Nikolay Mihaylov

Data Consolidation for Data Sources of Non-uniform Sample Rate

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Thesis

21 April 2017

Helsinki Metropolia University of Applied Sciences

Author(s) Title Number of Pages Date	Nikolay Mihaylov Data Consolidation for Data Sources of Non-uniform Sample Rate 43 pages 21 April 2017
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor(s)	Keijo Länsikunnas, Senior Lecturer

The goal of the thesis was to create a web application which would allow users to configure different sources of time series data with a non-uniform sample rate, and to unify that data into a single output with a constant sample rate. The final result would then be visualized into a chart.

In today's world it is important to provide data in digital form about all aspects of life. It is expected that there is information about everything and at any time. This is often achieved with the help of different sensors that measure certain values. The reality of how time series data is gathered and stored is that it is often impossible to have data taken at fixed time intervals, or in other words the data's sample rate is non-uniform. This raises some problems with visualizing the data, when the users expect it to be taken at regular intervals. People often need the data to be taken at regular intervals, as it makes it easier to understand it, and it could be used more easily for statistics and for noticing certain trends.

This project is a continuation of an Innovation Project, which aimed to create a Web user interface for visualizing time series data into charts. As such, several decisions regarding the used technologies were already made in advance. The project mostly used the Symfony 2.8 PHP framework with some additional technologies that are often used with it: twig PHP template engine for building the HTML5 output of the pages, Doctrine object relational mapper (ORM) for interacting with the relational database management systems (RDBMS). In addition, the project used the jQuery JavaScript library for handling some of the user interface and user interaction. The charts visualizing the time series data, which were built primarily during the Innovation Project phase, used the Data-Driven Documents (D3.js) JavaScript library.

The web application designed for this thesis shows several different techniques for solving the problem of consolidating the non-uniform data into one with a constant sample rate. The web application is aimed to work for different types of time series data, and as such there are several different possible desired results that have been considered. The thesis describes when a certain technique is more suitable, depending on the expected result and on its practical application in the world.

Keywords	data consolidation, Symfony, Doctrine, sampling

Contents

1 Introduction

Computer technologies are an important part of the contemporary world, and to most people it is difficult to imagine living without them. Most types of information are expected to be available in electronic format and to be available for use over the Internet. Certain types of information have common features, which allow them to be handled in the same way. Time series is one of those types of information where it is known that the series consists of data taken at multiple different points in time.

The project done for this thesis is a continuation of my Innovation Project, where the aim was to visualize air quality related time series data into a line chart using a web user interface (UI). During the Innovation Project phase, it was possible to display a line chart of air quality data from a single sensor without taking into consideration the time intervals between the data measurements.

The purpose of this thesis is to find a way to unify time series data taken from different sources, and to consolidate, or, in other words, to combine, the data into a single data set so that it can be visualized to the user in a more useful way. Some of the challenges in the project consist of the fact that the data sources may provide time series data with different time intervals, which means that the data has a non-uniform sample rate. The project aims to visualize the data to the users in such a way that it would appear that the measurements have been taken at regular intervals, or in other words the end result should have a constant sample rate.

The goal of this research is to demonstrate certain methods of time series data consolidation, to identify the challenges in the process, and to reflect on the usefulness of the end result. In addition, this paper aims to explore the different types of mathematical methods that can be used in the process of changing the sample rate of the data, a process also known as resampling.

2 Data consolidation

2.1 Reasons to combine data

Combining data can be understood in numerous different ways. Perhaps the most obvious one is that the data may be available in different places (for example, in different databases), and that it needs to be transferred to a single place, so that it can be processed and understood more easily. Once the data is in a single place, it can be handled in any way that is necessary. However, there are a couple of main outcomes that I have considered when handling time series data.

One possible desired outcome is that the data gathered from different data sources should still be handled as separate data sets. If we think of a scenario where we have three separate sensors that measure the air quality in the Finnish cities Helsinki, Espoo, and Vantaa, we would then want to visualize to the users the air quality in each of those cities separately. The people living in Helsinki, for example, would likely often be interested in only seeing the air quality of their own place of residence. At the same time, if they feel that the air in Helsinki is polluted, they may decide to compare the air quality to that of another city, such as Espoo. In both of these cases, it is important to keep the data sources separate in the end result.

When it is clear that the data sources are to be kept separate in the end result, the only task would be to gather the data from the different data sources into a single place, and to visualize it to the user in such a way that it can be easily comparable. In order to be easily comparable, the time series data would have to be formatted in such a way that it would be taken at regular intervals. For example, users are often interested in hourly, daily, and monthly values. In order to do that, it is often necessary to resample the data using mathematical methods.

Another possible outcome of combining the data is if we want to make a conclusion about a trend based on multiple data sources. For example, using our three theoretical air quality sensors mentioned earlier, placed in the cities of Helsinki, Espoo, and Vantaa, (cities, which are located close to each other), we can make some conclusions about the air quality in the Helsinki region in general. In this case, we would treat the data gathered from the three data sources as a single data set, and then it would be possible to apply the relevant resampling methods on the combined set.

2.2 Resampling of data

When dealing with time series data in digital form, which is based on sensor measurements, we are essentially dealing with a discrete-time signal. As Lyons (2004) states in his book, when we know the value of signal only at specific points in time, we use the term discrete-time signal to describe it. The science of analyzing the physical processes dealing with such signals is called digital signal processing. Usually we obtain the values for the discrete instances in time by *sampling* an analog (continuous) signal within regular time intervals, thus turning it into a digital signal. [1,2-4.]

There are numerous practical use cases where the sample rate of a discrete signal needs to be changed - a process known as sample rate conversion or resampling. For example, in satellite image processing resampling is needed in scale change and image enhancement [1,381]. Resampling is also often needed in digital video processing (e.g. downconversion from High-definition (HD) to Standard-definition (SD)) and digital audio signal processing (e.g. changing sample rate from 48 kHz to 44.1kHz in one-dimensional resampling) [2,221]. One possible way to achieve sample rate conversion is to convert a digital signal into an analog signal, and then back into digital but using a new sample rate. However, this is non-optimal as it introduces spectral distortions, and all-digital resampling should be used instead. [1,381.]

When resampling a discrete signal, it may be necessary to increase or decrease the sample rate. In order to increase the rate, it is necessary to estimate intermediate sample values. This process is known as upsampling or interpolation (described further in section 2.2.1). If the discrete-time signal has a constant sample rate, we could decrease the sample rate by retaining every Nth sample and discarding the remaining ones. This process is known as decimation or downsampling, and it is described further in section 2.2.2. [1,382.] Resampling produces new samples that assume the input samples are related to each other by a continuous function. If that is not the case, then problems such as artifacts (or signal processing errors) can be expected [2,222]. If we are dealing with computer-generated random data, this may be problematic.

2.2.1 Upsampling

The main concept about upsampling of a discrete signal is that the produced result samples are more than the input samples [2,221]. When using interpolation, we need

to calculate new sample values. [1,387.] An example of this is shown in Figure 1. There are many available methods to apply interpolation. The so-called *Lagrange interpolation* uses polynomials to compute the new sample values. Often used in digital processing are the *linear interpolation* and *cubic interpolation*, both of which are cases of *Lagrange interpolation*. *Quadratic interpolation* (second-degree Lagrange interpolation) is rarely used in signals processing. [2,227.] Equation (1) shows the formula of the linear interpolation function interpolation, if we take two sample points [x_0, s_0] and [x_1, s_1] [2,226].

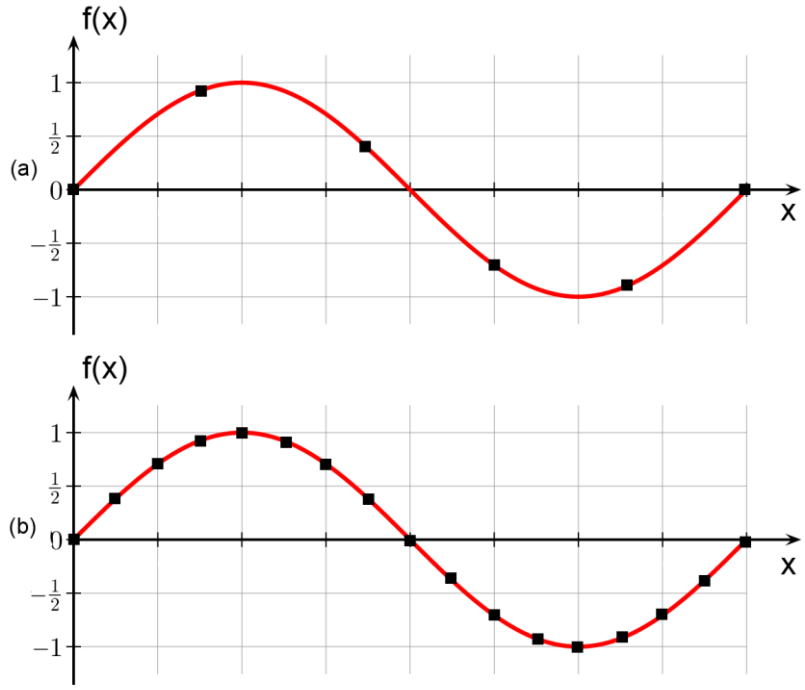

Figure 1. Discrete-time signal interpolation: (a) original sample rate; (b) increased sample rate by computing new sample values.

$$\tilde{g}(x) = S_0 + \frac{x - x_0}{x_1 - x_0}(S_1 - S_0) \tag{1}$$

In practice in signal processing a sinc weighing function is often used. That function is not a polynomial and does not come from classical mathematics. [2,230-231.] The sinc function derives its name from the *sine cardinal* function [3]. Using this function addresses certain problems in polynomial interpolation functions. One example issue that may arise when using polynomials is that the value may go to plus or minus infinity

when calculating a sample value outside the region where the original sample values were placed. [2,230.]

As I have no previous background in digital signals processing, it is beyond my understanding how to implement most interpolation algorithms in practice. For this reason, in this project I decided to use the simplest, to me personally, type of interpolation: *linear interpolation*. The practical implementation of this is described further in chapter 6.

2.2.2 Downsampling

Downsampling is the process of reducing the sampling rate of a signal. If we decimate a discrete-time signal by a factor of 3, that means we need to retain a sample, then discard the following two samples, and then repeat this same process again for the next samples [1,382]. This is illustrated in Figure 2.

