jamk.fi

Weather API

Markus Paappanen

Bachelor's thesis
December 2017
Technology, communication and transport
Degree Programme in Software Engineering

Jyväskylän ammattikorkeakoulu
JAMK University of Applied Sciences

jamk.fi

Description

Author(s)	Type of publication	Date
Paappanen, Markus	Bachelor's thesis	December 2017
		Language of publication: English
	Number of pages: 28	Permission for web publication: x

Title of publication
Weather API

Degree programme
Degree Programme in Software Engineering

Supervisor(s)
Huotari Jouni, Väänänen Olli

Assigned by
Enegia Consulting Oy

Abstract

Creating a Weather API was a project assigned by Enegia Consulting Oy, a subsidiary of Enegia Group Oy. Enegia Group's business idea is to offer services to customers helping them to reduce energy related costs.

The objective of the project by the Enegia Consulting Oy was to plan and implement Weather API for other Enegia services to consumers. The Weather API would serve weather information to a given postal code, time frame and resolution to all authorized requests to the API.

The weather information was gathered from the open data service of the Finnish Meteorological Institute, and stored to the database using Weather API endpoint. The information was gathered daily using Azure WebJob.

The Weather API was written with C# using .NET Core 1.0 development platform. The application used time series database InfluxDB to store the weather information. Other data such as the postal code to geolocation mapping was stored to Azure SQL database in Microsoft Azure. The solution ran in Microsoft Azure App services.

The Weather API was released to production in May 2017 and since then it has been running continuously without interruption. Further development has been planned, however, the implementation has not yet been started.

Keywords/tags (subjects)
API, .NET Core, InfluxDB, Azure SQL, XUnit, OpenAPI Specification

Miscellaneous (Confidential information)

jamk.fi

Kuvailulehti

Tekijä(t)	Julkaisun laji	Päivämäärä
Paappanen, Markus	Opinnäytetyö, AMK	Joulukuu 2017
	Sivumäärä 28	Julkaisun kieli Englanti
		Verkkojulkaisulupa myönnetty: x

Työn nimi
Weather API

Tutkinto-ohjelma
Ohjelmistotekniikan koulutusohjelma

Työn ohjaaja(t)
Jouni Huotari, Olli Väänänen

Toimeksiantaja(t)
Enegia Consulting Oy

Tiivistelmä

Säätieto-ohjelmointirajapinta-projekti toteutettiin Enegia Group Oy:n tytäryhtiön Enegia Consulting Oy:n toimeksiannosta. Enegia Groupin liiketoiminta-ajatuksena on tarjota asiakkailleen palveluita, jotka auttavat vähentämään energiakustannuksia.

Säätieto-ohjelmointirajapinta-projektin (lyhyemmin Weather API) tavoitteena oli suunnitella ja toteuttaa säätietorajapinta, jota muut Enegian palvelut voisivat käyttää. Weather API palvelisi säätietoja tietylle postinumerolle, aikavälille ja resoluutiolle kaikille valtuutetuille pyynnöille.

Säätiedot kerättiin Ilmatieteen laitoksen avoimesta tietopalvelusta ja tallennettiin tietokantaan Weather API:n rajapinnan avulla. Tiedot kerättiin päivittäin Azure WebJobin avulla.

Weather API kirjoitettiin C # -ohjelmalla käyttäen .NET Core 1.0 -kehitysalustaa. Sovellus käytti aikasarjatietokanta InfluxDB:tä säätietojen tallentamiseen. Muut tiedot, kuten postinumeron paikoitustieto, tallennettiin Azure SQL -tietokantaan Microsoft Azure -palvelussa. Ratkaisu toteutettiin Microsoft Azure App -palveluissa.

Weather API julkaistiin tuotantoon toukokuussa 2017, ja sen jälkeen se on ollut käynnissä keskeytyksettä. Jatkokehittämistä on suunniteltu, mutta toteutusta ei ole vielä aloitettu.

Avainsanat (asiasanat)
API, .NET Core, InfluxDB, Azure SQL, XUnit, OpenAPI Specification

Muut tiedot

Contents

Figures

Tables

1 Introduction

1.1 Company

Enegia Consulting Oy is a subsidiary of Enegia Group Oy (later Enegia). Enegia Consulting Oy mainly produces the IT, design, consulting, and training services required by Enegia Group. Enegia Group Oy also includes Enegia Portfolio Services Oy, Enegia Sweden AB and intStream Oy. Enegia's head office is located in Hämeenlinna and branches in Helsinki, Jyväskylä and Stockholm. (Yritys [Company] n.d.)

Enegia is the market leader in energy management in Finland. The company offers solutions for energy acquisition, sales and operational efficiency. The purpose of a customer-oriented service is to reduce the use of energy and related costs in accordance with the company's slogan. The customers include municipalities and other real estate managers, energy companies and large companies from various industries such as Kone, Patria and Valio. Half of Finland's top 100 companies use Enegia's services. The company manages nearly 25% of Finland's electricity consumption. (Yritys [Company] n.d.)

The company's competitive advantage is independence, as it does not itself generate energy or become a major user. The Group, which was established in 1995, employs around 160 people and has a turnover of EUR 18 million in 2015. (Yritys [Company] n.d.)

1.2 Assignment

The aim of the thesis was to create programming interface for a weather information application (later Weather API) using the new .NET Core software component library. The function of the Weather API is to provide weather information to other services in Enegia, mainly for energy reporting. Energy reporting is a service in EnerKey that makes reports from energy consumptions. EnerKey is an energy management system that helps organizations identify possibilities to save energy. With EnerKey, companies can monitor and plan energy consumption. (EnerKey 2017.)

1.3 Methods

The Weather API project was conducted in two-week sprints alongside Enegia's normal software development team. The development process contained sprint planning, retrospectives, daily standups and sprint reviews, which all are part of agile development framework *scrum*. (What is Scrum Methodology. n.d.)

Scum is a sub-group of agile with the focus on delivering new software capability every two to four weeks. Scrum tackles complexity in software development process by making information transparent, allowing inspection and adaption based on current conditions instead of predicted conditions. This allows better handling of ever-changing requirements, time estimations, cost and resources. (What is Scrum Methodology. n.d.)

Planning documentation was written to Enegia's internal Confluence site. Confluence is a team collaboration software that allows teams to publish, organize and access company information in one centralized location. (Confluence 2017).

2 Requirements

Enegia's business is based on services, that highly rely on information system solutions. The company has internal and external tools that are consuming data sources such as API's and internal data storages. Weather API is one of these valuable data sources.

Specifications were not finalized before coding phase in the spirit of agile software development (Manifesto for Agile Software Development 2001)

2.1 Functional requirements

The functional requirements for Weather API were:

- provide weather information in hourly accuracy,
- links weather information to data about customer premises
- and stores weather data from 2013 onwards.

