>
<tr>
<td>Netatmo</td>
<td>Yes</td>
<td>Netatmo Weather Station</td>
<td>$169</td>
</tr>
<tr>
<td>Netatmo</td>
<td>Yes</td>
<td>Netatmo Thermostat (EN)</td>
<td>$179</td>
</tr>
<tr>
<td>Netatmo</td>
<td>No</td>
<td>Indoor Module</td>
<td>$69</td>
</tr>
<tr>
<td>Netatmo</td>
<td>No</td>
<td>Rain Gauge</td>
<td>$69</td>
</tr>
<tr>
<td>Weather Underground</td>
<td>Yes</td>
<td>Personal Weather Station</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.3: Devices available.

3.2.4 Devices
Table 3.3 list all devices that are either required or optional when using the API. The prices are from the companies own websites.

3.2.5 History data
This section describes if the API store data and if the end user can view the history data.

- **Ecobee** can export up to 31 days (at one time) of data over the last 15 months.
- **Nest** - Not available.
- **Netatmo** have 2 weeks in the device and up to no limit in the cloud.
- **Weather Underground** depends on the weather station where some stations have decades of data.

3.2.6 Humidity
This section describes if the API with their devices can measure humidity.

- **Ecobee** can measure humidity from the *Netatmo Thermostat*. There are options for reading outdoor humidity, the average humidity for a zone and high or low humidity in a zone.
- **Nest** can measure humidity from a *Nest Thermostat*.
- **Netatmo** can measure humidity from a *Weather Station device*, notice not from a *Thermostat*.
- **Weather Underground** can measure humidity.
<table>
<thead>
<tr>
<th>Price per year</th>
<th>Queries per month</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0</td>
<td>85 000</td>
</tr>
<tr>
<td>$500</td>
<td>255 000</td>
</tr>
<tr>
<td>$1 000</td>
<td>425 000</td>
</tr>
<tr>
<td>$1 500</td>
<td>595 000</td>
</tr>
</tbody>
</table>

Table 3.4: Ecobee options and limitations.

<table>
<thead>
<tr>
<th>Limit</th>
<th>Requests every 10 seconds</th>
<th>requests every hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application limit</td>
<td>(2 * number of users)</td>
<td>(20 * number of users)</td>
</tr>
<tr>
<td>User limit</td>
<td>50</td>
<td>500</td>
</tr>
</tbody>
</table>

Table 3.5: Netatmo options and limitations.

### 3.2.7 Licensing agreements

This section contains the licensing agreements that need to be approved when creating an account at their website.

- **Ecobee** Licensing Agreement [16].
- **Nest** Nest Developer Terms of Service [17] and Nest UI & Marketing Guidelines [18].
- **Netatmo** Terms of use [19].
- **Weather Underground** Terms and conditions of use [20].

### 3.2.8 Limitations

Table 3.4 shows a selection of the Ecobee different limitations. Nest can have a maximum of 1000 users then a request for more users is needed. Table 3.5 shows that Netatmo have two kinds of limitations for the private API. Weather Underground have some standard options and limitations to select from, see Table 3.6, but it can also be customized.

### 3.2.9 Measurement interval

This section describes how often data is measured and stored.

- **Ecobee** Data is measured with an interval of 5 minutes on every day.

<table>
<thead>
<tr>
<th>Monthly price</th>
<th>Calls per day</th>
<th>Calls per minute</th>
<th>History</th>
</tr>
</thead>
<tbody>
<tr>
<td>Developer</td>
<td>$0</td>
<td>500</td>
<td>10</td>
</tr>
<tr>
<td>Drizzle</td>
<td>$20</td>
<td>5 000</td>
<td>100</td>
</tr>
<tr>
<td>Shower</td>
<td>$200</td>
<td>100 000</td>
<td>1000</td>
</tr>
</tbody>
</table>

Table 3.6: Weather Underground options and limitations.
• **Nest** Not specified.

• **Netatmo** The Weather station measures every 5 minutes. However there are 7 options for retrieving measured data:
  
  – **max**: every value stored will be returned
  – **30min**: 1 value every 30 minutes
  – **1hour**: 1 value every hour
  – **3hours**: 1 value every 3 hours
  – **1day**: 1 value per day
  – **1week**: 1 value per week
  – **1month**: 1 value per month

• **Weather Underground** The weather forecast is updated around every 6 hour. When the update is done varies between for different locations.

### 3.2.10 Other functionality

This section describes some of the other methods and parameters that are not mentioned elsewhere in this report. This is not a complete list, see respective documentation for a full list.

• **Ecobee** has 45 different object’s, for example:
  
  – *Event*
  – *Device*
  – *Status*
  – *Thermostat*
  – *Location*
  – *SecuritySettings*

All these objects have a number of properties that can be changed. The *Location* object has the following 10 properties:

  – timeZoneOffsetMinutes
  – timeZone
  – isDaylightSaving
  – streetAddress
  – city
  – provinceState
  – country
  – postalCode
  – phoneNumber
  – mapCoordinates

• **Nest** has plenty of functions and here is a small selection of them.
- `software_version` The software version.
- `structure_id` Unique identifier.
- `name` Name of the device.
- `last_connection` Time-stamp, in ISO 8601 format, of the last successful interaction.
- `is_online` If the device has connection with the Nest Service.
- `can_cool` If the system can cool.
- `can_heat` If the system can heat.
- `is_using_emergency_heat` Emergency heat status.
- `has_fan` System ability to control the fan independently from heating or cooling.

**Netatmo** For Weather Station

- `CO2` measured value in ppm.
- `Pressure` measured value in mbar.
- `Noise` measured value in db.
- `Rain` measured value in mm.