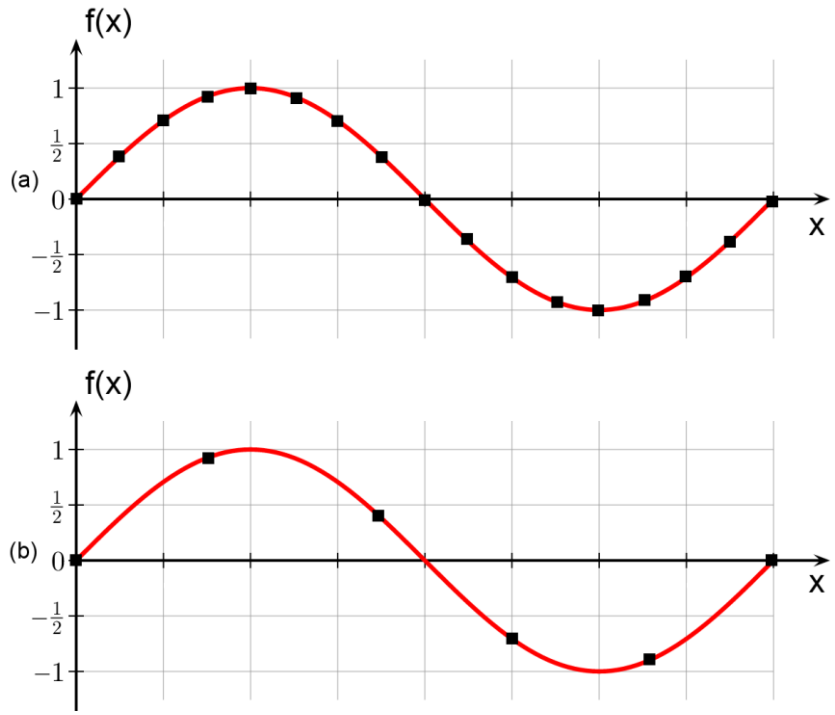

Figure 2. Discrete-time signal decimation: (a) original sample rate; (b) reduced sample rate by a factor of 3.

As becomes apparent in Figure 2, we simply get rid of certain samples in the discrete-time signal. This looks good when the underlying function is clearly defined and follows a specific pattern (in this case, this is a sine function). However, when dealing with arbitrary values, as was planned in this project, this might not be the optimal solution. I

decided that it would be better if the users are instead able to see the average measurement values for different intervals of times (e.g. by minute, by hour, by day, or by month). My own solution to downsampling is further described in chapter 6.

When using downsampling, there is an important concept to note: Nyquist rate. This is a rate equal to half the sampling rate. In order to guarantee signal sampling without aliasing, all of the frequency components of the signal must be contained within the Nyquist rate. A signal that conforms to this standard is said to satisfy the Nyquist criterion. [2,193.] This is yet another complication to the traditional decimation problem, and another reason why I chose to implement downsampling in a different way.

3 Data-Driven Documents Library

The web user interface (UI), which was built for visualizing air quality time series data into a line chart during my Innovation Project, uses the Data-Driven Documents (D3.js) JavaScript library. As the thesis is a continuation of the Innovation Project, and as it was necessary to make some changes to the line charts' code, it is relevant to describe the basics of how this JavaScript library was used in the project.

The D3.js library provides the necessary tools for drawing many types of charts. However, since the library is designed to be as flexible as possible in order to satisfy many use-cases, the code necessary for the desired type of chart can become very long and confusing. For these reasons, I decided to define higher-level JavaScript classes to make it easier to separate the functionality and logic. As an example, with D3.js there are concepts like "chart axes" and "nodes" (data points), whereas in my custom classes I define concepts like "line chart", "filters" (controlling what is visualized in the chart), and "legend" (explanation of what each value displayed in the chart means).

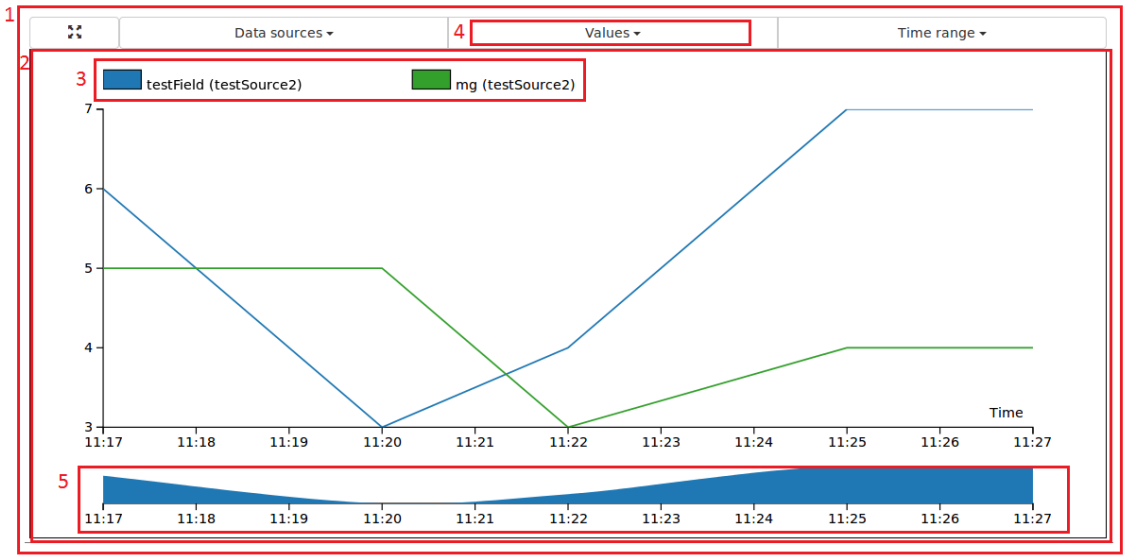

Figure 3. A measurement diagram showing different elements defined as separate JavaScript classes.

Figure 3 visually demonstrates the use of some custom classes that I have defined. They are as follows: MeasurementDiagram (numbered *1*), LineChart (numbered *2*), Legend (numbered *3*), Filters (numbered *4*), and LineChartContext (numbered *5*).

3.1 Scalable Vector Graphics and HTML Canvas Element

During the Innovation Project phase, I decided that I was going to use the D3.js library to generate a Scalable Vector Graphics (SVG) output, which is essentially XML-based code used for visualizing the line charts. The alternative was to use a HyperText Markup Language 5 (HTML5) canvas element, which, like SVG, is used for drawing graphics on a web page. The difference between the two is that in SVG, every shape that is drawn is a separate XML element and it is straightforward to register JavaScript event listeners (actions that occur when interacting with the element) for the separate shapes. In practice that means that when using SVG, it is easy to identify when the mouse, or cursor, is placed over a specific shape or object. When using the HTML5 canvas element, the drawn shapes are practically just pixels with different colors, and it would be necessary to store into memory or to recalculate the coordinates of every drawn shape, in order to identify on which object the user has placed the mouse.

Another reason why I chose SVG over the HTML5 canvas element was that the line charts needed to be more scalable. The charts were to be displayed correctly both in a desktop web browser, and in a mobile web browser. As the screen sizes and resolution

of the desktop and mobile devices can differ significantly, I decided that it was better to use SVG, since the drawn shapes are vector-based. When using vector graphic, it will look the same even if the user zooms into it.

The drawback of using SVG is that, as each shape in the chart is a separate XML element, it could become heavy on the user's web browser to visualize all the elements in case there are too many of them. Even more so, if every single SVG shape has a JavaScript event listener attached to it, the user's web browser may get slowed down further. In practice, this limitation means that a line chart for a time series data should not be visualizing too many data points at once. For example, if the user wants to observe the trend of the values over the past year, it may be worth providing a single measurement for each of the respective months. On the other hand, if the user wants to observe the trend of the values over the past day, then it is possible to show data points that have, for example, an hour-long time interval between them.

3.2 Multiple Diagrams with Multiple Data Sources

During the Innovation Project phase, the line charts created with D3.js were only able to display data measurement values from a single data source. The data source was also known as "sensor" as the project dealt with air quality measurement data. As part of the thesis phase, it was necessary to expand the functionality of the D3.js line chart so that users would be able to see different charts on the same website, as well as to compare data from different data sources in the same chart.

In order to provide data from multiple data sources, there was a need to create a new backend Application Programming Interface (API), which serves the time series data to the frontend (the user's web browser). The user's browser makes one or multiple Asynchronous JavaScript and XML (AJAX) requests to the backend API and it receives the time series data in JavaScript Object Notation (JSON) format. The process flow of this is illustrated in Figure 4.

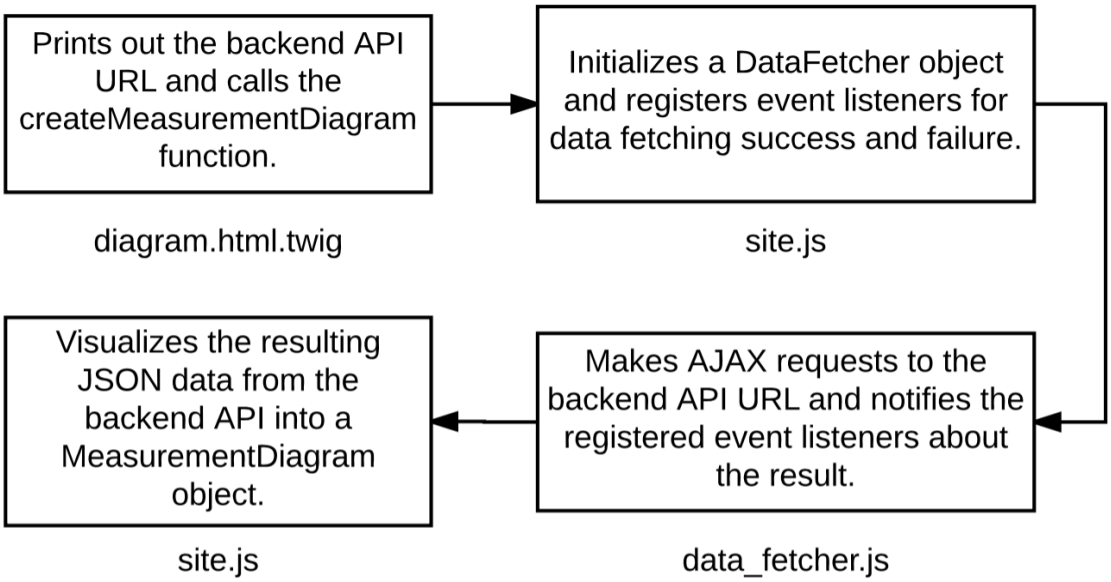

Figure 4. AJAX requests' workflow used for fetching JSON data from the backend API and visualizing it into a diagram.

