Bachelor's Thesis

Project Report

Reduction of Physical Supervision

Amahdya Delkescamp	256523
Mihail Kanchev	266106
Dominika Kubicz	266148

Supervisor:

Poul Væggemose

Number of characters: 76503

Software Engineering

7th Semester

18th December 2020

Table of contents

Abstract

The purpose of the Reduction of Supervision system is to produce a proof of concept for Herning Vand that demonstrates a system to monitor the pumps that the company manages at various locations around Denmark. This proof of concept will be constructed according to the requirements specified by Herning Vand. According to the Software Requirement Specification’s Product Functions(See Appendix A) The proof of concept should be able to analyze Herning Vand’s data sets and find patterns within the data that suggests an abnormality in the pump station functionality. These recordings and detected abnormalities should be stored in a database and displayed in an HTML web page for the user to view.
According to the Software Requirement Specification Constraints(See Appendix A), the system must be standalone, while the system uses sensors it is not meant to implement its own, the algorithm for anomaly detection may only rely on provided datasets for accuracy.

The technical aspect of the system involved a multi-layered architecture. The temperature and pressure sensors that were connected to an Arduino. A bridge application that made use of LoRaWAN to send the signal. While this does violate a delimitation on use of internet-based applications, it was necessary due to further constraints from Herning Vand. The web application is mainly built using Blazor Server and the EF Core framework for the database connection. The database itself is made using PostgreSQL and finally the algorithm for anomaly detection is made using Python and Flask.

The result of the project was a proof of concept that is able to collect data from the sensors, send the sensor data through LoRaWAN to a socket connection that the web application listens on. From there the sensor data is stored in the database and is retrieved by the web application, displayed to a UI in the form of a web page. The web application also allows for data input to be analyzed by an algorithm built in Python that will return a response detailing if the input contains anomalies that would suggest that employee intervention may be necessary.

Introduction

To paraphrase from (APEC Water 2013), water and sewage management and treatment, the impact on the environment and maintaining the consistency and reliability of such an important utility have posed a logistical challenge to society for as long as civilization has existed. Technology has increased to accomodate the need to properly manage waste water and countries, such as Denmark, have enjoyed the payoff. However, there are still some challenges that need to be overcome in reference to (USEPA, 1998) and (DANVA, 2020).

According to Herning Vand (Projects from Herning Vand. 2020, pp. 8), they monitor the condition of 170 water pumping stations and 14 wastewater treatment plants across Denmark for about 16 hours a day to ensure the consistent water quality and service. While the utility network is normally supervised with the help of a comprehensive SCADA system, there is still a need to check the water pumping stations manually to make sure that all devices in the pipes are working and to avoid potential breakdowns. The management at Herning Vand has noticed that such a method as a preventative measure against breakdowns within the system is not efficient or cost-effective.

This project will contribute to a way to help Herning Vand with efficiency along with reducing the possibility of an interruption in service due to a potentially preventable breakdown according to (Projects from Herning Vand. 2020, pp. 8).

Due to the complexity of the problem and the time allotted for the project, some limitations needed to be set. The result will be a proof of concept for the management of Herning Vand to consider and not a deployment ready system. Further this contributed to the delimitations of the project.

Delimitations

1. We will not consider video recording devices, such as video cameras, as a sensor to be monitored
2. We will not consider the usage of the internet.
3. We will not consider the general public as potential users
4. We will not consider the marketability of the system
5. We will not consider any literal adaptation or implementation between the system and Herning Vand's current SCADA system
6. We will not consider additional actuators in the system
7. We will not consider an android application.
8. We will not consider system notifications.
9. We will not consider machine learning pipelines.
10. We will not consider a PLC as an interface in the system.

Requirements

The Employee Entity

There will not be any levels of access considered for the system. One reason is that the system is expected to be used within Herning Vand's SCADA. Another reason is, as a result of being used within SCADA, it is expected that the users of the system will be employees who have already logged into the larger SCADA system before accessing the proof of concept system. The system is also not being considered as a marketable product and no further users will be identified.