For Thermostat

- `BoilerOn` Duration, since the previous data point, during which the boiler has been on (in seconds).
- `BoilerOff` Duration, since the previous data point, during which the boiler has been off (in seconds).
- `sum_boiler_on` Total duration during which the boiler has been on, at a specific scale (in seconds).
- `sum_boiler_off` Total duration during which the boiler has been off, at a specific scale (in seconds).

For both

- `date_begin` Starting timestamp (UTC).
- `date_end` Ending timestamp (UTC).
- `limit` Limits the number of measurements returned.
- `optimize` Can be used for saving bandwidth usage.

**Weather Underground** have a lot of weather data like the following:

- `wspdm` WindSpeed kph.
- `wspdi` Windspeed in mph.
- `wdird` Wind direction in degrees.
- `vism` Visibility in Km.
- `pressurem` Pressure in mBar.
- `windchillm` Wind chill in Celsius.
3.2.11 Security

For communication between the public API and the application the security lies in the different protocols used for communicating and authorization, see their respective sections.

The communication between the devices and the public API is hidden for security reasons. The API takes care of this since the communication is between their devices and servers.

3.2.12 Temperature

This section briefly describes how the temperature is presented, if there are options to set and get a targeted temperature and if there are other useful options.

- **Ecobee** measure temperature in Fahrenheit and raw data values are multiplied by 10. So 72.1 degree Fahrenheit is presented as 721. It can set both a targeted cool and targeted heat temperature. There is support for setting a weakly schedule or enter when going away on a vacation.

- **Nest** measure temperature in Celsius (half degrees) or Fahrenheit (whole degrees) at the device. It can set target, target high and target low temperatures, and can also set high and low temperatures when no one is home.

- **Netatmo** measure temperature in Celsius with one decimal. It can get and set the target temperature. The thermostat has support for setting an away temperature when no one is at home.

- **Weather Underground** measure temperature in Celsius and in Fahrenheit with one decimal.

3.2.13 Temperature scales

Table 3.7 shows which temperature scales that are supported by the APIs.

<table>
<thead>
<tr>
<th></th>
<th>Ecobee</th>
<th>Nest</th>
<th>Netatmo</th>
<th>Weather Underground</th>
</tr>
</thead>
<tbody>
<tr>
<td>Celsius</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Fahrenheit</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 3.7: Table of which temperature scales the APIs support.

3.2.14 Time zone

This section describes how the time zone where the end user lives is stored. The UTC format comes from French *universel coordonné* and is translated to Coordinated Universal Time.

- **Ecobee** end user needs to enter their country and postal code. Timestamps are in UTC.

- **Nest** end user needs to enter their country and postal code.
3.2.15 Weather forecast

This section describes if the public API supports weather forecasts.

- **Ecobee** have support for viewing the weather forecast for different areas, up to 7 days or the next 24 hours in more detail.
- **Nest** - Not supported.
- **Netatmo** have up to 7 days weather forecast.
- **Weather Underground** have up to 10 days weather forecast.

4 System architecture

This section introduces the background, purpose and scope, and how system architectures are organized as well as which type of stakeholders that may be interested in these system architectures. Finally the system architectures are presented in their respective section.

4.1 Introduction

There are many public APIs available that can connect devices with the heating system in smart homes. Below are a few different system architectures that describe how an existing software system can integrate and use public APIs. The system architectures are based on well known patterns.

4.1.1 Background

Many companies try to be the central point in smart homes. Since many companies release a public API for others to use I will try to create a system architecture that easily can add new and other public APIs easily and without too much effort. Many authorization processes use OAuth2.0 therefore I will explore the options to use it as a general authorization process for the all the system architectures. OAuth2.0 is a restriction that the public API needs to support.

4.1.2 Purpose and Scope

To be able to integrate new public APIs in an existing system a supporting architecture is necessary to have. Therefore I have developed some different architectures that could be suitable for different existing systems. I have decided to limit the public APIs to the ones that uses OAuth2.0 to be able to create some general system architectures. The architectures will show how to authenticate and extract at least temperature and humidity data.
4.1.3 Stakeholder Representation

This architecture is suitable for the stakeholders listed below. Since each architecture description is a couple of pages each I have chosen not to extract the information to different viewpoints based on the stakeholder’s needs.

- Management - The managers which monitor the system.
- Developers - Constructs the system from specifications.
- Testers - Test the system to ensure it is suitable to use.
- Users - Uses the system.

4.1.4 How each system architecture is organized

The list below shows the different things that are included in each system architecture, and a brief description of them.

- System architecture patterns, with a brief description, that have been used with their advantages and disadvantages.
- UML diagrams that explain the architecture in a graphical representation.
- Documentation that explains the solution and relate to the UML diagrams.
4.2 SA 1 - Service oriented architecture (SOA) pattern

This system architecture builds on the service oriented architecture pattern.

4.2.1 Overview

SOA is a large framework that can be customized in many ways. Micro services are one type of pattern that is a part of the SOA framework. SOA in general is a set of cooperating components that can both consume and/or provide services. Each service has its own area of responsibility. For connecting services to each other both SOAP, REST and asynchronous messaging can be used. Since the communication is handled with these generic interfaces, services can be implemented in different languages and are very loosely coupled [21, 22].

4.2.2 Elements

Below is a brief description of some of the elements used in SOA [21]:

- Service providers - which provide at least one service through public interfaces. Can also be a service consumers.
- Service consumers - which call services.
- ESB - which acts as an intermediary that can route and transform messages between service consumers and providers.
4.2.3 UML diagrams

Figure 4.1: Class diagram for the SA 1 architecture.
4.2.4 Documentation

I have decided not to include an ESB in this architecture because I want to clarify the important parts of how the various services work. If this architecture shall be integrated to a system that already has an ESB it should be easy to reconstruct the architecture to fit with the ESB. I would recommend using the ESB if it is already constructed to get more functionality of the SOA pattern and let the ESB manage the messaging between services [21, 23].