The code in Listing 1 shows the content of web/diagram/js/site.js in relation to requesting the diagram data from the backend API.

```
// Create a measurement data fetcher.
var measurementEntriesLimit = 50;
dataFetcher = new DataFetcher(backendUrl, measurementEntriesLimit);

// Create the measurement diagram.
// // The object is needed at this stage for some of the listener
functions.
var measurementDiagram = new MeasurementDiagram(dataSources, valueTypes,
chartData);

// Register listeners to react to the data updating.
// Successful data fetch listener.
var onDataFetchSuccess = function () {
    try {
        // Get the last retrieved data.
        chartData = dataFetcher.getData();
        // Set the data to the measurement diagram.
        measurementDiagram.setData(chartData);
    }
    catch (err) {
        console.log("Failed to apply the fetched measurement data to the
diagram.");
        console.log(err);
    }
};
// Register the event listener function to be executed on data fetch
success.
dataFetcher.addDataUpdateListener(true, onDataFetchSuccess);

...

// Unsuccessful data fetch listener.
var onDataFetchFailure = function (err) {
    console.log("Failed to fetch measurement data.");
    console.log(err);
};
// Register the event listener function to be executed on data fetch
failure.
dataFetcher.addDataUpdateListener(false, onDataFetchFailure);

// Fetch the initial data and schedule continuous updating.
dataFetcher.init(updateIntervalMilliseconds);
```

Listing 1. The createMeasurementDiagram function in web/diagram/js/site.js. The function uses additional custom-defined JavaScript classes to periodically make AJAX requests to the backend API, and to react to data fetching success and failure.

Listing 1 shows code from the *createMeasurementDiagram* function. This function is used in the *diagram.html.twig* template, once the Twig template has “rendered” some JavaScript variable values to be used. The contents of the Twig template are shown in Listing 2 and an example HTML output of the same section in code is shown in Listing 3.

```
{% block javascripts %}
    {{ parent() }}
...
    <script>
        // jQuery document ready.
        $(function() {
            var dataSources = {{ data_sources|json_encode|raw }};
            var valueTypes = {{ value_types|json_encode|raw }};
            var backendUrl = '{{ backend_url }}';

            createMeasurementDiagram(dataSources, valueTypes,
backendUrl);
        });
    </script>
{% endblock %}
```

Listing 2. The twig template diagram.html.twig “rendering” Twig variables as values to JavaScript variables, and then creating a measurement diagram.

```
<script>
        // jQuery document ready.
        $(function() {
            var dataSources = ["testSource2","source 3"];
            var valueTypes = ["testField","mg"];
            var backendUrl = '/data-fetcher/get-
content/DefaultEntityManager/diagram1';

            createMeasurementDiagram(dataSources, valueTypes,
backendUrl);
        });
</script>
```

Listing 3. HTML output of an example diagram page, showing seemingly hard-coded JavaScript variable values, which are in fact generated by Twig.

The Twig variables used in Listing 2 are passed directly from a Symfony Controller to the Twig template. This is demonstrated in Listing 4.

```
/**
* Visualizes a diagram to the user.
*
* @param string $sanitized_entity_manager_name The sanitized entity
manager name associated with the doctrine entity.
* @param string $entity_name The name of the doctrine entity.
*/
public function showDiagramAction($sanitized_entity_manager_name,
$entity_name)
{

...

    // Get all available data sources in this diagram.
    $diagramDataSources = $doctrineEntityHelper-
>getDiagramDataSources($em, $fullyQualifiedClassName);

    // Get all available value types in this diagram (diagram
properties).
    $diagramProperties = $doctrineEntityHelper-
>getDiagramProperties($fullyQualifiedClassName);

    return $this-
>render('DataConsolidationDiagramDataFetcherBundle:Default:diagram.html.t
wig', array(
            'diagram_name' => $entity_name,
            'data_sources' => $diagramDataSources,
            'value_types' => $diagramProperties,
            'backend_url' => $this-
>generateUrl('data_consolidation.diagram_data_fetcher.get_content',
array(
             'sanitized_entity_manager_name' =>
$sanitized_entity_manager_name,
            'entity_name' => $entity_name,
        )),
    ));
}
```

Listing 4. Symfony Controller's action for showing a diagram. The code demonstrates how relevant values are passed to the diagram.html.twig template.

During the thesis phase of the project, there was also a need to display multiple diagrams on the same website. As the project's scope was no longer limited to air quality data, but rather to time series data consolidation and visualization, there could be a need to display multiple diagrams per website. For example one diagram might deal with the air quality in the Helsinki region, whereas another diagram might deal with the temperatures in different regions in Finland. Thanks to the Twig template engine for PHP (described in section 4.2), it is easy to generate charts that use a different Uniform Resource Locator (URL) in order to access the backend API in the correct way, so that they can get the necessary data. This was demonstrated in Listing 2 in combination with Listing 3**.**

4 Symfony PHP Framework

During the Innovation Project phase of this project, there was a need to create a web UI containing a line chart of air quality measurements' data. While the initial phase focused on the chart visualization from a single sensor, the project was meant to be expanded to have further functionality. The idea was that there would also be registered users with different roles and permissions. Authenticated users would be able to see the line charts to which they had viewing permissions. They would also be able to see a summary of the air quality: would it be good, moderately good, or bad. Administrator users would be able to define the limits controlling the air quality status summary. During the thesis phase some additional features benefited from different user roles. Defining different data sources and diagrams is a feature that is only intended for administrator users, for example.

By using a software framework, a lot of functionality is achieved in a faster, or sometimes easier, way. The idea is that there are more readily available solutions for problems faced often in software development. Some of the challenges in using a framework are that it takes time to learn it and knowing the programming language, in which it is written, is often not enough. Additionally, when using a software framework, there are certain restrictions in the way the software should be designed. In some cases it is impossible to achieve certain functionality if that use-case was not taken into consideration when designing the framework.

I chose the Symfony PHP framework back when I did the Innovation Project. My reasoning for choosing this specific framework was that it was popular in the web development world. The Symfony PHP framework can be used as a whole, or alternatively it is possible to use just separate decoupled software libraries, known as Symfony Components [4]. The open source content management platform Drupal 8 uses some Symfony Components. As such, I decided that the Symfony PHP framework was something worth learning and exploring, as I believed it to be something that would be useful for my work life. This project has been a good way to get some experience with the said framework. In addition, by following the commonly established guidelines when using the software framework, the code is likely better structured than it would have been if I had written everything from scratch.

4.1 Symfony Bundles

A bundle in Symfony PHP framework is a structured set of files that implement a single feature. Everything in Symfony is a bundle, including the core framework functionality. [5.] The functionality that I wrote for this thesis's project is also structured in multiple bundles. In addition to the bundles that come with the core framework, it is possible to use also community-contributed bundles. This can be achieved by downloading them using Composer, a tool for dependency management in PHP. Using Composer, it is easy to install and update software libraries (in this case, Symfony bundles) that are required for the software project. [6.]

The community-contributed bundles that I am using in the project are the following:

- SymfonyContrib\Bundle\ConfirmBundle\ConfirmBundle – used for displaying action confirmations to the user, for example when deleting certain configuration data.
- Craue\FormFlowBundle\CraueFormFlowBundle – used for creating multi-step forms required for certain administrative configurations in the project.
- FOS\UserBundle\FOSUserBundle – provides a ready-made solution for registering and authenticating website users. [7.]

4.2 Twig templating

In most situations, web pages are not very useful when their content is static (when it never changes). That means that if the content of the pages is to change, the code sent to the user's browser needs to be generated again. Whether for generating HTML, Cascading Style Sheets (CSS), Extensible Markup Language (XML), or some other type of text-based output, it is often useful to use a templating engine in the output generating process. [8.]

The PHP language is able to generate text-based output. This is often done using a PHP template, which is essentially a file parsed by PHP, which contains a mix of text and PHP code. There are certain technologies that are designed specifically for the purpose of templating and are thus more powerful in these types of tasks. One of them is the templating language, called Twig. [8.]

Twig templates are compiled down to optimized PHP code, which means the overhead, compared to using basic PHP templates, is small. [9.] The generated PHP code is also cached [8], meaning there is no need for recompiling it unless the actual Twig template's content change. Twig also has a lot of functionality that is not straightforward to achieve in PHP templates. Some of them include automatic HTML escaping, and the inclusion of custom functions only relevant to templates. [8.]

There are several main features that make Twig templating especially useful to me: easy variable outputting, conditional tags, block tags, inheritance, and macros. Let us take a look at a few Twig templates in order to demonstrate each of those features in practice.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8" />
        {% block metatags %}{% endblock %}
        <title>
            {% block title %}
                {# Include a custom page title if it was set #}
                {% if page_title is defined %}
                    {{ page_title }}
                {% else %}
                    Welcome!
                {% endif %}
            {% endblock %}
        </title>
        {% block stylesheets %}{% endblock %}
        <link rel="icon" type="image/x-icon" href="{{
asset('favicon.ico') }}" />
    </head>
    <body>
        {% block body %}{% endblock %}
        {% block javascripts %}{% endblock %}
    </body>
</html>
```

Listing 5. The base Twig template (located at app/Resources/views/base.html.twig), acting as an inheritance parent for most other Twig templates in the project.

Listing 5 illustrates the base Twig template in the project. It defines the HTML5 structure and outputs certain variables to the page. In case a *page_title* variable is defined, its content will be displayed as the title of the HTML page. Otherwise only a static "Welcome!" title is displayed. The code in Listing 6 demonstrates how to use the *if* (conditional) Twig tag, and how to output variables:

```
{% if page_title is defined %}
    {{ page_title }}
{% else %}
    Welcome!
{% endif %}
```

Listing 6. Outputting a variable in a Twig template.

If we take a further look at Listing 5, we see that there are a lot of *block* Twig tags. The blocks are useful when using template inheritance [10]. They define specific places in the text-output that can be replaced by child templates, inheriting from the current one. Additionally, it is possible to provide a default value for the block, so there is no need to replace it if it is not relevant. For example, in the case shown in Listing 7, we provide default content to be placed within the “title” Twig block:

```
<title>
    {% block title %}
        {# Include a custom page title if it was set #}
        {% if page_title is defined %}
            {{ page_title }}
...
        {% endif %}
    {% endblock %}
</title>
```

Listing 7. Twig template block with some default content.

Twig template inheritance is one of the features that I have used extensively in the project. While Twig does not support multiple inheritance, in the sense that a single template cannot have two parents at the same time [11], it is still possible to define a more complicated structure with multiple levels of inheritance. For example, in this project the template *app/Resources/views/default/bootstrap_base.html.twig* extends (inherits from) *app/Resources/views/base.html.twig.* Then the template *app/Resources/views/default/layout.html.twig* extends the *bootstrap_base.html.twig* template, which means it also indirectly extends the *base.html.twig* template. This is illustrated in Figure 5. The Twig code for extending a template is simple, and it is shown in Listing 8.

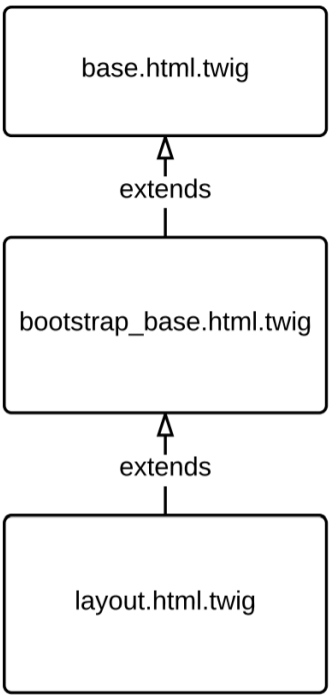

Figure 5. Twig template inheritance used in the project.