Later aggregation and sum of weather data were added to the requirements.

2.2 Other requirements and constraints

Non-functional requirements for Weather API were mainly related to performance to be able to response to requests in a reasonable time. In normal circumstances everything under two seconds can be considered reasonable time.

The usage of Weather API requires authentication and authorization for business and cost related reasons.

While one of the used data sources is the Finnish Meteorological Institute, Creative Commons Attribution 4.0 International license agreements for open data need to be followed (License n.d.).

3 Technologies

3.1 .NET CORE

.NET Core is a general-purpose open source development platform maintained by Microsoft and the .NET community on GitHub. It supports Windows, MacOS, and many Linux environments. (Lander and Wenzel 2017.)

Behind the selection of .NET Core were characteristics, such as cross-platform support, good tooling, compatible to .NET framework, open source and it is supported by the Microsoft. (Lander and Wenzel 2017.)

3.2 Entity Framework Core

Entity Framework Core (later EF Core) is an object-relational mapping (ORM) technology that allows databases to work directly from .NET classes on a variety of platforms, which eliminates the need to write the layer logic of the data usage. (Miller and Vega 2017)

To highlight some of the feature, Entity framework core supported code first migrations, allowing database changes to be made from the code. Migrations will also provide a way to apply changes to the database incrementally, allowing preserving existing data in the database. (Lambson 2017.)

3.3 InfluxDB

In the weather API project, there were two different data sources, Azure SQL and InfluxDB.

InfluxDB is a database optimized for time series processing, which allows for large amounts of data to be queried and aggregated over time. (InfluxDB Version 1.3 Documentation 2017.)

InfluxDB is a database that is optimized for time-series data performance, that leads to some tradeoffs at the cost of functionality. These tradeoffs make InfluxDB good at handling aggregate data and large data sets. The database can also handle a high volume of reads and writes. (Design Insights and Tradeoffs in InfluxDB. n.d.)

Rapid search of data from large mass and aggregation was the main reason to experiment with InfluxDB's potential in this pilot project. In the future this kind of time-series database could resolve problems with traditional databases when the data reaches critical volume.

3.4 Azure SQL Database

Azure SQL Database is a general-purpose relational database service for Microsoft's Azure cloud computing service. The SQL database provides scalable performance and database pooling. (Rabeler 2017)

Azure SQL was chosen because it is easy to setup, easy to use and scales up if needed. Therefore, there was no need for deep database management skills, which allowed to centralize resource use to the development of the application code. In addition to the weather data time series, there were non-time series data such as weather station information and geolocation information. All non-timeseries data were stored to Azure SQL database.

3.5 Microservice architecture pattern

Microservice is an architectural model to make services. In the model, the service package is broken into small services. This is the opposite of the monolithic model

that brings together all the services. As the services remain small, they are easier to understand, easier to scale and each service can be developed and published as an independent component. In addition, all long-term commitments to the technology stack are eliminated. (Richardson 2017.)

Because weather information was easy to differentiate as a new service, it was a good idea to create it as a microservice and thus enable it to be used in other services in the future.

Microservice thinking had already moved ahead of the weather information programming interface in the implementation of the facility information programming interface.

3.6 Repository pattern

In repository pattern repository contains the logic to retrieve the data and then maps it to the entity model. The business logic should be agnostic to the type of data that comprises the data source layer. Repository is a layer between client business logic and the data source as can be seen in figure 1. (The repository pattern 2017.)

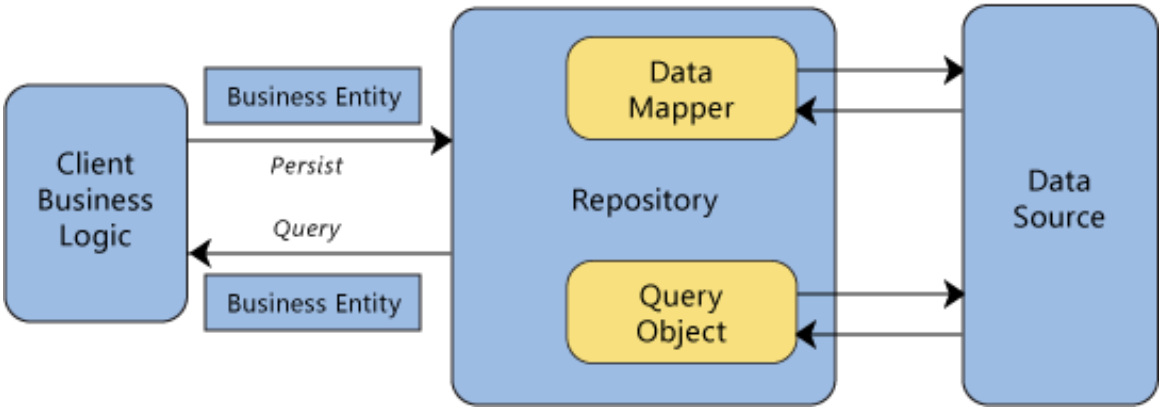

Figure 1. Interactions of the repository (The repository pattern, 2017)

Repository pattern allowed *Dependency Injection* (DI) of repositories to controllers, that makes possible business logic to be dependent only of the abstractions (Interfaces) of the repositories. That means database can be changed without changing the business logic of the application. (Smith and Addie 2016.)

3.7 OpenAPI Specification

The OpenAPI Specification (formerly known as the Swagger Specification) defines a non-language specific standard to describe RESTful API's. The definition can be used to generate servers and clients for multiple programming languages to consume the API. (OpenAPI Specification 2017.)

OpenAPI Specification provided way to document the Weather API, such a way that implementation of the service could be done with a minimum work to multiple different frameworks and languages.

3.8 XUnit

XUnit is an open source unit testing tool for the .NET Framework, that can unit test projects written in C#, F# and VB.NET. It was written by the original inventor of the NUnit v2. (Wilson 2016.)

XUnit offers many advantages over the other unit testing frameworks, such as more assertions, easier to follow and debug and less attribute decoration. Also building test data in constructors and tearing it down in dispose is more natural to C# developer. XUnit supports data driven tests called theories using attributes. (Killeen and Stewart. 2017.)

4 Implementation

4.1 First draft

The planning began in November 2016 and lasted two months. In the design phase, a weather data source, data models and technologies were designed. The purpose of the design was to produce a weather information programming interface and the services it needs. The first draft of the architectural design (Figure 2) was created in the planning process, after which the implementation phase started.