Here is a basic set of services that are needed for this architecture:

- Authentication service (AS) - Handles the authentication of users through OAuth2.0.
- Select data service (SDS) - Selects one API provider and map a request to that provider.
- Communicator service (CS) - Creates and handles the HTTP requests and responses.
- Interpret data service (IDS) - Interpret the HTTP response and format it to be displayed in a general way.

The class diagram in Figure 4.1 shows services and a basic set of methods to authenticate a user and be able to get temperature and humidity data from a public API.
The SelectDataService holds the authentication token that will be used to make requests to the public APIs. It also has a method named getMethods that returns a list of methods that are supported by the system. One of those methods are Temperature that can be extracted from all public APIs. Since different APIs can call this in different ways either the public API implementation needs to map these to the correct method in the API or there can be another service that handles this. In my example I have chosen to put this in the public API implementation.

The AuthenticationService handles the authentication. Different authentication methods that are supported by OAuth2.0 can be implemented. When the authenticationToken has expired a new token can be requested by providing the refresh token. The refresh token should be stored with the user data, for example in a database.

The InterpretDataService (IDS) contains all the logic of processing data. Public APIs return the temperature in Celsius and others in Fahrenheit and it is the IDS responsibility to convert these temperatures to the temperature scale set by the user.

Below is a flow of how a user can request to display the current temperature from a public API. It can also be seen as a sequence diagram in Figure 4.2.

1. User selects to display temperature data form the Netatmo API.
2. SDS verifies the authentication (or authenticate the user) with AS.
3. SDS call getData from CS with Netatmo and Temperature.
4. CS requests the URL from the public API (Netatmo in this case).
5. CS creates and makes a request to the public API.
6. CS sends the response to the IDS for interpreting the data to the correct format.
7. IDS returns the formatted temperature data to the CS.
8. CS returns the formatted temperature data to the SDS.
9. SDS returns the formatted temperature data to the user.

Note that the user can request this from many devices. It can be from a web interface, console application or something else where the temperature is displayed.

4.2.5 Pros

Each service should be designed to be independent so it does not depend on other services. This leads to loose coupling between services and it is easy to maintain or replace implementations of various services. The components should have interfaces that describe the service so it will even be possible to have different implementation languages for the components. SOA is a large framework and can be tailored to match specific needs. There are many different connectors and a few of them are SOAP and REST that both uses HTTP. An ESB can be good for some systems, but it is not always needed [21, 23].

4.2.6 Cons

The SOA pattern can be very complex to design and implement because of its size and it can be difficult to guarantee a certain performance between services. Services can become performance bottlenecks. It is not always possible to control the evolution of independent services [21].
4.3 SA 2 - Model-View-Controller (MVC) Pattern

This architecture pattern is based on the MVC Pattern.

4.3.1 Overview

The MVC pattern separates the user interface from the application functionality. By creating three different components, that handles different areas, it makes the application easy to modify and the components loosely coupled. The communication between the components are handled by events or callbacks notifications [21]. For a graphical representation see Figure 4.3.

4.3.2 Elements

There are three elements (model, view, controller) in the MVC pattern [21, 24]:

1. **Model** which handles the application data or state and provides an interface to the application logic. It responds to changes from the controller. When a change is made it notifies the views.

2. **View** represent a presentation layer, based on the data, to the user. The view interacts with the user and receives user inputs. The input is then sent to the controller that handles the input. It renders the changes from the models and requests updates from the model.

3. **Controller** translate the user interaction into changes on the view and/or model. The controller mediates changes between the model and the view and defines the application behaviour.
4.3.3 UML diagrams

Figure 4.3: Overview diagram for the SA 2 architecture.
Figure 4.4: General class diagram for the SA 2 architecture.

Figure 4.5: Class diagram for handling temperatures in the SA 2 architecture.
Figure 4.6: The Model’s class diagram for the SA 2 architecture.
4.3.4 Documentation

Figure 4.4 shows a general MVC design class diagram. There are three interfaces (IModel, IView, IController) and a class implementing each interface. Below is a description of each element in Figure 4.4:

- **IModel** - Interface for the models. It has methods for notifying views, and attach and detach ModelObservers.
- **IView** - Interface for the views. It has references to a controller class and an update method for updating the view.
- **IController** - Interfaces for the controllers. It has references to a view and model.
- **aModel** - A model that inherits the IModel.
- **aView** - A view that inherits the IView.
- **aController** - A controller that inherits the IController.

This is a general solution of how the interfaces and the relations among the classes can look like. Figure 4.5 shows how an architecture could look like for the temperatures. This is not how the final architecture looks like but is intended to show how the MVC pattern is organized using a small example.

Figure 4.6 shows a UML diagram with model classes. The OAuth2.0 authentication generates tokens and to store that information there is an AuthenticationToken class. The AuthenticationToken class stores the token, refresh token and when the token expires. This AuthenticationToken class is then used for each request to the public API. When the expire time has ended the refresh token is used to get a new token and a new expire time. This is handled by the AuthenticationController, seen in Figure 4.7, handle.

The User class is abstract as the idea of MVC is that every public API should have a class that extends the User class. The User class has information like a user ID, a list of IDs of various devices, as well as email and password.

The Device class represents a device. Since all devices from different public APIs are different there may be classes extending the Device class later on. For now it has a device ID, a type and a list of owners. Other information like where the device is located, language and time settings could be added.