```
{% extends 'base.html.twig' %}
```

Listing 8. An *extends* tag used to mark the inheritance from a parent Twig template.

Twig macros are similar to PHP functions, except the arguments of a macro are always optional, even if no default value is provided for them. [12.] This is similar to how JavaScript functions work. A Twig macro does not have access to the variables in the current template, but rather only to the arguments passed to the macro. Macros are useful for creating reusable elements, so that there is less need for code repetition. [12.] Listing 9 shows an example macro that I have defined in the app/Resources/views/default/*configuration_menu.html.twig* template, used for displaying an HTML list item.

```
{# Define a macro for printing a single regular (non-URL) menu item #}
{% macro list_regular_item(value, glyphicon_class) %}
    {% if glyphicon_class|default %}
        {# Add the glyphicon before the value. #}
        <li><span class="glyphicon {{ glyphicon_class }}"></span> {{
value }}</li>
    {% else %}
        {# No glyphicon to be displayed. #}
        <li>{{ value }}</li>
    {% endif %}
{% endmacro %}
```

Listing 9. A macro that displays a list item in a different way, depending on whether a glyphicon_class argument has been passed to it.

The *list_regular_item* macro shown in Listing 9 is then used within another macro in the same template. This is demonstrated in Listing 10.

```
{% macro list_items(menu_items) %}
    {# Import the other macros defined in this template as well as the
current macro itself (for recursive calls). #}
    {% import _self as menu_utils %}

...

    {{ menu_utils.list_regular_item(menu_item.name,
menu_item.glyphicon|default) }}

...

{% endmacro %}
```

Listing 10. Importing a Twig macro defined in the same template and then using it.

4.3 Doctrine ORM

The Symfony Framework does not force the use of a specific component to work with databases. However, the framework provides integration with a third-party library called Doctrine, whose purpose is to ease interaction with databases. [13.] A part of Doctrine is a Database Abstraction Layer (DBAL), which provides a single way to interact with many popular relational databases. Additionally, Doctrine provides an object-relational mapper (ORM) which is built on top of the Doctrine DBAL. [14.] While it is possible to use only the DBAL without the ORM, for this project I decided that it is better to use the ORM. This decision made a big impact on the technical solution to many problems. When using a DBAL or an ORM, the code is not bound to a single type of relational database management system (RDBMS), such as MySQL. This was the primary reason why I decided to use these additional layers of abstraction that Doctrine provides.

While there are some alternatives to the use of Doctrine in Symfony, they are no longer described in detail in The Symfony Book, at least in version 2.8 and later. As stated in the book, the Propel ORM toolkit used to be one very popular alternative to Doctrine but its popularity has rapidly declined. [15,113.] In addition to the popularity factor, I decided to choose Doctrine over Symfony also because I found it easier to dynamically generate the classes to be used in the Object-relational mapper (ORM), which Doctrine provides. This is described in more detail in section 4.3.3.

As Doctrine is an object-relational mapper (ORM), it allows the developers to use PHP objects when dealing with data that should be persisted to the database. In Doctrine, a PHP class used for holding data is often called an *entity*. The class needs to have mapping information that specifies which class properties go into which database table and column. One of the formats to provide that mapping information is by using annotations written as comments directly in the PHP class. [13.]

```
/**
 * DataSourceToDiagramMapping
 *
 * @ORM\Table()
 *
@ORM\Entity(repositoryClass="DataConsolidation\CustomNodesBundle\Entity\D
efaultEntityManager\Base\DataSourceToDiagramMappingRepository")
 * @UniqueEntity(
 *     fields={"dataSource", "diagram"},
 *     message="This data source + diagram combination already exists."
 * )
 */
class DataSourceToDiagramMapping
{
    /**
     * @var integer
     *
     * @ORM\Column(name="id", type="integer")
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;

    /**
     * @var string
     *
     * @ORM\Column(name="dataSource", type="string", length=500)
     */
    private $dataSource;

    /**
     * @var string
     *
     * @ORM\Column(name="diagram", type="string", length=500)
     */
    private $diagram;

    /**
     * @var ArrayCollection
     *
     * @ORM\OneToMany(targetEntity="FieldMapping",
mappedBy="dataSourceToDiagramMapping", cascade={"persist", "remove"},
orphanRemoval=true)
     * @Assert\Valid()
     */
    protected $fieldMappings;

}
```

Listing 11. Doctrine Entity definition with annotations taken from DataSourceToDiagramMapping.php

Listing 11 shows a Doctrine entity class called DataSourceToDiagramMapping. With the help of PHP annotations in the comments, it is defined that the entity will get its own database table with a name which is the same as the class name. This is shown in Listing 12.

```
* @ORM\Table()
```

Listing 12. A Doctrine annotation stating that the Doctrine entity class name should be used as the database table name.

Next, in Listing 13, it is defined that the combination of the properties *dataSource* and *diagram* cannot be duplicated in the database entries.

```
* @UniqueEntity(
*     fields={"dataSource", "diagram"},
*     message="This data source + diagram combination already exists."
* )
```

Listing 13. A Doctrine annotation defining a unique database constraint for multiple fields.

The class properties are then mapped to specific database columns. The annotations then also specify the type of the database column, for example *integer* or *string*. In the case of the *fieldMappings* property, the annotations define a one-to-many database relationship, as shown in Listing 14.

```
* @ORM\OneToMany(targetEntity="FieldMapping",
mappedBy="dataSourceToDiagramMapping", cascade={"persist", "remove"},
orphanRemoval=true)
...
*/
protected $fieldMappings;
```

Listing 14. A Doctrine annotation defining a one-to-many database relationship.

In practice this means that a *DataSourceToDiagramMapping* object will be linked to one or more *FieldMappings* objects. Furthermore, a *FieldMappings* object may be linked only to one *DataSourceToDiagramMapping* object. This is illustrated in Figure 6.

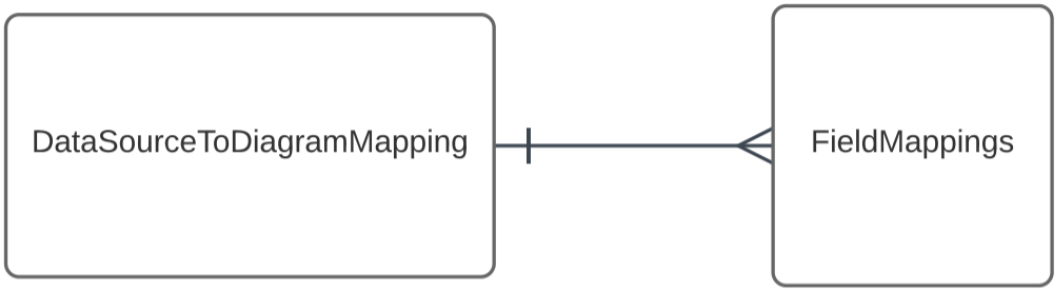

Figure 6. A one-to-many relationship between two Doctrine entity classes, which is reflected in the database entries through the use of Doctrine ORM.

The benefits of using objects for holding the data is that once the classes and the database mappings are defined, it is no longer necessary to worry about how the data is persisted to the database. It is easy to use the Doctrine entity objects the same way as any PHP object, so it feels natural to process the data, especially given that Symfony is

highly object-oriented. Furthermore, the Doctrine entities are easily bound to Symfony forms, which makes it more straightforward to expose the data to the end user for editing using a web UI. Listing 15 demonstrates how a Doctrine entity is bound to a form and how it is persisted to the database. The comments in the listing explain the important points regarding the persistence and form handling.

```
/**
 * Handles creating a new custom node.
 */
public function addAction(Request $request)
{
    // Create a new NodeConfig Doctrine entity object.
    $nodeConfig = new NodeConfig();
    // Create a form that is specific to this Doctrine entity class.
    $form = $this->createForm(NodeConfigFormType::class, $nodeConfig,
array(
        'label' => false,
    ));

    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        // Valid form submission.
        // Persist the node config, with all possible associated node
config fields, to the database.
        $em = $this->getDoctrine()->getManager();
        // Make the $nodeConfig object MANAGED by the entity manager.
This does not persist the object until the entity manager's flush method
is invoked.
        $em->persist($nodeConfig);
        // Synchronize all MANAGED objects to the database in the most
efficient way possible.
        $em->flush();
...
    }

    // Display the form for adding a NodeConfig entity object.
    return $this-
>render('DataConsolidationCustomNodesBundle:NodeConfig:add.html.twig',
array(
        'form' => $form->createView(),
    ));
}
```

Listing 15. A DataSourceController.php action for displaying a web form and then persisting a newly added NodeConfig Doctrine entity object to the database.

Using Doctrine has, however, also brought some complications to this project. Since I aimed for the project to be usable with all types of time series data that the administrator users can configure, this raised the problem of how the Doctrine entity classes for interacting with the databases could be defined. The solution to this took me a large amount of time and it made a big impact on what the final code looks like. At times, I was forced into a certain ways of designing things because of my choice of using Doctrine. Of course, that would likely be the case with almost any types of chosen technologies, but in this case there were certain issues that became apparent: the need to

generate Doctrine entities (explained further in section 4.3.3), the way the files had to be organized in the project (more information in section 4.3.1), and the potentially high use of computing resources, due to the use of objects (see chapter 7).

4.3.1 Configurable Databases

The first prerequisite for using Doctrine is to provide database configuration information. This is usually done by modifying the *app/config/config.yml* and the *app/config/paremeters.yml* files. Listing 16 shows the configuration for Doctrine DBAL and ORM, specified in the *app/config/config.yml* file used in this project.