Functional Requirements

1. Accurately label pump state by given pump attributes.
2. Store and display continuously collected sensor signals
3. Expose an API from python
4. Store information in a database
5. Host a web application

Non-Functional Requirements

1. Has a single user interface panel
2. Does not access the internet
3. Predictions are not automated (input should be done manually)
4. Utilizes a multi layered architecture
5. The system will be heterogeneous
6. The system will request an update from the database at least once an hour or when prompted by the user
7. The system will not include a login feature.
8. The system utilizes a LoRaWAN connection.

The proof of concept for Herning Vand will only be made for use by the company and will not be a fully fledged, marketable product. As mentioned in a correspondence with Herning Vand (A, Jeanette,. 2020), this also means that the system will not be allowed to connect or communicate with the SCADA system used by Herning Vand but should display the possibility for extension. There is also no need to further monitor existing recording devices, such as external cameras. Further, Herning Vand does not need any additional controls in the form of actuators as they already have the systems they want to use in place for such functionality.

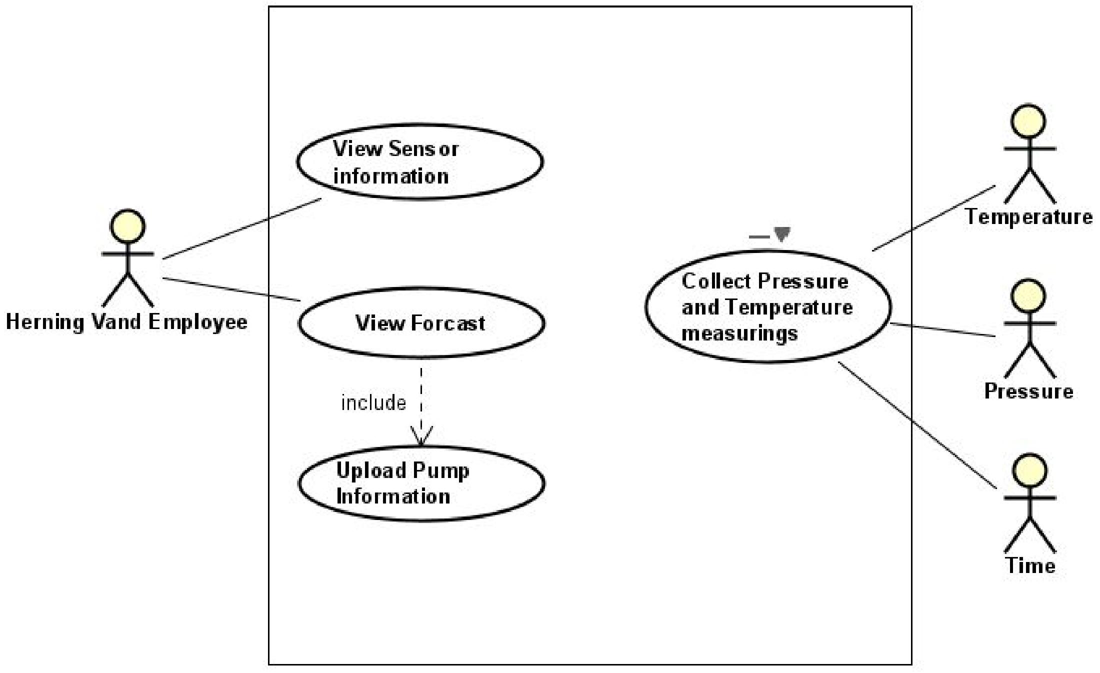
Due to the project being a proof of concept, the universal approach for the system will be oriented towards a web application, so it can run on devices that Herning Vand is operating. As a result of the limited amount of data points provided by the company, the prediction algorithm will be trained once beforehand and will not continue to learn after it is trained, resulting in the decision of a system with no notifications. Prototyping is a crucial part of the development process and time is of the essence, leaving us with complications when it comes to reenacting the PLC/Gateway that the company is operating.

The rest of the report to follow will contain details on the process that went into building the proof of concept for Herning Vand. The group worked according to Kanban and Unified Process and as such the sections below will detail work on the system from Inception and Requirements all the way to Transition and Testing.