The PublicAPI interface is meant to have information like the different URIs and what methods that can be used for that API, and information mapping each method to the correct URI.
The SensorMeasurementFactory creates an object that extends the abstract class SensorMeasurement. Depending of which measurement types that the architecture will support it is easy to add new measurement classes.

This makes it easy to add different new public APIs to the architecture. All that is needed is to be done is to create a new class that implements the PublicAPI and the User class. Depending on the User information the application needs, it is possible to create only one general User class for all different public APIs.

Figure 4.7 displays the controller classes. The AuthenticationController has different methods for authentication by the OAuth2.0 standard. Note that all authentication methods are not supported, but can be added if needed. The Resource Owner Password Credentials Grant, Authorization Code Grant and Refreshing an Access Token, from the The OAuth 2.0 Authorization Framework, is currently designed in the architecture [25].

I have decided to not show all the model and view classes here but they are instead presented in the implementation code in the appendix.

4.3.5 Pros

The separation of the different concerns into three components makes it loosely coupled. The components are easier to test and develop when they are separated. Components can easily be replaced with other implementations. To make the application support different platforms can be handled by creating different views which often can have the same logic. It is also easy to support different types of view with the same data at the same time. Developers can specialise and focus on different components without affecting others. Views are easily switched out without the user loses any data since that is taken care of by the models [21, 26].

4.3.6 Cons

MVC can be too complex for simple applications and is not always appropriate for every situation. It can be costly to create a design and implement three different types of components for simple user interfaces [21].
4.4 SA 3 - Multi-tier Pattern

This architecture is based on the multi-tier pattern.

4.4.1 Overview

Tiers can either be a group components with the similar functionality, or it can be defined by looking at the system it will run on. It can then have a client tier in an enterprise system or be grouped by having the same runtime purpose [21].

4.4.2 Elements

There is only one kind of element that is named: tier. It can be seen as a logical grouping of different software components [21].
4.4.3 UML diagrams

Figure 4.8: A multi-tier view for the SA 3 architecture.

4.4.4 Documentation

In Figure 4.8 I have created four tiers. The first tier represent the client tier where the client can use a web browser, console application or something else for making requests to the web tier. We can also create for example a mobile phone tier beside the web tier and then clients can access the system from mobile devices. The web tier can be seen as a front-end that makes requests to the back-end tier. The back-end tier is where all the logic is found. It processes the requests from the clients and have all logic about the different public APIs that are supported. It controls the authentication of clients and interpret the data that comes back from the public API tier. The Public API tier have the logic for the different supported APIs. It may have URLs to the different APIs, which methods that
are supported and other stuff related to the APIs.

4.4.5 Pros

Independent tiers make it possible to change and update tiers without affecting the other tiers. It has good scalability where the different tiers can be scaled, for example the database tier can scale up with more databases without affecting other tiers. Security controls can be added for each tier and be tailored to meet the demands for that tier [27].

4.4.6 Cons

Creating a multi-tier system often have a substantial up-front cost and high complexity. This makes the pattern not suitable for smaller projects. Performance can be slow if the hardware and bandwidth is not good enough since more computers, networks and servers are involved [21, 27].
5 Implementation

This whole section is about the implementation of one public API and the chosen system architectures.

5.1 Introduction

As a final step I have chosen one public API and one system architecture to implement. The motivation can be found below. I present the important code fragments below and the whole implementation can be found in Appendix 1. The focus is on making different requests to the API. Very little time has been spent on how to display the result. Most of the time the raw data is displayed on a web page.

5.2 Motivation

To choose a public API was difficult. All the public APIs I have looked at have both pros and cons and could be implemented in this project. All of them were more developed and larger than I could ever expect before I started working on this project.

I have chosen Netatmo as public API since:

- Documentation - Their documentation is well written with many examples on how to use the API, as well as how to use REST calls and information about OAuth2.0.

- Devices - I managed to get hold of a Thermostat and a Weather Station that I could test my application on. This was one important aspect since it is always good to see that the implementation works on the intended hardware.

- World wide - I got the impression that Netatmo was used world wide. Weather Underground could maybe have more users since it does not require any hardware.

I decided to implement a web application, and by using the MVC pattern it is easy to change the views to support other platforms or devices. With this solution others can use this code for starting up their own integration to their systems and only change the views if they do not use a web interface.

The MVC pattern is a good approach to split up the code into different sections. My focus will not be on the view section and this is something that others may improve to meet their demands and requirements.

The SOA pattern could be a better fit if there were other services that the system should interact with. In my case I will build a standalone application that can be imported to an external system. I could not motivate the time needed to build an ESB with the only service these application uses. Many of the classes I will build can be transferred to services without too much problems.

5.3 Result

I implemented the client credentials option where the user enter an email address and a password. The application then provides the client secret and client id together with the user credentials to access the public API. To make this method general to work with different APIs there needs to be a way for setting specific parameters to the request. See Code 5.1 for example where the client id is loaded for that specific API. Code 5.2 displays a method for making a request to an API and asking for all devices that are connected to.
if (APIName.Equals("Netatmo"))
{
    api = new APINetatmo();
    client_id = WebConfigurationManager.AppSettings
        .Get("client_id_netatmo");
    client_secret = WebConfigurationManager.AppSettings
        .Get("client_secret_netatmo");
    parameters += "&scope=read_station read_thermostat
write_thermostat";
}

Code 5.1: Fragment from AuthenticateUser method.

the user’s token. Depending on how many methods that needs to be supported different kinds of methods like DisplayDevices needs to be created. A future research could create a general method that takes different parameters and then can create a request to the API that the user asks for. A simple web page that shows the login options can be seen in Figure 5.9, where the user has the option of logging in with email and password, also called Resource Owner Password Credentials Grant in the OAuth rfc6749 documentation. The first option is to select an API to login to. Then the user needs to enter their email and password for their account on the selected API. When the login button is pressed a web request is sent to the API server and if the user information is correct an authentication token is returned to the application that is used for making other requests to the API.