```
# Doctrine Configuration
doctrine:
dbal:
    default_connection: default
    connections:
       default:
          driver:   "%database_driver%"
          host:     "%database_host%"
          port:     "%database_port%"
          dbname:   "%database_name%"
          user:     "%database_user%"
          password: "%database_password%"
          charset:  UTF8
orm:
    auto_generate_proxy_classes: "%kernel.debug%"
    entity_managers:
        default:
            connection: default
            auto_mapping: true
            mappings:
                DataConsolidationCustomNodesBundle:
                    prefix:
DataConsolidation\CustomNodesBundle\Entity\DefaultEntityManager
```

Listing 16. Doctrine database abstraction layer (DBAL) and object-relational mapper (ORM) configuration, specified in the project's default configuration at app/config/config.yml.

The YAML code in Listing 16 configures only a single database connection, named *default*. The parameters for accessing the database, such as host, user, and password, are provided as variables in the *app/config/parameters.yml* file. The default database is where all the consolidated data goes once it is processed. This *default* database is also needed for storing certain configurations related to the data mapping and generation of custom Doctrine entities. The ORM section of the configuration in Listing 16 makes it so that all Doctrine entities with PHP class namespace prefix *DataConsolidation\CustomNodesBundle\Entity\DefaultEntityManager* will automatically use the *default*

database connection. The namespace prefix cannot conflict with other defined mapping prefixes, as that would result in some Doctrine entities not being able to be found [16].

Symfony uses a class autoloader, whose functionality is essentially to automatically include the relevant PHP files containing the relevant PHP classes needed for that process or session. This is PHP's way of automatically loading and parsing the relevant files, based on the current need, and trying to reduce the needed computing resources. Symfony's autoloader requires a PHP file to be placed in a specific path in the file system, based on the PHP class namespace. This, combined with the fact that Doctrine ORM configurations cannot have conflicting prefixes, meant that I needed to come up with a specific file system location where the Doctrine entities would be placed. More information on that is provided in sections 4.3.2 and 4.3.3.

As the project requires operations with multiple databases, it was necessary to be able to easily add multiple database connections. To make this task simpler for the administrator users, I decided that creating a database connection had to be doable through the web UI. In order to handle all functionality related to that, I created a Symfony bundle named *DatabaseConfigurationBundle*. In this bundle I defined a custom class called *DatabaseConfigurator*, which is registered as a Symfony service. When a class is registered as a Symfony service, it means that it is easily reusable within other contexts (often times within Symfony controllers – a component that is a part of the Model-View-Controller (MVC) web design pattern). The *DatabaseConfigurator* class has methods for writing and reading database configurations to YAML files.

Considering that Symfony bundles are supposed to be decoupled (as independent from each other as possible), I decided that it was not a good idea that my *DatabaseConfigurationBundle* would modify the whole project's base configuration, located at *app/config/config.yml*. Instead, I had to define a custom configuration that was to be stored in the *DatabaseConfigurationBundle*'s own directory. Doing this was not straightforward, and in order to explain my solution, I need to clarify the concepts of Symfony's service container and compiler passes.

As briefly mentioned earlier in this section, Symfony uses objects known as *services* for doing specific tasks. For example, there could be a *Mailer* service object whose purpose is to send email messages. Whenever new bundles are installed to the project, chances are there will be additional *services* available for use. These *services* reside

within a special object, named *service container*. Once there is access to the *service container* object, it is possible for the developer to fetch any *service* that is currently available in the project. Symfony's Controller classes get access to the *service container* automatically. [17.]

Symfony's service container is compiled into an optimized DependencyInjectionComponent. A compiler pass allows the developer to interact with the definitions of Symfony's services, their configurations, and configuration parameters, once they have been loaded but before they have been compiled. [18.] Using compiler passes allows developers to, for example, prepend the settings for any bundle as if they have been written directly in the application's default configuration, located at *app/config/config.yml*. This is precisely what I used in order to apply changes to Doctrine's configuration without modifying the application's configuration file. As the changes are applied during the service container compilation, it is essential to clear the application's cache whenever the database connection configuration data is modified so that a recompilation can take place.

Listing 17 shows my solution to how to modify Doctrine's configuration at the compile time of the service container. The comments in the listing describe what is happening.

```
class DataConsolidationDatabaseConfigurationExtension extends Extension
implements PrependExtensionInterface
{
    ...

    /**
     * Allow an extension to prepend the extension configurations.
     *
     * @param ContainerBuilder $container
     */
    public function prepend(ContainerBuilder $container)
    {
        // Obtain Doctrine's configuration.
        $configs = $container->getExtensionConfig('doctrine');
        if (!isset($configs['0'])) {
            // Seems like the extension config for doctrine could not be
fetched.
            return;
        }

        // Select the first element of the doctrine configs.
        $config = reset($configs);

        // Get a DatabaseConfigurator instance directly.
        // The DatabaseConfigurator service cannot be obtained through
the container
        // because it is not fully compiled yet.
        $databaseConfigurator = new DatabaseConfigurator();

        // Read the custom parameters config defined in the
DatabaseConfigurationBundle.
        $parametersConfig = $databaseConfigurator-
>getDatabaseParametersValue();
        if (!isset($parametersConfig['parameters'])) {
            // Invalid parameters config. Do not do anything else.
            return;
        }

        // Register parameter values for the container.
        foreach ($parametersConfig['parameters'] as $parameterName =>
$value) {
            $container->setParameter($parameterName, $value);
        }

        // Read the custom database config defined in the
DatabaseConfigurationBundle.
        $databaseConfig = $databaseConfigurator-
>getDatabaseConfigValue();
        if (!isset($databaseConfig['doctrine'])) {
            // Invalid database config. Do not do anything else.
            return;
        }

        // Merge the existing doctrine config with the custom one defined
in this bundle.
        $config = array_merge_recursive($config,
$databaseConfig['doctrine']);

        // Prepend the extension config, effectively modifying doctrine's
config.
        $container->prependExtensionConfig('doctrine', $config);
    }
}
```

Listing 17. Use of Symfony's service container prepend compiler pass in DataConsolidationDatabaseConfigurationExtension.php.

Using the solution from Listing 17, I am able to use separate configuration files that are stored in *DatabaseConfigurationBundle/Resources/config/*. The relevant files in this directory are named *database_config.yml* and *database_parameters.yml*, and they are read and written to by the *DatabaseConfigurator* service. In order to provide access to the administrator users to create, modify, and delete database configuration data using the web UI, it is necessary to grant read and write permissions of the *database_config.yml* and *database_parameters.yml* files to the Linux user (if using Linux) that runs the web server, such as *nginx*.

4.3.2 Pre-defined Doctrine Entities

There are some pre-defined Doctrine entity classes that I wrote for the purpose of this project. Those entity classes are located in the *CustomNodesBundle/Entity/DefaultEntityManager/Base/* directory. As described briefly in section 4.3.1, the Doctrine ORM configuration requires that entities' namespace prefixes do not conflict with each other. In this case, I have defined in the application's *config.yml* file that the entities located under this directory are managed by the *default* Doctrine entity manager, meaning that they use the *default* database connection, as shown in Listing 18.

```
mappings:
                    DataConsolidationCustomNodesBundle:
                        prefix:
DataConsolidation\CustomNodesBundle\Entity\DefaultEntityManager
```

Listing 18. Doctrine ORM configuration for auto-mapping of entity classes to the default entity manager.

The specified *prefix* in Listing 18 translates into the *CustomNodesBundle/Entity/DefaultEntityManager/* directory in practice. This means that Doctrine entities located in the directory *CustomNodesBundle/Entity/DefaultEntityManager/Custom/* will also be managed by the default entity manager. The *Custom* subdirectory is intended to be used for User-defined Doctrine entities, described in section 4.3.3.

The pre-defined Doctrine entity classes are the following:

- *NodeConfig* – holds data about a custom Doctrine entity to be generated. Users can specify a name for the entity and an optional database table name if it differs from the entity name. Additionally, the NodeConfig class contains information about the entity managers to be associated with the custom Doctrine entities to be generated. In practice, this determines which database connections

would be used for the custom Doctrine entities. A NodeConfig class instance can also hold one or several *NodeConfigField* objects.

- *NodeConfigField* – holds data about a single class property that a custom generated Doctrine entity will have. This is used to define a database column name for a custom generated Doctrine entity. The NodeConfigField class also contains a *NodeConfigOptions* object. The NodeConfigOptions class is not a Doctrine Entity in itself. It is simply a JSON-serializable class, which includes database-level metadata about a field, defining properties such as whether the field is nullable and whether there are any field length restrictions.
- *DiagramConfig* – extends the *NodeConfig* class. One reason for this class definition is to provide certain pre-defined values. For example, it was known already during the project's design time that each diagram needed to have at least the following properties: source (indicating the data source of the entity object), and measurement_time (indicating the time of the time series data measurement point). Additionally, by having the *DiagramConfig* defined as a separate class from NodeConfig, it is easier to distinguish between custom generated Doctrine entities which are used as data sources, from those used for the consolidated diagram data.
- *DataSourceToDiagramMapping* – holds information of which data source (a Doctrine entity generated from a *NodeConfig*) is mapped to which diagram (a Doctrine entity generated from a *DiagramConfig*). The *DataSourceToDiagramMapping* also contains one or more *FieldMapping* class instances, as well as one ore more *ConsolidationState class instances*.
- *FieldMapping* – is used within the context of *DataSourceToDiagramMapping*. The FieldMapping class holds information about the data source entity getter, as well as the target diagram entity getter and setter. The point is to define how a data source entity field can be translated to a diagram field.
- ConsolidationState – is used within the context of DataSourceToDiagramMapping. This class holds information about the consolidation type (indicating a sampling interval, such as hourly, daily, or monthly), and the last measurement time that has been processed during upsampling or downsampling. This is described further in section 6.

4.3.3 User-defined Doctrine Entities

User-defined Doctrine entities are a crucial part of the solution to one of the main problems in this project, which arose from my decision to use Doctrine ORM. The user-

defined entities, which I also refer to as *custom generated Doctrine entities*, are needed in order to interact with the database data indirectly using PHP classes and objects. As described briefly in section 4.3.2 the *NodeConfig* and *DiagramConfig* classes are pre-defined Doctrine entities, which hold data of how a user-defined Doctrine entity should be generated. The *NodeConfig* class is used for data source user-defined Doctrine entities, and the *DiagramConfig* – for the resulting diagram user-defined Doctrine entities, which represent the consolidated data.

In order to explain the workflow, let us take a look at Figure 7, which visualizes the full process of how a data source user-defined Doctrine entity is generated.

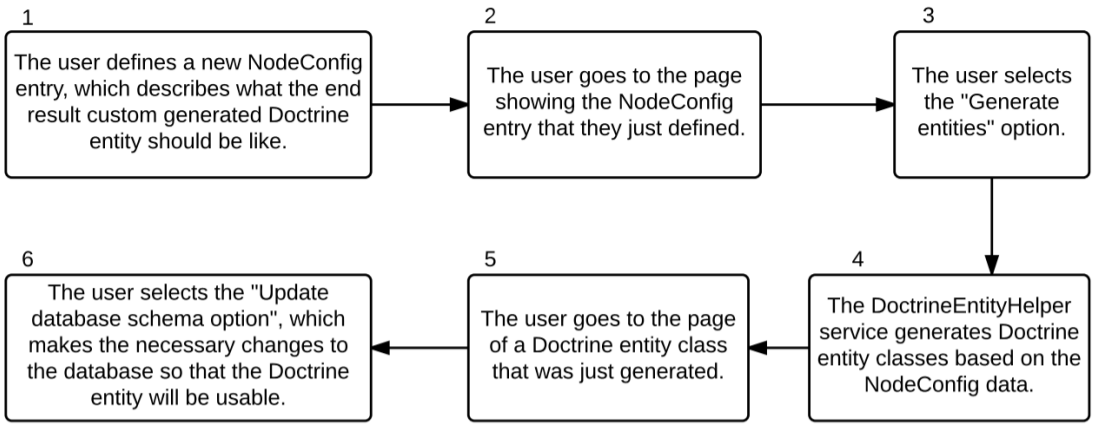

Figure 7. The process of generating a user-defined Doctrine entity to be used as a data source.

Figure 7 is rather self-explanatory for the purpose of the higher-level understanding of the process. However, there is an important component in step 4 in the figure: the *DoctrineEntityHelper* class. This is a custom class that I wrote, which handles various types of custom functionality related to Doctrine entities (hence the rather vague class name). The interesting functionality here is how a custom Doctrine entity is generated, shown in Listing 19.