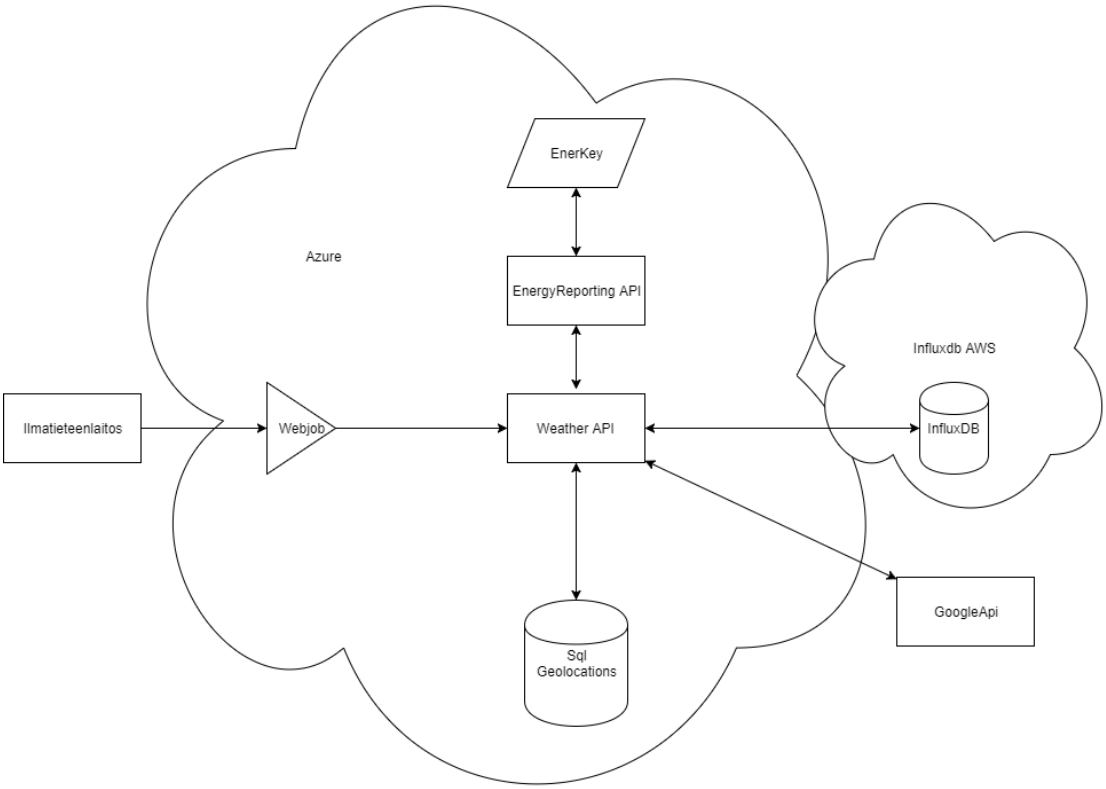

Figure 2. First draft

4.2 Weather data

The planning was started by mapping various weather information services. The preliminary mapping was based on the open data interface of the Finnish Meteorological Institute. In the actual survey, the Foreca weather API and the Norwegian yr.no weather service were also studied.

The interface provided by Foreca was good and it could have been scaled to other countries. The interface could have been directly used, making the whole project unnecessarily. Foreca's programming interface was good, however, far too extensive and expensive to meet requirements.

The Norwegian yr.no weather service provides weather information for 10 million locations worldwide in XML format. The instructions and requirements are in Norwegian because the service is produced in Norwegian taxation, and the service providers are afraid that the demand for services will grow uncontrollably if the instructions

are provided in English. (Information about the free weather data service - yr.no 2016)

The interface offered by Yr.no seemed to be the best option, scaling well to areas outside Finland. Unfortunately, the API itself was getting a complete reform in 2017. (Vêrvarsel and XML-format [Weather forecast in XML format] 2017).

Finally, it was decided to use the open data interface of the Finnish Meteorological Institute, which provided the necessary information for the present need at reasonable cost.

4.3 Weather API

The basic operation of the Weather API was described in the sequence pattern (Figure 3). The Weather API was designed for use through the High-Level Programming Interface (Energy Reporting API). This was later changed so the Weather could handle direct requests from all the other services. The first call would check whether the postal code was previously requested. After that, the programming interface would search for the coordinates of the postcode from the Google Programming interface and save it in the database.