If the user selects to login at an API page instead the user goes through the Authorization Code Grant process, described in the OAuth rfc6749 documentation, instead. This option redirects the user to the APIs page and a login page is displayed, see Figure 5.10. The user needs to login and then accept that this application gets access to the user’s devices. When the process is complete the application get an authentication token for that user. Figure 5.11 shows a list of devices that the user has. In this case there are only one device connected. Current measurement data like temperature and humidity is also displayed. To get a more detailed measure of example temperature another method can be used and the result are displayed as shown in Figure 5.12.
public async Task<ActionResult> DisplayDevices()
{
    var token = await getToken();

    // Get devices
    var method = "devicelist";
    // Add parameters
    var parameters = "?access_token=" + token.Token;
    var devs = await GetData(netatmo.getDataURL() + method + parameters);

    var jTokenBody = devs["body"];  
    var jTokenDev = jTokenBody["devices"];  
    var list = new List<NetatmoDevice>();  
    foreach (var item in jTokenDev)
    {
        var dev = new NetatmoDevice();
        dev.DeviceId = (string)item["_id"];  
        dev.Measurements = JsonConvert.DeserializeObject<NetatmoMeasurements>(
            item["dashboard_data"].ToString());  
        list.Add(dev);
    }
    return View(list);
}

Code 5.2: Code from the DisplayDevices method.
Figure 5.9: Login page with OAuth Resource Owner Password Credentials Grant and Authorization Code Grant.

Figure 5.10: Login page at the Netatmo using the Authorization Code Grant authentication process.
Figure 5.11: A list of devices with a short list of currently measure data.

Figure 5.12: Page that display a measure value from one device, in this case temperature data for a Netatmo thermostat.
6 Discussion

In this section I will present my own discussion regarding this project. I begin by discussing the problem description, planning and the method approach I have worked after. After that follows a section about the comparison, system architecture and finally implementation.

6.1 Initial discussion

I felt that the problem I was suggested by NIBE is interesting, large and a new area for me. I felt like it was a good challenge. Smart homes have always interested me, from early age I have tried to improve things at home and use technologies wherever I can. When NIBE suggested this problem, I realized how useful this can be in smart home. At the beginning I did not know that many of the API providers had their own equipment and different devices. I was surprised that this area had progressed so far, for example that there where several different API providers and that their devices were so advanced and could do a lot of things. I only used a small part of the API and I suggest that the interested reader look at the different API documentation. I will for sure get some devices to use in my home in the near future.

Since the problem area was large I had to set some limitations at the beginning. Now when the work is complete I realised that the limitations were not enough. I should have been more clear of what I was expected to implement, for example that I needed to limit the implementation to just extract temperature and humidity data. I did not know how much time the different things should take therefore I planned to be ready two weeks before the project should be completed. I feel like I did not have time to develop the third system architecture as I wanted. The research time was hard to estimate. I also wanted to start implementing to have some results to present from the implementation.

My method was to split up the project into three stages and I feel that was a good choice. I think if I had worked a little more iteratively it would be better. I had planned time for reading up on new technologies like OAuth2.0. It could have been good to plan to implement one or two of the OAuth2.0 authorization methods before starting on the system architectures. I ended up implementing one authorization method before the system architectures were finished since I needed the knowledge of how OAuth2.0 really worked. A more iterative process and plan could have been better.

6.2 Comparison discussion

My approach to create a list of things I thought was important first worked out pretty good. Many of the things from the first list are in the comparison now. Asking people at NIBE gave me a couple of other interesting aspects to look into, for example adding an explanation of different technologies used. I also noticed what the API providers thought was important by looking at their advertisement and documentation. All these things together created the comparison that is now presented in this report.

I noticed that the different API providers use different terms for the same things, for example functions and methods. These two words mean the same thing but is used in different areas. I also noticed that the thermostats were more intended for the American market where the thermostats control the heating system. On the European market the heating system often control the temperature and the thermostats are more dumb and just report the current temperature to the heating system. This was a new area for me that gave me some more insight when I found out how this worked.
When I had looked deeply in the results I found that all APIs could do more or less the same things. They differ in how some the API methods are constructed, but all work almost the same. The application make requests and gets a lot of data. The requests can be customised by asking for specific parameters or set interval of measurement that shall be returned.

There is a lot of other data that I have not looked so much into, for example the settings for different devices, user settings and heating settings. There are many options of setting and reducing the energy consumption. This is very important in the smart homes to be able to reduce energy consumption and is another motivation of why to get smart thermostats or a similar system for the heating in a home. A so simple thing like lowering the temperature when no one is home is a good way to save some energy. This was supported by most of the API providers.

Weather Underground was a different type of API. We decided to include it to get some inspiration and see the differences by not needing to have devices. This showed that a lot of information can be found by using a weather station located near the home. This shows that everyone does not need their own equipment to get some features. To lower the energy usage, you need your own devices to be able to control the heating system for maximum results.

6.3 System architecture discussion

Creating the system architectures was the hardest part of this project, since it is difficult to obtain enough information on the technology required to be able to create a good architecture that takes all important factors into consideration.

The first two (SOA and MVC) patterns were good choices to start with. The third was hard to select, partly because I thought I already had found a good architecture in one of the previously two. I realized once again that one must have a lot of knowledge in order to develop an architecture. It ended up that I worked iteratively with the first two architectures and then I made the third one afterwards.