```
protected function generateEntityFromNodeConfig($fullyQualifiedClassName)
{
...

    // Base directory for the generated entity classes.
    // Note that the entity namespace will control the subdirectory
structure.
    $customEntityDirectory = $this->getBaseSrcDirectory();

    $em = $this->container->get('doctrine')->getManager();
...

    // Set a custom driver that generates doctrine entity ClassMetadata
based on custom node configuration.
    $driver = new CustomNodesDriver($em);
    $em->getConfiguration()->setMetadataDriverImpl($driver);

    // Setup the factory for generating ClassMetadata.
    $cmf = new DisconnectedClassMetadataFactory();
    $cmf->setEntityManager($em);
    // Setup the class name with a proper namespace for this custom node
config.
    $classMetadata = $cmf->getMetadataFor($fullyQualifiedClassName);

    $generator = new EntityGenerator();
    $generator->setUpdateEntityIfExists(true);
    $generator->setGenerateStubMethods(true);
    $generator->setGenerateAnnotations(true);
    // Fully regenerate existing entities, since the changes to them may
be major.
    $generator->setRegenerateEntityIfExists(true);
    // Do not make backups. The user has already been warned that the
entity will be overwritten.
    $generator->setBackupExisting(false);
    $generator->writeEntityClass($classMetadata, $customEntityDirectory);
}
```

Listing 19. DoctrineEntityHelper's method for generating a custom Doctrine entity based on a NodeConfig.

As Listing 19 shows, I am using a Doctrine driver that I wrote, called *CustomNodesDriver*. The purpose of this driver is to build Doctrine class metadata in the proper format, based on the information that a NodeConfig object provides. I had to read some of Doctrine's ready-made drivers in order to figure out some of the details of how this functionality can be achieved. Nevertheless, once the class metadata is built correctly with the help of the *CustomNodesDriver* and a Doctrine-provided *DisconnectedClassMetadataFactory*, it is possible to write the file defining the PHP class for a custom Doctrine entity, with all the relevant Doctrine annotations.

Once the user-defined Doctrine entities are generated (in step 4 in Figure 7), they end up in the corresponding directories, as shown in Figure 8.

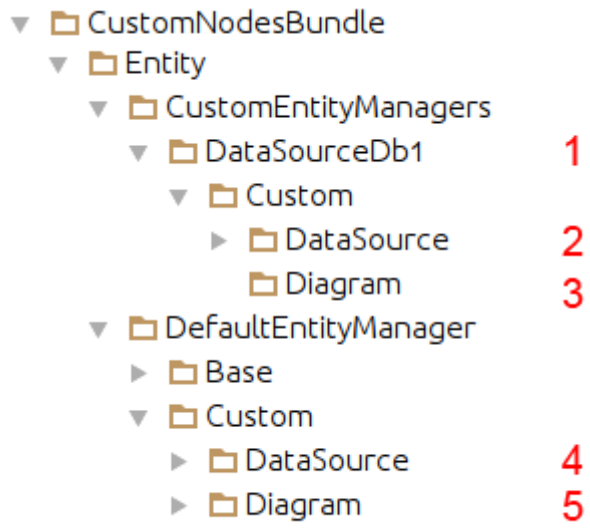

Figure 8. Directory structure showing the places where user-generated Doctrine entities are written.

In Figure 8 the directory numbered *1* is based on the name of a custom entity manager (a user-provided database connection), which is named *data_source.db_1* in the configuration. When translated to a directory and PHP namespace name, it uses the upper camel case notation, i.e. *DataSourceDb1*. User-defined Doctrine entities, which use the *data_source.db_1* entity manager, are placed in the directory numbered *2* if they are generated from a *NodeConfig*, or in the directory numbered *3* if they are generated from a *DiagramConfig*. The same logic goes for the user-defined Doctrine entities which use the *default* entity manager: they either go to the directory numbered *4* (when using *NodeConfig*), or to the directory numbered *5* (when using *DiagramConfig*).

In case there is a need to change the server host, in order to ensure that the user-defined entities are properly transferred, it would be the easiest to backup the *default* database containing NodeConfig and DiagramConfig entries, and to re-generate the custom Doctrine entities from them. While it is possible to just copy the generated Doctrine entity classes, they would not be easily modifiable if their corresponding NodeConfig or DiagramConfig objects do not exist.

5 Data mapping

Data mapping is an essential part of combining the data in this project. Since the administrator users are allowed to use different databases with different structures (for

example with different column names), it is important to be able to merge that data into a single structure, so that it can be processed, or visually compared, more easily. In the project this is achieved with the help of several classes. The basic process flow is illustrated in Figure 9.

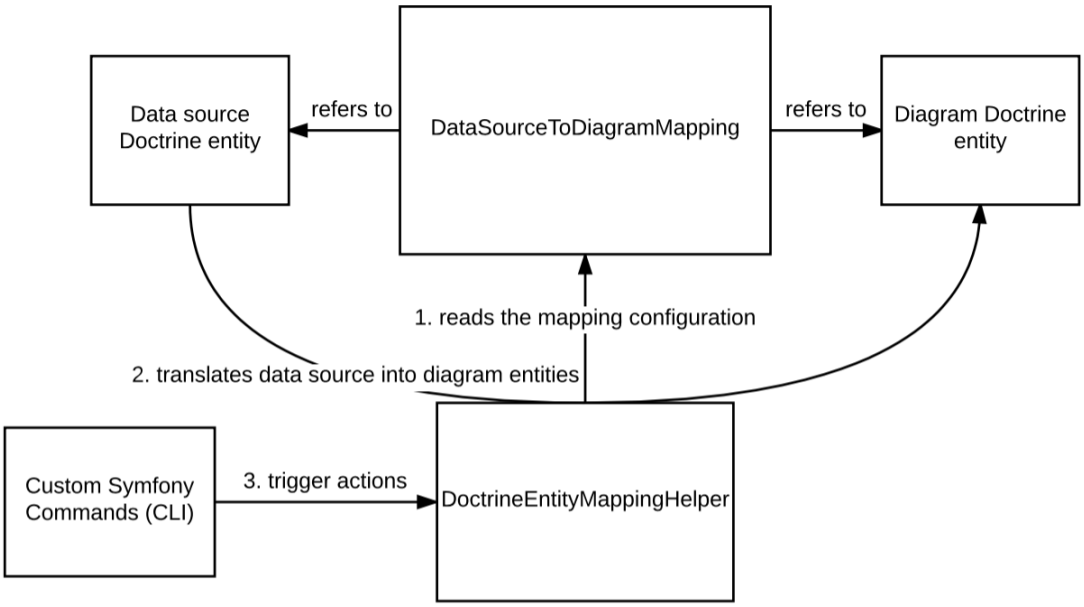

Figure 9. Interaction between software components in the Doctrine entity mapping done in the project.

The *DataSourceToDiagramMapping* is a pre-defined Doctrine entity, briefly described in section 4.3.2. It refers to a data source Doctrine Entity and to a diagram Doctrine entity by using their fully qualified class names (which include the PHP namespace and the class name). It also holds *FieldMapping* instances, each of which describe how a single field of the data source entity can be translated into a diagram entity field. The *DoctrineEntityMappingHelper* is a utility class that I defined, which reads the *DataSourceToDiagramMapping* (an action numbered *1* in Figure 9), and then translates (or maps) the relevant fields from the data source to the diagram entity (numbered *2* in Figure 9).

I also wrote two custom Symfony Commands, which make it possible to translate data source entities into diagram entities for a single, or multiple *DataSourceToDiagramMappings* respectively. Listing 20 demonstrates the use of the command that works

with a single *DataSourceToDiagramMapping* instance, whereas Listing 21 shows the use of the command that works with multiple instances.

```
$ php app/console app:process-entity-mapping 6 3

Translating entities from data source
'DataConsolidation\CustomNodesBundle\Entity\CustomEntityManagers\DataSour
ceDb1\Custom\DataSource\AirQuality' to diagram
'DataConsolidation\CustomNodesBundle\Entity\DefaultEntityManager\Custom\D
iagram\AirTrackingDiagram'.
Attempted to process data source entities measured later than '2017-04-
19T18:36:21+03:00'.
Processed a total amount of 0 entities. Maximum allowed amount was 3.
```

Listing 20. A custom Symfony Command that translates data source to diagram entities for a single DataSourceToDiagramMapping. In this case the ID of the DataSourceToDiagramMapping is 6, and the maximum allowed entities to be translated is 3.

```
$ php app/console app:process-entity-mappings
"DataConsolidation\CustomNodesBundle\Entity\DefaultEntityManager\Custom\D
iagram\AirTrackingDiagram" 10