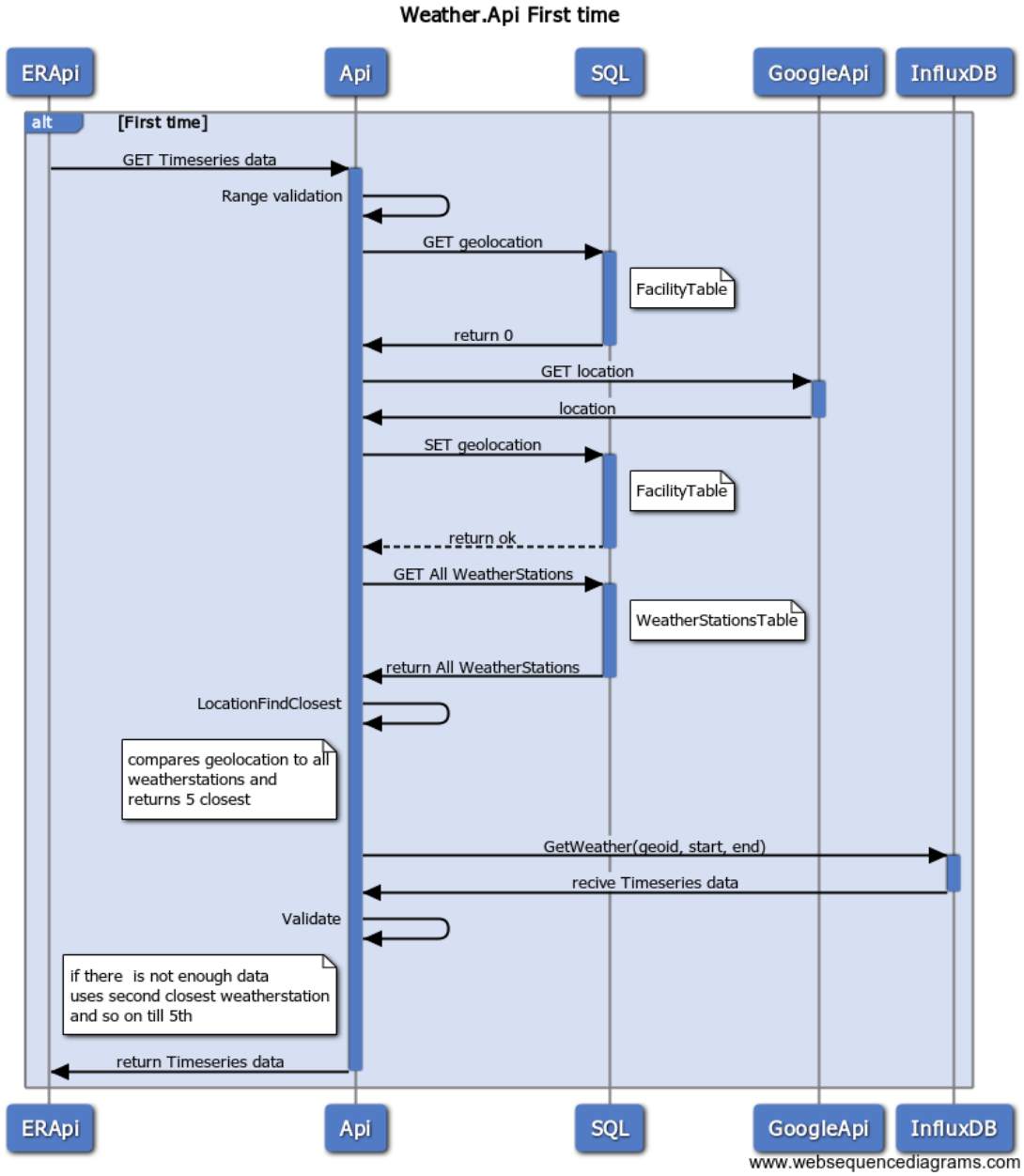

Figure 3. First call sequence diagram

After receiving the postcode, the coordinates of the programming interface would search for weather stations on their own table and compare the location of the nearest weather station. EF Core did not yet support spatial data types in the database, however, this is on the roadmap of the development team. (Miller and Vega 2017). Spatial data types would allow comparisons to occur in the database, thus speeding up the programming interface.

With Weather Information, the Weather API would search the InfluxDB database for the desired time horizon meteorological data. According to the original plan, the integrity of the data had to be verified, however, later the time was added to the

WeatherStation table when the weather station was in use, which made the interface faster, more logical and easier to maintain.

When calling the programming interface for a second time (Figure 4), the Google Interface query and the storage facility table are avoided because the information is already stored on that table.

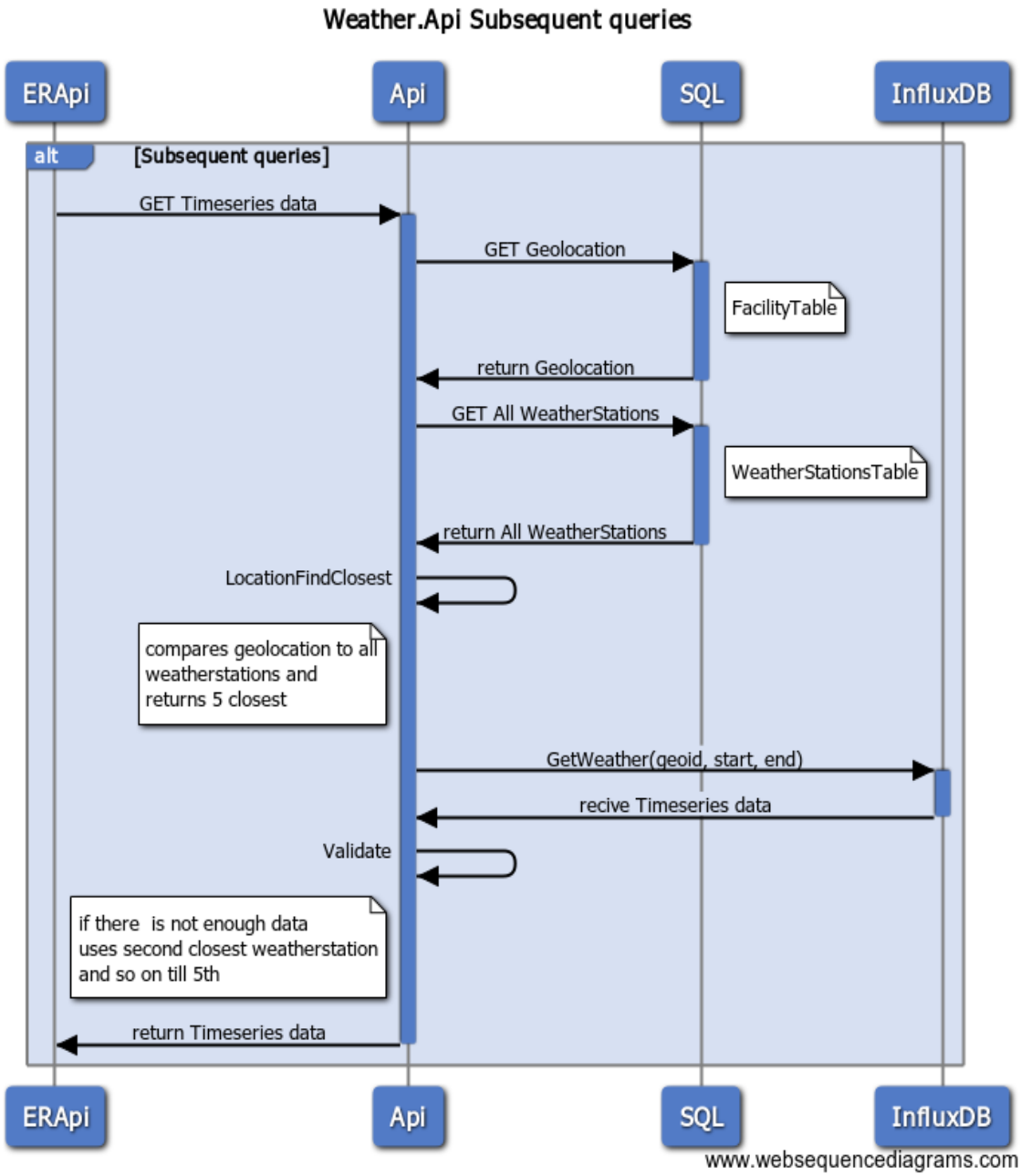

Figure 4. Later call sequence diagram

4.4 InfluxDB data model

The InfluxDB data model was designed to be simple; yet, it left the opportunity to supplement the model with other data such as wind speed.

Designing avoided bad practices such as writing information to the measurement name, adding more than one value to the individual tag, and creating too many series using tags. Good practices include saving data to tags if the information contained frequently asked metadata, or want to use the information in a group by-expression. Fields are not indexed and should be preferred if cardinality is to be avoided, or the value is interpreted as something other than a string data type, such as a number for calculation. (Schema Design. 2017).

4.5 Weather WebJob

Azure WebJob is a feature in Azure App Service that enables running programs or scripts in the same context as the web app. There are two types of WebJobs, *continuous* and *triggered*. Continuous WebJob starts when the WebJob is created and runs usually inside an endless loop. Triggered WebJob must be started manually or on a schedule. (Gailey, Cephas and Dykstra 2017.)

Weather WebJob was planned as a triggered WebJob and it has a simple task, download weather information from the Finnish Meteorological Institute once a day and sends them to the weather information programming interface. This is illustrated in the sequence diagram in Figure 5. In the implementation phase, the validity period of the weather station was added to the weather station table. This changed the sequence that Weather API would also change weather station validity times to correspond to the newly added values, which eliminated the need to request weather information from the five closest stations.