I realized that I should have created more limitations of what should be included in the architectures. Sure that the methods for retrieving data is general and can use for different measurements but I would have wanted to specify in more detail how it would be done via an architecture and not allow the implementation to be so free of restrictions.

The third architecture I selected was to do an architecture based on Multi-tier pattern. That solution could be a good choice for some systems. I think the other two patterns where more suitable for my situation. Hence, it did not come far in the implementation election.

It was hard to select one to continue and implement. I feel that both the SOA and MVC architectures would be good choices. Now afterwards I feel that the decision was right to proceed with the MVC pattern, mostly because it is easy for others to change the views to match their projects and then they can continue to develop the controllers and models.

One aspect that I have not mentioned before is that it could be good to use a database for storing some information about the user and for example the refresh token. This is something that I recommend to do and can be found in the future work section. If I had more time I would add database support both to the architectures and the implementation.

It would be interesting to implement the SOA pattern as well. It would then be easier to compare them and evaluate if I made the right choice by choosing MVC pattern. It depends a lot on what kind of system this should be integrated into to get full power of
the SOA and MVC pattern. Implementing the MVC pattern in a SOA system could lose many of the benefits of MVC.

6.4 Implementation discussion

I decided that the programming language will be C# since the existing system at NIBE is mostly written in C# and it is a modern and interesting language. I had some previous experience with C# for some years ago and I thought it would be a good experience to refresh those skills. This was something that I planned to do outside the project time budget. I am happy with that I selected C# since I learned a lot and refreshed my programming. This was maybe too time consuming since a lot had changed since I used it and even if I planned to learn it outside the project some coding took longer time than I would have wanted. This summed up that I did not have time to implement all features that I wanted. For example the views are very basic and I miss features like selecting different measurements readings, data history, user settings and most importantly that I did not have time to implement all of the OAuth2.0 authentication options that I first wanted. Same as with the testing, I would like to have more time on creating unit tests and make improvements to the code.

The code I did implement shows how to use OAuth client credentials to login to Netatmo API. Temperature and humidity is fetched from a thermostat located in Sweden and pictures of how it looks like can be found in section 5. Note that I have erased the client id and client secret from the implementation and pictures since it is not intended that they should be open to the public.

Selection of public API was both hard and not hard. Since all APIs were very similar and had all the usual features I had to make my choice based on other things. I could get access to a Netatmo device. This was a pretty important aspect since I could then test my implementation worked on real hardware. I did not want to implement Weather Underground since they did not have any devices that could be used in the smart homes. All these things led to that I selected Netatmo as API to implement and test against.
7 Conclusion

This section contains if I have answered my research questions and suggestions for further research.

7.1 Conclusions

The conclusions for RQ1 are the following: Most of the public APIs are very similar, and they can perform almost the same things. There are some differences like temperature scales, limitations for the free version and if they can measure CO and CO\textsubscript{2}. Weather Underground offers a lot of options even if the user does not have an own hardware device.

The conclusions for RQ2 are the following: The system architectures vary a lot depending of which pattern they are based on. My conclusion is that it is important to look at the system it will be integrated to and more in-depth information of that system will impact the decision on which architecture to choose. If it will be a new system I would recommend the SA 2 - Model-View-Controller option with the same motivation I have given in section 5.2.

The conclusions for RQ3 are the following: The implementation works well with a thermostat from Netatmo. In the general implementation is it easy to add other APIs it manages to login to an API and it can extract information form the device.

7.2 Further research

I have found some areas that I think is worth investigating more:

- Other architectures could be developed to see if they are a better fit?
- Create a way to map different API calls into one call, for example is there a way to create one method to find out the humidity in all different public APIs? If not, how could that then be implemented without too much trouble?
- Support for databases to store user information (if wanted) and refresh tokens.
- Add support for other sensor measurements, more device types and other authentication methods.
- A general method that create a request to the correct API depending on different parameters.
References


Appendix 1 - Implementation

A.1 DataController

```csharp
public class DataController : Controller {
    private APINetatmo netatmo = new APINetatmo();

    /// <summary>
    /// Index page.
    /// </summary>
    /// <returns></returns>
    public ActionResult Index() {
        return View();
    }

    /// <summary>
    /// Returns the authentication token.
    /// Note that this method could be changed to fetch the token from a database or a file.
    /// </summary>
    public async Task<AuthenticationToken> getToken() {
        // Read from the chache.
        if (token == null) {
            token = await new AuthenticateController().RefreshToken();
        }
        return token;
    }

    /// <summary>
    /// Display all devices.
    /// </summary>
    public async Task<ActionResult> DisplayDevices() {
        var token = await getToken();
        // Get devices
        var method = "devicelist";
        // Add parameters
        var parameters = "?access_token=" + token.Token;
        var devs = await GetData(netatmo.getDataURL() + method + parameters);
    }
}
```
```csharp
var jTokenBody = devs["body"];  
var jTokenDev = jTokenBody["devices"];  
var list = new List<NetatmoDevice>();  
foreach (var item in jTokenDev)
{
    var dev = new NetatmoDevice();
    dev.DeviceId = (string)item["_id"];  
    dev.Measurements = JsonConvert.DeserializeObject<NetatmoMeasurements>(
        item["dashboard_data"].ToString());
    list.Add(dev);
}  
return View(list);

/// <summary>
/// Display measure result.
/// </summary>
public async Task<ActionResult> DisplayMeasure()
{
    // TODO Not hardcoded.
    var device_id = "*******";
    var token = await getToken();

    // Get measure
    var method = "getmeasure";
    // Add parameters
    var parameters = "?access_token=" + token.Token +
        "&device_id=" + device_id +
        "&type=Temperature" +
        "&scale=1week";
    JObject devs = await GetData(netatmo.getDataURL() +
        method + parameters);

    var jTokenBody = devs["body"];  
    var list = new List<TemperatureMeasurement>();  
    foreach (var item in jTokenBody)
    {
        var begtime =
            DateTime.Now.AddTicks(-(long)item["beg_time"]).Ticks;
        var stepTime = (long)item["step_time"];  
        var value =
            item["value"].Children().Values<decimal>();
        foreach (var v in value)
        {
            list.Add(new TemperatureMeasurement(device_id, new DateTime(begtime), v));
            begtime = begtime + stepTime;
        }
    }
```
return View(list);