Translating entities from data source
'DataConsolidation\CustomNodesBundle\Entity\CustomEntityManagers\DataSour
ceDb1\Custom\DataSource\AirQuality' to diagram
'DataConsolidation\CustomNodesBundle\Entity\DefaultEntityManager\Custom\D
iagram\AirTrackingDiagram'.
Attempted to process data source entities measured later than '2017-04-
19T20:13:15+03:00'.
Processed a total amount of 4 entities. Maximum allowed amount was 10.
Translating entities from data source
'DataConsolidation\CustomNodesBundle\Entity\DefaultEntityManager\Custom\D
ataSource\HelsinkiSensorTest' to diagram
'DataConsolidation\CustomNodesBundle\Entity\DefaultEntityManager\Custom\D
iagram\AirTrackingDiagram'.
Attempted to process data source entities measured later than '2017-04-
19T20:14:34+03:00'.
Processed a total amount of 2 entities. Maximum allowed amount was 10.
```

Listing 21. A custom Symfony Command that translates data source to diagram entities for a multiple DataSourceToDiagramMapping. In this case the fully qualified class name of the diagram doctrine entity is specified, followed by a limit of 10 for maximum allowed entities to be translated.

Let us take a bit deeper look into how a single field is translated from a data source entity into a diagram entity. This is done with the help of the data held by the *FieldMapping* objects. Figure 10 shows the properties that the class has.

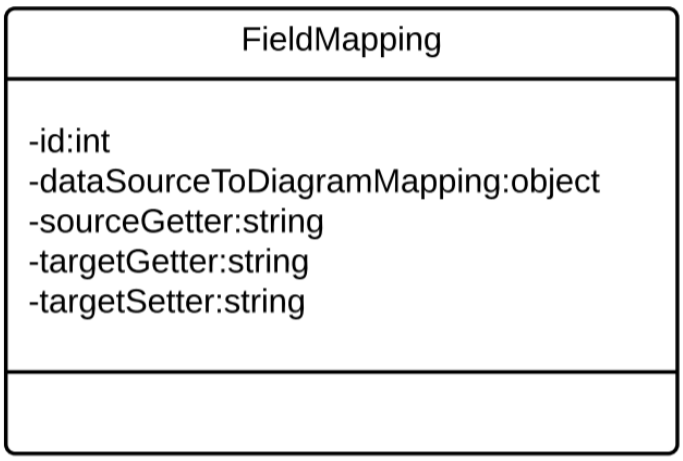

Figure 10. A UML class diagram of the FieldMapping Doctrine entity.

Each *FieldMapping* object contains a string property, named *sourceGetter,* for the getter method name of the data source entity. In this way it is possible to access the value in the data source entity during the translation (mapping) process to a diagram entity. In order to set the value to the diagram entity, I use the *targetSetter* string property, which holds the name of the setter for the respective class property in the diagram entity. Finally, the *targetGetter* property exists in order to make sure that there is a way to retrieve the diagram entity property value when visualizing the content into a diagram.

6 Implementation of Data Resampling

There are a couple of important decisions that I made during the practical implementation of the data resampling. First, I decided that the upsampling will be done through the use of linear interpolation, since to me that was the simplest solution to implement in practice. I decided that the upsampling would be performed on the original diagram data, and that the target sample rate interval would be one minute. The interpolated values would be stored to the table of the diagram Doctrine entity but with a different value for a property, which I named *consolidationType*. I defined several different possible *consolidationType* values:

- *none* - indicates the raw diagram data as it comes from the sensors.
- *minute* – indicates consolidated diagram data with a sampling interval of one minute. This is achieved by interpolating the data of *consolidationType none.*
- *hour* – indicates consolidated diagram data with a sampling interval of one hour. This data is built by downsampling the data of *consolidationType minute*.

- *day* – indicates data downsampled from *consolidationType hour* data entries. The resulting sampling interval is 24 hours.
- *month* – indicates downsampled data from *consolidationType day. The sampling interval is the average month length in a year.*

Since the linear interpolation may not be applied to all raw diagram data values at once, as that process could take a lot of computer resources and could potentially never complete successfully, it was necessary to solve this problem by introducing a database-persisted state for the progress of the interpolation. For this purpose I wrote a new Doctrine entity class named *ConsolidationState* (described briefly in section 4.3.2). A *ConsolidationState* object is essentially a combination of *DataSourceToDiagramMapping*, a *consolidationType*, and a property *lastMeasurementTime*, which stores the last processed measurement time in the upsampling or downsampling process. The *lastMeasurementTime* property is used in different ways for the upsampling and downsampling processes. The basic process flow of the linear interpolation is shown in Figure 11.

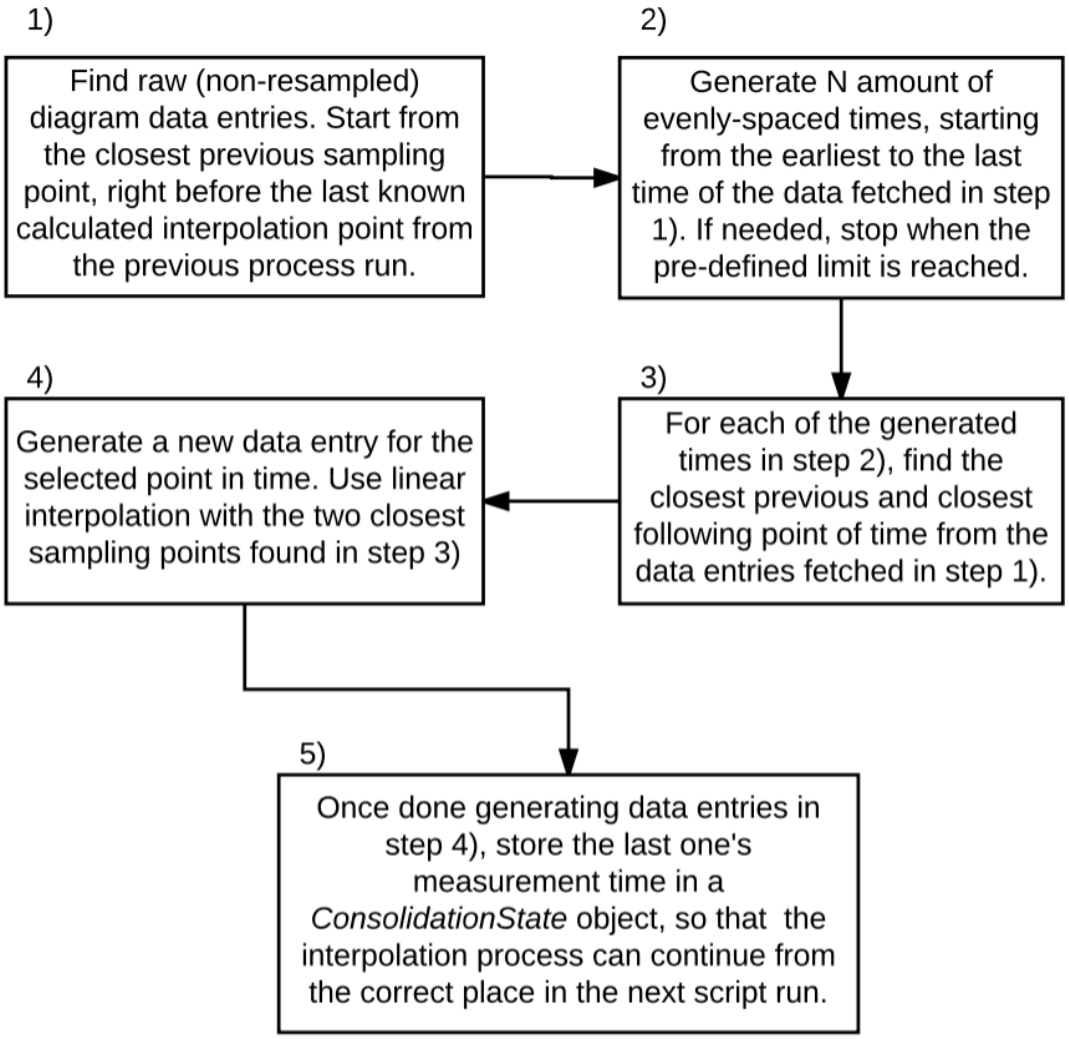

Figure 11. Process flow of how data is interpolated in the project. This takes into consideration how to apply sensible limits in processing the data, and how to resume from the last reached point during the next script run.

The downsampling in the project is built on the basis of calculating average values for a period of time. Downsampling is applied by reading data of *consolidationType minute*, *hour*, and *day*, and the resulting data is respectively of *consolidationType hour*, *day*, and *month*. This means that the downsampling relies on the fact that the data with *consolidationType* with the next higher sample rate, compared to the current one, has been processed. This is illustrated in Figure 12.

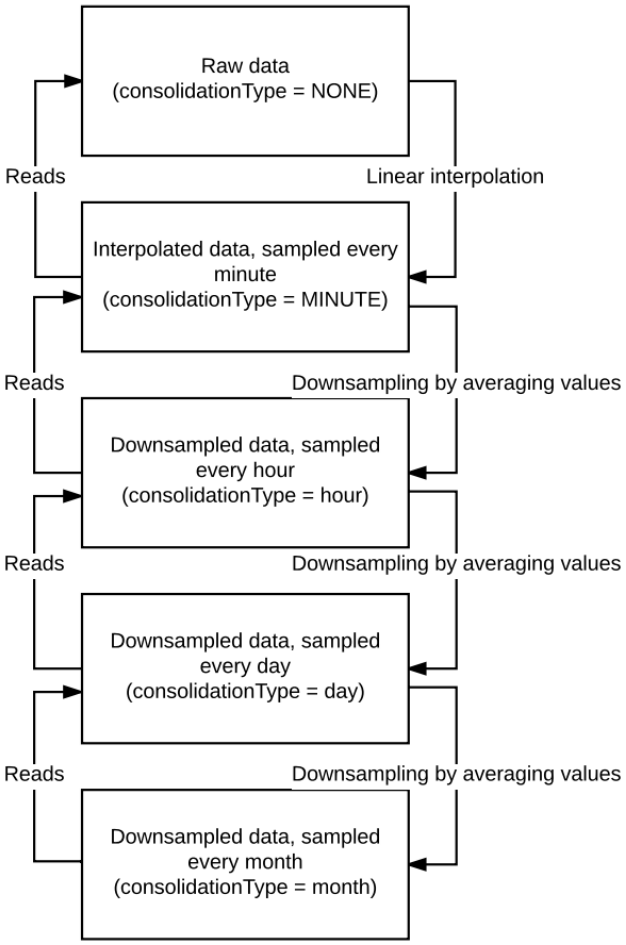

Figure 12. Dependencies between data of different *consolidationTypes* when applying up-sampling (interpolation) or downsampling in the project.

Similar to the interpolation, downsampling makes use of *ConsolidationState* objects to store the progress of the data processing. To explain the workflow of downsampling, it may be the easiest to take a look at Figure 13. With number *1* is marked the *ConsolidationState*'s *lastMeasurementTime* value. It indicates the last sampling point processed when the downsampling function was last executed. With number *2* is indicated the time interval, in which the last processed sampling point falls. In order to make sure the average of the data values are calculated correctly, it is necessary to fetch all points that fall in the time interval numbered *2*. This includes a previously processed sampling point (numbered *3*).

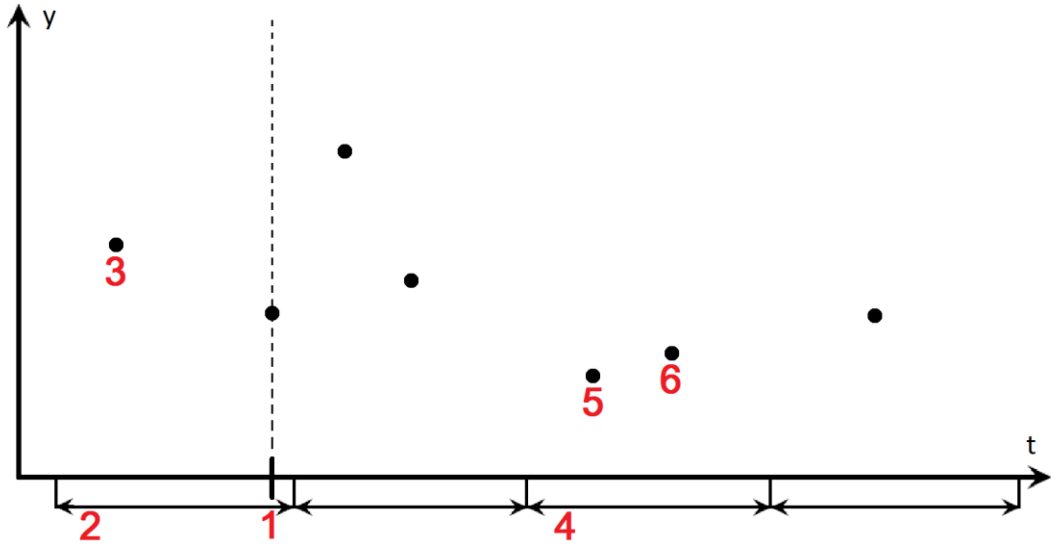

Figure 13. Sampling points processed during downsampling.

Continuing the explanation for Figure 13, the downsampling process is normally done with certain limits so that not all data is processed at once. Processing all data at once will cause problems when there is a lot of data. Let us assume that the downsampling process has reached the pre-defined limit, once it goes to the point numbered *5*. In order to calculate correctly the average value for the interval (numbered *4*), where the point falls, it is necessary to fetch all remaining points in the interval. In this case that would be the point numbered *6*. That means that essentially the pre-defined limit is sometimes exceeded but due to the data's *consolidationType* hierarchy, shown in Figure 12, the limit will not be exceeded by a lot.

I created a Symfony command for executing all downsampling tasks for a diagram. This command can be run by a crontab, for example, so that the process is automated. Listing 22 shows an example output of the command.