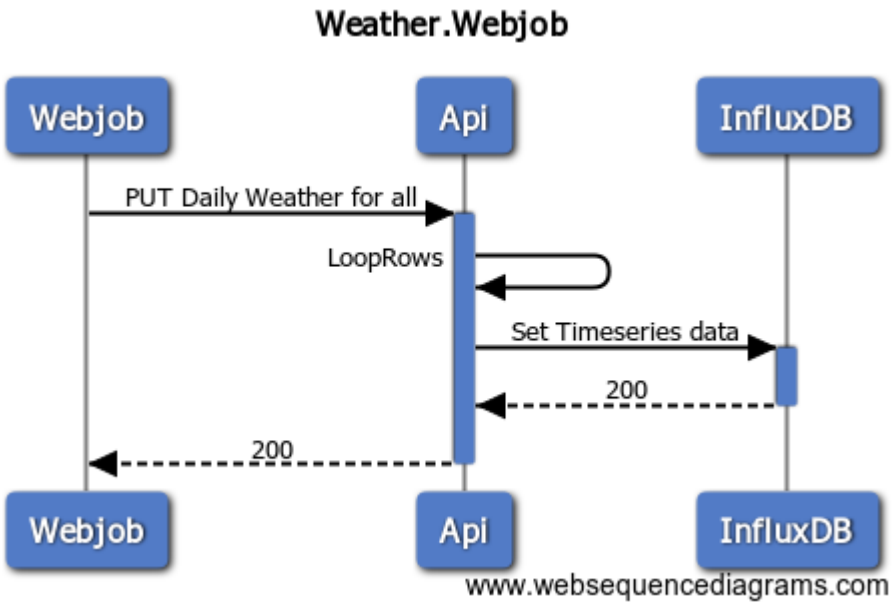

Figure 5. Webjob sequence diagram

4.6 Azure SQL

The database and model were created with Code First approach. In Code First database is created through model classes using entity framework. Tables (Table 1) were simple and had no relations. They inherited base class entity from Enegia's internal libraries, that gave Created, CreatedBy, LastModified and LastModifiedBy fields. Migrations history table is for entity framework migrations.

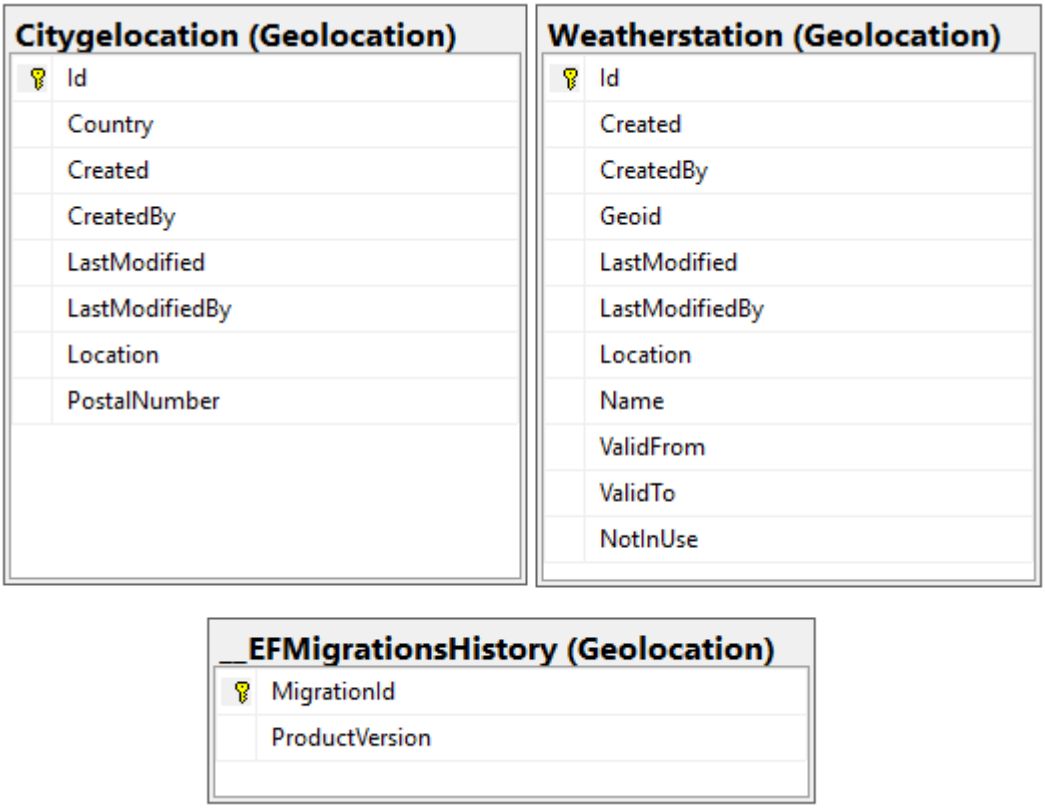

Table 1. Azure SQL tables

5 Continuous integration and testing

5.1 Integration testing with xUnit

Test automation was implemented as integration tests, testing endpoints of the Weather API. Unit testing was ignored to minimize resources used for testing. Instead of concentrating on narrow scope issues, such as functions and classes, integration tests provided more robust tests at decreased maintenance cost. Better coverage was achieved with less effort for a constantly developing product. The major advantage of skipping unit tests was avoiding mockup unit dependencies.

Test automation was implemented with xUnit unit testing framework and test data generation was also handled with it. Before the test run, XUnit generates test data based on provided data structures in fixture and saves the test data to databases by calling the Weather API. During the test data creation process, xUnit also verifies the endpoint for storing data to the Weather API. (Appendix 2) This arrangement guarantees that test data is always consistent and manageable.

There are two types of tests in xUnit, facts and theories. Facts are tests that should always pass regardless of the used test data. Theories are data driven tests that are depend on a particular set of test data. The tests can contain fixture where test context can be initialized, and then shared between all the tests in the test class. (Wilson 2017.)

There are only two theory tests, one for basic weather data query endpoint and one for average weather data endpoint. Using inline data parameters, it is possible to call those endpoints multiple times with different starting parameters. This allowed testing different timeframe and resolution combinations effectively. (Appendix 1)

5.2 Test automation

Every commit to the develop branch was done by using pull request from the feature branch. Accepting pull request would trigger pull request gate, that will build the solution, run tests and publish test results. Development branch will always have solution that builds and a solution that passes tests.

5.3 Testing environments

Most of the manual testing was carried out locally using Postman and Swagger UI. In the local environment, it was possible to disable authorization, making debugging easier. In the cloud, there are three different environments: development, testing and production.

5.4 Dashboard and results

Code coverage was measured using Open Cover tool, and the results of the test coverage are shown in the visual studio team service dashboard (Figure 7). Measuring is the first step towards quality.