/// <summary>
/// Method for making get requests.
/// </summary>
/// <param name="request">The requested url with parameters.</param>
private async Task<JObject> GetData(string request)
{
    var client = new WebClient { };
    return JObject.Parse(await client.DownloadStringTaskAsync(request));
}

A.2 AuthenticateController

class AuthenticateController : Controller
{
    private PublicAPI api;
    private string client_id = "";
    private string client_secret = "";

    /// <summary>
    /// Index page
    /// </summary>
    public ActionResult Index()
    {
        return View();
    }

    /// <summary>
    /// Login page
    /// </summary>
    public ActionResult Login()
    {
        var LoginModel = new LoginModel();
        return View(LoginModel);
    }

    /// <summary>
    /// Handling login.
    /// </summary>
    /// <param name="user"></param>
    [HttpPost]
    public async Task<ActionResult> Login(LoginModel user)
var username = user.Email;
var password = user.Password;
var apiName = user.SelectedItem;
var token = await AuthenticateUser(apiName, username, password);

// Store the token in a cache, preferably should been stored in a db.
AppCache.AppCache.Insert("token", token,
    AppCache.CacheTime.DEFAULT);
return RedirectToAction("Netatmo", "Home");

/// <summary>
/// Authenticate User by the OAuth2.0. "Resource Owner Password Credentials Grant."
/// </summary>
/// <param name="APIName"> </param>
/// <param name="username"> </param>
/// <param name="password"> </param>

public async Task<AuthenticationToken> AuthenticateUser(string APIName, string username, string password)
{
    var parameters = "&grant_type=password";

    if (APIName.Equals("Netatmo"))
    {
        api = new APINetatmo();
        client_id = WebConfigurationManager.AppSettings.
            Get("client_id_netatmo");
        client_secret = WebConfigurationManager.AppSettings.
            Get("client_secret_netatmo");
        parameters += "&scope=read_station read_thermostat
            write_thermostat";
    }

    parameters += "&client_id=" + client_id;
    parameters += "&client_secret=" + client_secret;
    parameters += "&username=" + username;
    parameters += "&password=" + password;

    client = new WebClient {
        client.Headers[HttpRequestHeader.ContentType] = "application/x-www-form-urlencoded;" + "charset=UTF-8";
    var httpResult = await client.UploadStringTaskAsync(api,
        parameters));
var jsonResult = JObject.Parse(httpResult);
var token = new AuthenticationToken();
token.Token = (string)jsonResult["access_token"];
token.RefreshToken = (string)jsonResult["refresh_token"];
token.ExpiresAt = (int)jsonResult["expires_in"];
AppCache.AppCache.Insert("token", token,
   AppCache.CacheTime.DEFAULT);
return token;
}

/// <summary>
/// Refresh the users token.
/// </summary>
public async Task<AuthenticationToken> RefreshToken()
{
    throw new NotImplementedException();
}

/// <summary>
/// Authenticate User by the OAuth2.0. "Authorization Code Grant"
/// </summary>
public ActionResult AuthenticateUserByRedirect()
{
    Response.BufferOutput = true;

    var _uri = new Uri(@"https://localhost:44300/Data/FetchRedirect");
    var _scope = "read_station%20read_thermostat%20write_thermostat";
    var _state = "aStringWithSomething";
    var _client_id = WebConfigurationManager.AppSettings.Get("client_id_netatmo");

    Response.Redirect("https://api.netatmo.net/oauth2/authorize?" +
    "client_id=" + _client_id +
    "&redirect_uri=" + _uri +
    "&scope=" + _scope +
    "&state=" + _state);
    throw new NotImplementedException();
    return RedirectToAction("DisplayDevices", "Home");
}
A.3 HomeController

```csharp
public class HomeController : Controller
{
    public ActionResult Index()
    {
        return View();
    }

    public ActionResult Netatmo()
    {
        return View();
    }
}
```

A.4 LoginModel

```csharp
public class LoginModel
{
    [Required()]
    [Display(Name = "Email Address")]
    [DataType(DataType.EmailAddress)]
    public string Email { get; set; }

    [DataType(DataType.Password)]
    public string Password { get; set; }

    public string SelectedItem { get; set; }

    public IEnumerable<PublicAPI> apis = new List<PublicAPI>
    {
        new APINetatmo(),
        new APIInest(),
        new APIEcobee()
    };
}
```

A.5 AuthenticationToken

```csharp
/// <summary>
/// Authentication token, refresh token and expires in.
/// </summary>
public class AuthenticationToken
{
    public string Token { get; set; }
    public int ExpiresAt { get; set; }
    public string RefreshToken { get; set; }
}
```