```
$ php app/console app:downsample-diagram
"DataConsolidation\\CustomNodesBundle\\Entity\\DefaultEntityManager\\Cust
om\\Diagram\\AirTrackingDiagram" 50 --data-
source="DataConsolidation\CustomNodesBundle\Entity\DefaultEntityManager\C
ustom\DataSource\HelsinkiSensorTest"

Downsampling entities for data source
'DataConsolidation\CustomNodesBundle\Entity\DefaultEntityManager\Custom\D
ataSource\HelsinkiSensorTest' to diagram
'DataConsolidation\CustomNodesBundle\Entity\DefaultEntityManager\Custom\D
iagram\AirTrackingDiagram'.
Processing target consolidation type: '1'
Read 0 and generated 0 amount of entities.
Processing target consolidation type: '2'
Read 0 and generated 0 amount of entities.
Processing target consolidation type: '3'
Read 0 and generated 0 amount of entities.
Processing target consolidation type: '4'
Read 5 and generated 3 amount of entities.
Finishing downsampling the diagram.
```

Listing 22. A custom Symfony command's output used for downsampling data of a diagram.

The usefulness of the resampling results is a strongly subjective matter. When using linear interpolation for the upsampling, I am essentially just drawing a line between each two existing sampling points of the raw data, and I am introducing new measurement points lying on that line. This may not be the best possible approach, as other interpolation methods could have produced better results. However, linear interpolation offers simplicity, and through using it I have achieved my main goal: providing data at a constant sample rate, which can then be processed more easily by the downsampling methods used.

7 Discussion

The choice of technologies in the project took most of my attention away from the main task: resampling of data. Since I chose to learn how to use the Symfony framework and to get some experience with it, it was definitely not the easiest path to achieving the goal of this project. Reflecting back on that choice, I feel that I needed to evaluate more carefully what would have been the most suitable set of technologies to be used for data consolidation. Using Doctrine has forced me into a specific way of designing things, and while this may be a good thing in some cases, I think that this did not work out too well here. There were a lot of workarounds that I needed to do in order to achieve tasks that seemed rather simple at first.

When using the chosen set of technologies, I noticed a tendency in myself to over-engineer (designing things in a more complicated way than necessary for the applica-

tion) some of the solutions. This left me with little time to explore the various types of mathematical methods that could be used in the resampling of data. Overall, the project was very time-consuming, and it felt that the majority of the implementation was just a preparation for the actual task that should have been achieved in the first place.

The custom D3.js classes that I defined (described in chapter 3) are not guaranteed to make it easier to understand the code for someone who is experienced with the D3.js library but who is unfamiliar with this project. This additional layer of object-oriented programming could have been skipped for the sake of allocating more time for the rest of the tasks.

The way that Doctrine entities are mapped to each other (described in chapter 5) is likely not the best solution if we measure how fast the data can be processed. If other technologies had been used instead, the same task could have been achieved with less computer memory consumption. While I have not tested my solution on a greater scale, I have doubts that it would scale well if there was a lot of data to be processed.

My solution to the database connections' configuration in Doctrine raises some concerns about security risks. As the configuration files need to be readable and writable by the web server user, this is a potential security issue. However, I did not find an alternative way to achieve the functionality that I wanted. Additionally, entering sensitive data, such as database connection credentials, should not be done over HTTP but rather over HTTPS, if it really needs to be accessible through a web UI. Admittedly, this is not something that belongs to the project's code itself but rather to the development and server environment. Nonetheless, it is something worth mentioning here as it was an issue that was not addressed while working on this thesis.

There may be some more exciting ways to achieve upsampling and downsampling of data, which remained unexplored due to me focusing too much on the wrong things. Nonetheless, I believe that the solutions used in this project demonstrate one way of how resampling can be done, even if it may be in the simplest form possible. The project did not deliver on the initial aim to compare the usefulness of the resampling results, as there was only a single solution implemented. In that sense, it is impossible to provide an objective comparison.

On a personal level this project has been a useful experience. I definitely learned a lot about Symfony and Doctrine and I perhaps know better not only how to use these technologies, but also when not to use them. While I would not personally use the resulting project of this thesis, I believe that one of the best ways to learn is by making mistakes. In that sense, this project has brought me some invaluable knowledge that I will try to remember in the future.

8 Conclusions

One of the goals of this project was to observe the processes involved in time series data consolidation. This has been largely achieved: in this paper I talked about using multiple databases, about merging the different data sources into a single database with the help of custom-defined mappings, and about resampling of data. Another goal of this project was to explore different types of mathematical methods involved in the process of resampling. This goal has only been partially achieved, in the sense that my implementation only demonstrates a single solution. The results would have been more useful if there were some practical comparisons between different methods.

Nonetheless, through the help of downsampling, end users are able to observe data visualized in a diagram in a way that is more easily understandable or more useful for statistical purposes. While some of the solutions used in this project are perhaps not optimal, they still demonstrate the main aspects involved in the process of data consolidation. Overall, this research can be useful in the case of planning the requirements for a similar project in the future. It may be worthwhile to think about the challenges faced here so that they can be avoided already in the project's planning phase.

If this specific project was to be continued, I would recommend that further study is done on the different methods for upsampling and downsampling. It could be useful to find an algorithm which is computationally efficient, and yet produces satisfactory results. If the project was to be carried out again, I would recommend careful planning in choosing the technologies used. While I have no suggestions on what they may be, I think that my choice of technologies was non-optimal.

References

1. Lyons. Understanding Digital Signal Processing. 2nd ed. Upper Saddle River, New Jersey: Prentice Hall Professional Technical Reference; 2004.

2. Poynton. Digital Video and HD: Algorithms and Interfaces. 2nd ed. Waltham MA, USA: Morgan Kaufmann Publishers; 2012.

3. Smith. Theory of Ideal Bandlimited Interpolation. In: Physical Audio Signal Processing. [online book].
URL: http://www.ccrma.stanford.edu/~jos/pasp/. Accessed 7 April 2017.

4. Sensiolabs. Symfony. Symfony Components [online]. Clichy Cedex, France: Sensiolabs International.
URL: http://www.symfony.com/components. Accessed 5 April 2017.

5. Sensiolabs. Symfony. The Bundle System (The Symfony Bundles Documentation) [online]. Clichy Cedex, France: Sensiolabs International.
URL: http://www.symfony.com/doc/2.8/bundles.html. Accessed 5 April 2017.

6. Adermann, Boggiano. Introduction - Composer [online]. Berlin, Germany: Composer, Dependency Manager for PHP.
URL: http://www.getcomposer.org. Accessed 5 April 2017.

7. Sensiolabs. Symfony. Getting Started With FOSUserBundle (The Symfony Bundles Documentation) [online]. Clichy Cedex, France: Sensiolabs International.
URL: http://www.symfony.com/doc/current/bundles/FOSUserBundle/index.html. Accessed 6 April 2017.

8. Sensiolabs. Symfony. Creating and Using Templates (2.8) [online]. Clichy Cedex, France: Sensiolabs International.
URL: http://www.symfony.com/doc/2.8/templating.html. Accessed 5 April 2017.

9. Sensiolabs. The flexible, fast, and secure template engine for PHP. Introduction - Documentation – Twig [online]. Clichy Cedex, France: Sensiolabs International.
URL: https://www.twig.sensiolabs.org/doc/2.x/intro.html. Accessed 7 April 2017.

10. Sensiolabs. The flexible, fast, and secure template engine for PHP. Block - Documentation - Twig [online]. Clichy Cedex, France: Sensiolabs International.
URL: https://www.twig.sensiolabs.org/doc/2.x/tags/block.html. Accessed 7 April 2017.

11. Sensiolabs. The flexible, fast, and secure template engine for PHP. Extends - Documentation - Twig [online]. Clichy Cedex, France: Sensiolabs International.
URL: https://www.twig.sensiolabs.org/doc/2.x/tags/extends.html. Accessed 5 April 2017.

12. Sensiolabs. The flexible, fast, and secure template engine for PHP. Macro - Documentation - Twig [online]. Clichy Cedex, France: Sensiolabs International.
URL: https://www.twig.sensiolabs.org/doc/2.x/tags/macro.html. Accessed 8 April 2017.

13. Sensiolabs. Symfony. Databases and the Doctrine ORM (2.8) [online]. Clichy Cedex, France: Sensiolabs International.
URL: http://www.symfony.com/doc/2.8/doctrine.html. Accessed 8 April 2017.

14. Doctrine Team. Welcome to the Doctrine Project [online]. Miami, USA: ServerGrove.
URL: http://www.doctrine-project.org/. Accessed 8 April 2017.

15. Sensiolabs. Chapter 11: Databases and Propel. In: Symfony – the Book. Version 2.8 [online]. Clichy Cedex, France: Sensiolabs International; 2016. p. 113.
URL: https://www.symfony.com/pdf/Symfony_book_2.8.pdf. Accessed 8 April 2017.

16. Sensiolabs. Symfony. DoctrineBundle Configuration ("doctrine") (The Symfony Reference) [online]. Clichy Cedex, France: Sensiolabs International.
URL: http://www.symfony.com/doc/2.8/reference/configuration/doctrine.html. Accessed 6 April 2017.

17. Sensiolabs. Symfony. Service Container (2.8) [online]. Clichy Cedex, France: Sensiolabs International.
URL: http://www.symfony.com/doc/2.8/service_container.html. Accessed 4 April 2017.

18. Luke Rotherfield. Unpacking Symfony2 compiler passes [online]. San Francisco, USA: DISQUS; 8 December 2014.
URL: http://www.lrotherfield.com/blog/unpacking-symfony2-compiler-passes/. Accessed 8 April 2017.