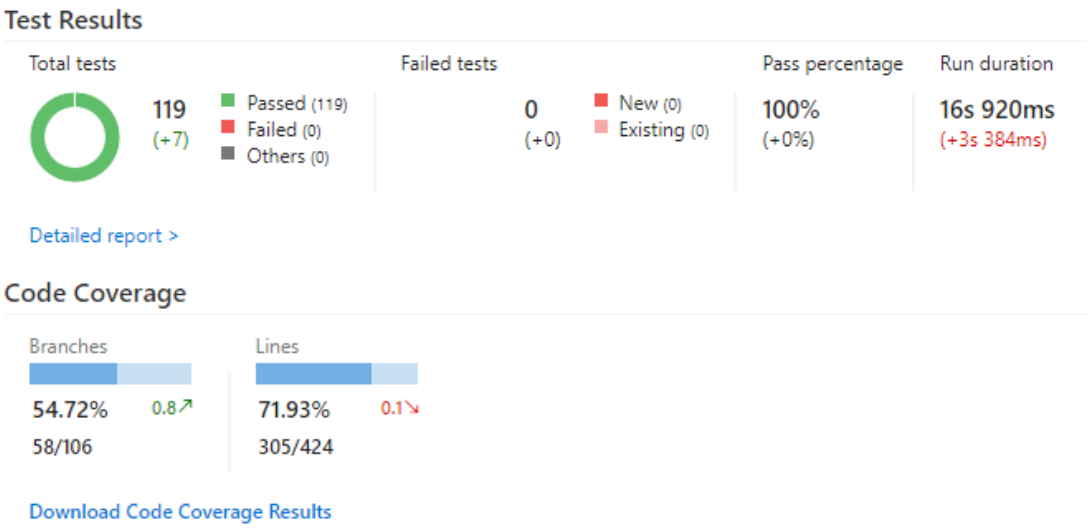

Figure 6. Azure portal dashboard. Provides test results and code coverage information.

5.5 Acceptance testing

Acceptance testing was carried out by a software project specialist at Enegia after the frontend implementation to the EnerKey energy reporting services was done.

5.5.1 Functional criteria

The first functional criterion was that user should get weather data in every time frame that the energy reporting uses. This included the following time frames: one hour, one day, seven days and one to twelve months' range.

The second functional criterion was that user should get weather data in very time resolution in energy reporting services including resolutions: one hour, one day, seven days, one month, three months, six months, nine months and one year.

User should get weather data for every valid postal code in Finland.

User should have yesterday's weather data after 3:00 am.

User should see where the data is collected (weather station name)

5.5.2 Non-functional criteria

Responses should use the same formatting as other API (upper PascalCase).

5.6 Information security

The code was written using the known frameworks and it was reviewed by other developers at Enegia. These are the building blocks of secure development.

5.7 Performance

the only performance criterion was that the weather data should load faster than the queries from the energy reporting API. This was tested and approved manually.

6 Final Product

6.1 Weather information in EnerKey

The result was the weather API, which is a complement to the EnerKey service reports (Figure 7). Users can quickly see how outside weather affects their energy consumptions, such as heating and electricity.

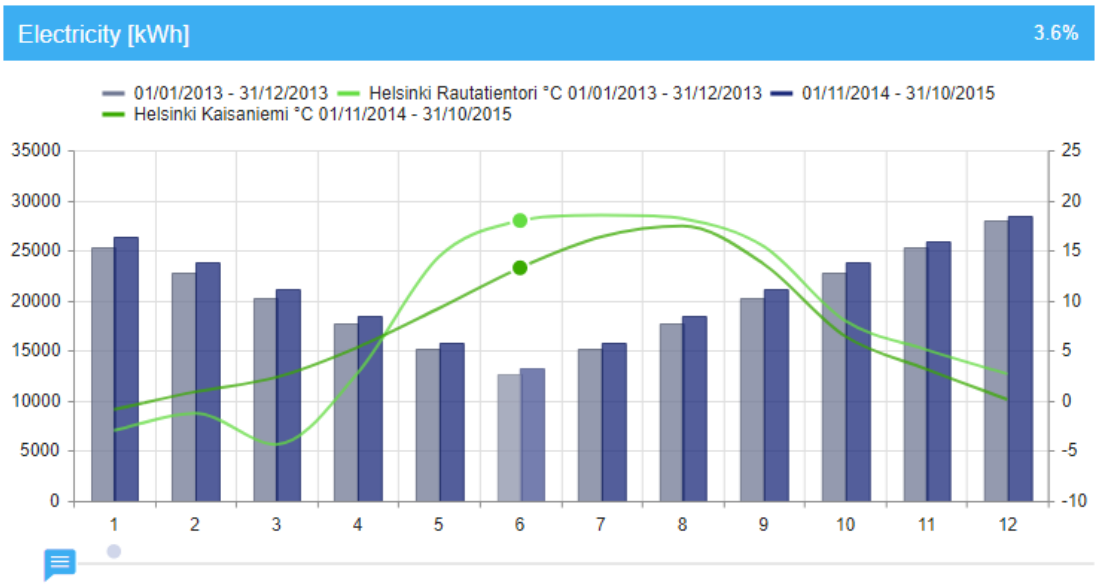

Figure 7. Graph in EnerKey web portal

As a by-product a great amount of information about potential new technologies such as InfluxDB, Azure SQL, and .NET Core were acquired.

6.2 Query

Weather API query (Figure 8) contains PostalCodeCountry body that accepts an array of postal code and country combinations. This way EnerKey can query weather information for multiple facilities at once. Next there is resolution and time frame that both accept values in ISO 8601 date format. After that the starting points are given. Start has key and value, key can be any string and value must be time given in ISO 8601 coordinated universal time (UTC) format. In figure 9 query example, resolution PT1H and time frame P1M returns hourly weather data for a month for every start point given.

Query

```
{
  "PostalCodeCountry": [
    {
      "PostalCode": "40100",
      "Country": "FI"
    },
    {
      "PostalCode": "40500",
      "Country": "FI"
    }
  ],
  "Resolution": "PT1H",
  "TimeFrame": "P1M",
  "Start": [
    {
      "Key": "Front defined: usually query ",
      "Value": "2017-01-01T00:00:00.000Z"
    },
    {
      "Key": "Front defined: usually query 2",
      "Value": "2015-01-01T00:00:00.000Z"
    }
  ]
}
```

Figure 8. Example query

6.3 Response

Response uses Enegia's timeseries container (figure 10) to deliver responses. Postal codes work as keys that contain many sub containers. Weather API only uses Value container, where keys defined in query work as a key for the response data group. Every measurement has a measurement point (weather station), value and timestamp.