A.6 PublicAPI

```csharp
public interface PublicAPI
{
    String getName();
}
List<String> getMethods();
Dictionary<String, Uri> getMethodsURL();
Uri getAuthenticationURL();
Uri getDataURL();

A.7  APINetatmo

/// <summary>
/// The Netatmo API
/// </summary>
public class APINetatmo : PublicAPI
{
    public Dictionary<string, Uri> getMethodsURL()
    {
        var dict = new Dictionary<string, Uri>();
        dict.Add("Temperature", new Uri("https://api.netatmo.net/api/getmeasure"));
        dict.Add("Humidity", new Uri("https://api.netatmo.net/api/getmeasure"));
        return dict;
    }

    public Uri getAuthenticationURL()
    {
        return new Uri("https://api.netatmo.net/oauth2/token");
    }

    public Uri getDataURL()
    {
        return new Uri("https://api.netatmo.net/api/");
    }

    public string getName()
    {
        return "Netatmo";
    }

    public List<string> getMethods()
    {
        var list = new List<string>();
        list.Add("Temperature");
        list.Add("Humidity");
        return list;
    }

    public override string ToString()
    {
        return getName();
    }
}
A.8 Device

/// <summary>
/// Device class
/// </summary>
public class Device
{
    public string DeviceId { get; set; }
    public string Type { get; set; }
    public IList<string> OwnerId { get; set; }
}

A.9 NetatmoDevice

/// <summary>
/// A Netatmo device
/// </summary>
public class NetatmoDevice : Device
{
    public NetatmoMeasurements Measurements { get; set; }
    public NetatmoDevice()
    {
        Measurements = new NetatmoMeasurements();
    }
}

A.10 NetatmoMeasurements

/// <summary>
/// Measurements of a netatmo device
/// </summary>
public class NetatmoMeasurements
{
    public double AbsolutePressure { get; set; }
    public double time_utc { get; set; }
    public double Noise { get; set; }
    public double Temperature { get; set; }
    public double Humidity { get; set; }
    public double Pressure { get; set; }
    public double CO2 { get; set; }
    public double date_max_temp { get; set; }
    public double date_min_temp { get; set; }
    public double min_temp { get; set; }
    public double max_temp { get; set; }
}

A.11 SensorMeasurementFactory

/// <summary>
/// Sensormeasurement factory
/// </summary>
/// </summary>
public class SensorMeasurementFactory
{
    public static SensorMeasurement Create(string id, string measurementType, DateTime timeStamp, decimal value)
    {
        switch (measurementType.ToLower())
        {
        case "temperature":
            return new TemperatureMeasurement(id, timeStamp, value);
        case "humidity":
            return new HumidityMeasurement(id, timeStamp, value);
        default:
            return new GenericMeasurement(id, measurementType, timeStamp, value);
        }
    }
}

A.12 GenericMeasurement

/// <summary>
/// A generic measurement.
/// </summary>
public class GenericMeasurement : SensorMeasurement
{
    public string Type { get; set; }

    public GenericMeasurement(string modelKey, string type, DateTime timeStamp, decimal value)
        : base(modelKey, timeStamp, value)
    {
        Type = type;
    }
}

A.13 SensorMeasurement

/// <summary>
/// Sensor measurement
/// </summary>
public abstract class SensorMeasurement
{
    public string ID { get; set; }

    /// <summary>
    /// Key/ID of the device.
    /// </summary>
    /// </summary>
/// TimeStamp of the measurement.
/// </summary>
public DateTime TimeStamp { get; set; }

/// <summary>
/// Value of the measurement.
/// </summary>
public decimal Value { get; set; }

protected SensorMeasurement(string id, DateTime timestamp, decimal value)
{
    ID = id;
    TimeStamp = timestamp;
    Value = value;
}

A.14 TemperatureMeasurement

/// <summary>
/// Temperature measurement.
/// </summary>
public class TemperatureMeasurement : SensorMeasurement
{
    public TemperatureMeasurement(string id, DateTime timestamp, decimal value)
    : base(id, timestamp, value) { }
}

A.15 Index.cshtml page for the HomeController

<div class="row">
    <h2>Welcome to the Smart Home API page.</h2>
</div>

A.16 Login.cshtml page for the AuthenticateController

@model LoginModel
@{
    ViewBag.Title = "Select a login option";
}
<h2>@ViewBag.Title.</h2>

@section aria-label="Main content" role="main">
    <div class="appViewContent">
        @using (Html.BeginForm("Login", "Authenticate"))
        {
            <h3>
                <label for="api">Select API</label><br />
                @Html.DropDownListFor(m => m.SelectedItem, new SelectList(Model.apis))
            </h3>
        }
    </div>
</section>
<h3>Login at a API page:</h3>
@Html.ActionLink("Login at Netatmo", "LoginRedirect", "Authenticate") <br />
@Html.ActionLink("Login at Nest", "LoginRedirect", "Authenticate") <br />
@Html.ActionLink("Login at Ecobee", "LoginRedirect", "Authenticate") <br />
</div>

A.17 DisplayDevice.cshtml page for the DataController

@model IEnumerable<NetatmoDevice>
{@
    ViewBag.Title = "DisplayData";
}

<h2>Display Devices</h2>
<div class="results">
    @foreach (var item in Model) {
    
    <h3>Device ID: @item.DeviceId <br /></h3>
    <h4>Absolute Pressure: @item.Measurements.AbsolutePressure <br />
    time_utc: @item.Measurements.time_utc <br />
    Noise: @item.Measurements.Noise <br />
    Temperature: @item.Measurements.Temperature <br />
    Humidity: @item.Measurements.Humidity <br />
}
A.18 DisplayMeasure.cshtml for the DataController

@model IEnumerable<TemperatureMeasurement>
@{
    ViewBag.Title = "DisplayMeasure";
}
<h2>DisplayMeasure</h2>
<table class="table">
    <tr>
        <th>
            @Html.DisplayNameFor(model => model.ID)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.TimeStamp)
        </th>
        <th>
            @Html.DisplayNameFor(model => model.Value)
        </th>
    </tr>
    @foreach (var item in Model)
    {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.ID)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.TimeStamp)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Value)
            </td>
        </tr>
    }
</table>