Response


```
{
  "40100": {
    "Aggregates": {},
    "ErrorCodes": [],
    "SubSeries": {},
    "Values": {
      "Front defined: usually query ": [
        {
          "WeatherStation": "Jyväskylä lentoasema",
          "Value": -4.70981182795699,
          "Timestamp": "2017-01-01T00:00:00Z"
        }
      ],
      "Front defined: usually query 2": [
        {
          "WeatherStation": "Jyväskylä lentoasema",
          "Value": -6.45443548387096,
          "Timestamp": "2015-01-01T00:00:00Z"
        }
      ]
    },
    "_Id": "40100",
    "_IdType": "PostalCode"
  },
  "40500": {
    "Aggregates": {},
    "ErrorCodes": [],
    "SubSeries": {},
    "Values": {
      "Front defined: usually query ": [
        {
          "WeatherStation": "Jyväskylä lentoasema",
          "Value": -4.70981182795699,
          "Timestamp": "2017-01-01T00:00:00Z"
        }
      ],
      "Front defined: usually query 2": [
        {
          "WeatherStation": "Jyväskylä lentoasema",
          "Value": -6.45443548387096,
          "Timestamp": "2015-01-01T00:00:00Z"
        }
      ]
    },
    "_Id": "40500",
    "_IdType": "PostalCode"
  }
}
```


Figure 9. Response Example

7 Discussion and conclusions

Weather API went to production in 30 May 2017 as a soft launch, which means users were not notified about this new feature. Only one bug has been found since that and it was fixed on 3 August 2017.

Research, planning and implementation phases took roughly 320 hours according to Jira issues. This includes only time before the launch and not project meetings like such as meetups and project planning sessions.

All requirements were fulfilled, and Weather API has been fully operational since the launch.

7.1 Lessons learned

There is no need to reinvent the wheel, there are plenty of good patterns and principles already figured out. Basic principles such as *don't repeat yourself* (DRY) and SOLID principles, should always be followed. DRY pattern was introduced by Andy Hunt and Dave Thomas in their book Pragmatic Programmer and DRY pattern dictates that every piece of system knowledge should have one authoritative, unambiguous representation (Venners 2003). SOLID acronym contained five principles that Robert Martin (2000) introduced in his article about Design Principles and Design Patterns. Five principles are: Single Responsible Pattern, Open/Closed principle, Liskov Substitution principle, Interface Segregation principle and Dependecy Inversion principle (Martin 2000).

Developer should pursue clean and maintainable code. Over engineering six-layer logic when two is more than enough is counter intuitive.

Planning good test cases will save developers time in the long run. It is important to priorities testing cases and at least cover every basic case. Writing tests that will not test anything worth testing and are hard to maintain, will take time from the development process.

Code reviews are good and inexpensive way to improve quality. Every developer has their unique way of looking challenges and can catch issues that others have missed. Asking for guidance and opinions will often lead to better result than working solo.

7.2 Future development

The process of updating Weather API to new .NET Core 2.0 and Entity Framework Core 2.0 has already started. In addition, as soon as EF Core supports spatial data types, they should be implemented to Weather API.

As EnerKey service are sold to other countries weather information should be collected from those countries. Most likely, the first new country to be added is Sweden followed by other Nordic countries after that.

There is also need for weather forecasts, so that the customers and Enegia's internal services can react to rapid weather changes. Cold weather will increase electricity consumption and creates spikes to energy consumption.

Implementing Weather API to other services can be done quickly using OpenAPI specification (OAS). With generated JSON file programs such as NSwagStudio can generate clients instantly.

References

Confluence. 2017. Confluence homepage. Accessed on 5 December 2017. Retrieved from https://www.atlassian.com/software/confluence

EnerKey. 2017. EnerKey Homepage. accessed on 5 December 2017. Retrieved from https://www.enegia.com/en/enerkey/

Design Insights and Tradeoffs in InfluxDB. N.d. Influxdata documentation about Design Insights and Tradeoffs in InfluxDB. Accessed on 25 October 2017. Retrieved from https://docs.influxdata.com/influxdb/v1.3/concepts/insights_tradeoffs/

Gailey, G. Cephas, L and Dykstra, T. 2017. Microsoft documentation article about Webjobs. Accessed on 7 December 2017. Retrieved from https://docs.microsoft.com/en-us/azure/app-service/web-sites-create-web-jobs

InfluxDB Version 1.0 Documentation. N.d. Influxdata documentation about InfluxDB. Accessed on 26 October 2017. Retrieved from https://docs.influxdata.com/influxdb/v1.3/

Information about the free weather data service – yr no. n.d. Norwegian Meteorological Institute article about free data service. Accessed on 25 July 2017. Retrieved from http://om.yr.no/verdata/free-weather-data/

Killeen, S and Stewart D. 2017. XUnit documentation about framework comparison. Accessed on 10 December 2017. Retrieved from https://xunit.github.io/docs/comparisons.html

Lambson, B. 2017. Microsoft documentation article about Entity Framework Core migrations. Accessed on 12 October 2017. Retrieved from https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/

Lander, R ja Wenzel, M. Microsoft documentation article about .NET CORE. Accessed on 20 July 2017. Retrieved from https://docs.microsoft.com/en-us/dotnet/core/index

License. N.d. License agreement on Finnish Meteorological institute homepage. Accessed on 4 December 2017. Retrieved from https://sites.google.com/site/unclebobconsultingllc/getting-a-solid-start

Manifesto for Agile Software Development. 2001. Homepage of Agile Manifesto. Accessed on 10 December 2017. Retrieved from http://agilemanifesto.org

Martin, R. 2009. Article about Design Principles and Design Patterns. Accessed on 10 December 2017. Retrieved from https://pdfs.semanticscholar.org/53d0/8de266fb80355400d10f7ea77eea971d48f9.pdf

Martin, R. 2000. Article about Design Principles and Design Patterns. Accessed on 10 December 2017. Retrieved from https://pdfs.semanticscholar.org/53d0/8de266fb80355400d10f7ea77eea971d48f9.pdf

Miller, R. and Vega, D. 2017. Microsoft documentation article about Entity Framework Core quick overview. Accessed on 12 October 2017. Retrieved from https://docs.microsoft.com/en-us/ef/core/

Miller, R. and Vega, D. 2017. Microsoft documentation wiki about Roadmap. Accessed on 12 October 2017. Roadmap. Retrieved from https://github.com/aspnet/EntityFrameworkCore/wiki/Roadmap

The repository pattern. N.d. Microsoft article about repository pattern. Accessed on 29 November 2017. Retrieved from https://msdn.microsoft.com/en-us/library/ff649690.aspx

OpenAPI Specification. 2017. Swagger.io documentation about OpenAPI Specification. Accessed on 10 December. Retrieved from https://swagger.io/specification/

Rabeler, C. 2017. Microsoft documentation article about What is the Azure SQL Database service? Accessed on 27 July 2017. Retrieved from https://docs.microsoft.com/en-us/azure/sql-database/sql-database-technical-overview

Richardson, C. 2017. Microservices.io article about Microservice architecture. Accessed on 26 October 2015. Retrieved from http://microservices.io/patterns/microservices.html

Schema Design. N.d. Influxdata documentation about Schema design. Accessed on 25 July 2017. Retrieved from https://docs.influxdata.com/influxdb/v1.3/concepts/schema_and_data_layout/

Smith, S and Addie, S. 2016. Microsoft documentation article about Introduction to dependency injection in ASP.NET Core. Accessed on 10 December 2017. Retrieved from https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection

Venners, B.2003. Artima developer article about Orthogonality and the DRY Principle. Accessed on 10 December 2017. Retrieved from http://www.artima.com/intv/dry.html

Vêrvarsel i XML-format [Weather forecast in XML format]. 2017. Accessed on 25 July 2017. Retrieved from http://om.yr.no/verdata/xml/

Wilson, Brad. 2017. XUnit documentation about Getting started with xUnit.net. Accessed on 17 November 2017. Retrieved from https://xunit.github.io/docs/getting-started-dotnet-core.html

Wilson, Brad. 2016. XUnit documentation about xUnit project. Accessed on 10 December 2017. Retrieved from https://xunit.github.io

What is Scrum Methodology. N.d. Article about scrum in VersionOne webpage. Accessed on 5 Dec 2017. Retrieved from https://www.versionone.com/agile-101/

Yritys [Company], N.d. Enegia homepage. Accessed on 19 July 2017. Retrieved from http://www.enegia.com/fi/yritys/

Appendices

Appendix 1. xUnit theory test in Weather API

```
//over year
[InlineData("P2M", "P4M", 2, "2013-12-01")]
[InlineData("P1M", "P2Y", 24, "2013-01-01")]
[InlineData("P1M", "P6M", 6, "2013-10-01")]
[InlineData("P1M", "P6M", 6, "2013-11-01")]
[InlineData("P1M", "P2Y", 24, "2013-01-01")]
[InlineData("P1M", "P15M", 15, "2013-01-01")]
[InlineData("P1D", "P62D", 62, "2013-12-01")]
[InlineData("P1D", "P730D", 730, "2013-01-01")]
//2014
[InlineData("P1M", "P1M", 1, "2014-01-01")]
[InlineData("P1D", "P1D", 1, "2014-01-01")]
[InlineData("P1M", "P3M", 3, "2014-01-01")]
[InlineData("P2M", "P4M", 2, "2013-12-01")]
//2 part
[InlineData("P1M", "P1Y6M", 18)]
[InlineData("P1M", "P1Y3M", 15)]
[InlineData("P1M", "P1Y2M", 14)]
[InlineData("P1M", "P1Y1M", 13)]
[InlineData("P1M", "P1Y9M", 21)]
[InlineData("P1M", "P1Y11M", 23)]
[InlineData("P1M", "P1Y12M", 24)]
#endregion
0 references | Markus Paappanen, 126 days ago | 2 authors, 6 changes | 0 exceptions
public async void Theory(string resolution, string timeframe, int expextedValue, string startDate = "2013-01-01")
{

    var date = DateTime.ParseExact(startDate, "yyyy-MM-dd", CultureInfo.InvariantCulture);
    var request = GetRequest(resolution, timeframe, date);
    await GenericResponses.Payload<Dictionary<string, TimeSerieCollection>>(p =>
    {
        var list = p.Values;
        Assert.True(list.Count > 0, "No values in list");
        //var count = p.Values.First().Values.Values.First().Count;
        foreach (var timeserie in list)
        {
            Assert.True(timeserie.ErrorCodes.Count == 0, "Has error codes in Error container");
            Assert.True(timeserie.Values.Count > 0, "No timeserie values");
            foreach (var collection in timeserie.Values.Values)
            {
                Assert.Equal(expextedValue, collection.Count);
                Assert.Equal(date, collection.First().Timestamp);
                Assert.NotNull(collection.First().WeatherStation);
                Assert.StartsWith("Enegia", collection.First().WeatherStation);
                //Timeline sanity check
                var dateTime = date.AddMilliseconds(-1);
                foreach (var weatheritem in collection)
                {
                    Assert.True(weatheritem.Timestamp > dateTime, "Invalid timeline");
                    dateTime = weatheritem.Timestamp;
                }
            }

        }
        Assert.Equal(2, list.Count);
    }).VerifyAsync(request);
}
```

Appendix 2. XUnit weather tests fixture

```
namespace Enegia.Weather.Tests.Fixtures
{
    5 references | Markus Paappanen, 232 days ago | 2 authors, 2 changes
    public class WeatherApiFixture : IntegrationTestFixture<Api.Startup>
    {
        public const string Geoid = "100000";
        public const string Geoid2 = "100001";
        public const string Postalcode = "40100";
        public readonly bool _useInMemory;

        0 references | Markus Paappanen, 232 days ago | 1 author, 1 change | 0 exceptions
        public WeatherApiFixture()
        {
            var request = this.MakeRequestAsync<AddWeatherDataJsonRequest>(req =>
            {
                req.TimeSeriesData = new List<UpdateWeatherStationTimeSeries>
                {
                    new UpdateWeatherStationTimeSeries
                    {
                        Geoid = Geoid,
                        Location = "60.30373 25.54916",
                        Name = "Enegia JKL sääasema",
                        TemperatureTimeSerie = Get2YearValues(2013)
                    },
                    new UpdateWeatherStationTimeSeries
                    {
                        Geoid = Geoid2,
                        Location = "60.10512 24.97539",
                        Name = "Enegia HKI sääasema",
                        TemperatureTimeSerie = Get2YearValues(2013)
                    }
                };
            });
            ResponseSpec.Ok.VerifyAsync(request).Wait();
        }


        2 references | Markus Paappanen, 232 days ago | 1 author, 1 change | 0 exceptions
        private static Temperaturetimeserie[] Get2YearValues(int year = 2013)
        {
            List<Temperaturetimeserie> tempList = new List<Temperaturetimeserie>();
            int hoursInYear = 24 * 365 * 2; //lazy
            DateTime start = new DateTime(year, 1, 1, 0, 0, 0, DateTimeKind.Utc);
            TimeSpan tHour = new TimeSpan(1, 0, 0);
            decimal tempSeed = -50.0m;
            for (int i = hoursInYear; i > 0; i--)
            {
                tempList.Add(new Temperaturetimeserie()
                {
                    Item1 = start,
                    Item2 = tempSeed.ToString(CultureInfo.InvariantCulture)
                });
                start = start.Add(tHour);
                tempSeed = tempSeed + 0.1m;
                if (tempSeed > 50)
                    tempSeed = -50.0m;

            }
            return tempList.ToArray();
        }

        //public override void Dispose()
        //{
        //    // jos on luotu jotain siivottavaa, siivoa ne täällä.
        //}
    }
}
```