Analysis

Based on the background of the problem, it became clear what the management at Herning Vand was expecting out of this a proof of concept. They wanted to see a system that monitors the data from various sensors within the pumps at the pumping station and is able to alarm workers at Herning Vand when an anomaly is detected in the pipes or equipment. With this, Herning Vand would only dispatch crews when it is necessary instead as a persistent preventative measure. Having such a system eventually included into their SCADA would increase the efficiency of Herning Vand's ability to prevent breakdowns within their utility network and maintain as much uptime as possible.

Figure - Use Cases



(For activity diagrams please visit Appendix B)

With the background, problems and requirements established, the use cases were derived. The interaction will primarily take place between a Herning Vand Employee and the system itself. In the case of View Forecast, the duration of the request is expected to last less than a second.

The interactions that a user will have with the system are the two use cases shown on the diagram. The first being a simple observation of the collected sensor data by the system sensors and the second being the forecast, returned as a result of inputting pump information.

Conceptual domain model

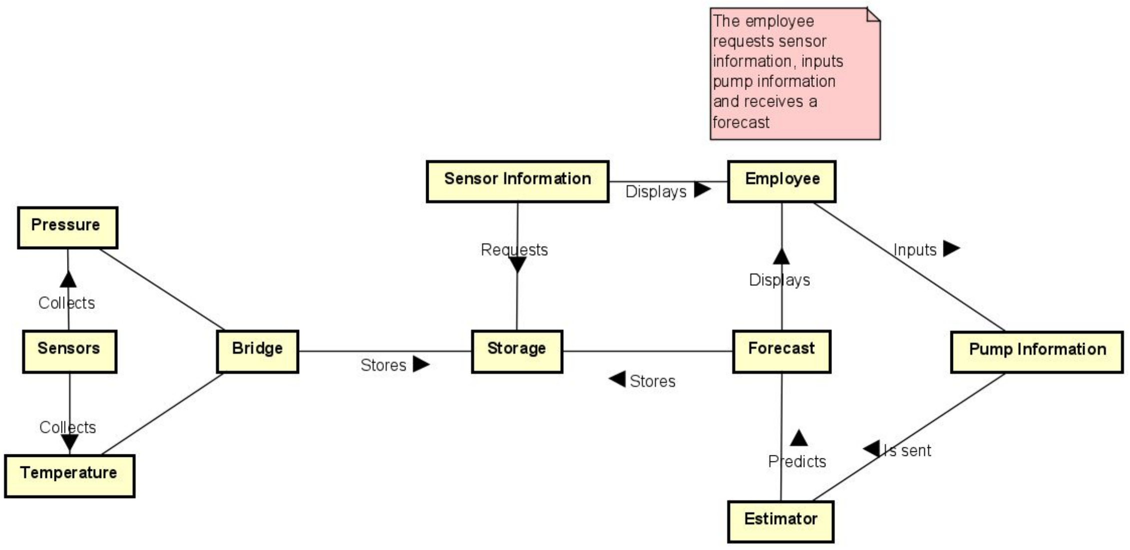


Figure - Conceptual Domain Model

This domain model was made with a web application in mind in response to web applications being used as a part of Herning Vand's own SCADA. An important consideration for the project was that the system to be made match the one at Herning Vand as closely as possible to lower the possibility for conflicts.

The Domain Model serves as a conceptual design model, or visual representation, of the system and mentions the domain objects such as potential data types, users and hardware parts of the system in accordance with Larman(2005, pp. 134).
It consists of two main parts which are communicating through the Storage, where data gets stored and retrieved by the respective parts.

The left side of the diagram, where the hardware part of the system is based, lie the sensors and the type of data they will be collecting - temperature and pressure. These sensors will have to establish a connection to the storage. This connection is labeled as "Bridge" and can be defined as two different approaches.

The first approach is the same model Herning Vand is using. The connection is established through a PLC with a gateway attached, which takes care of the task delivering the readings to a conceptual SCADA system. This approach is possible but highly unlikely due to its nature. In order for it to be tested, an engineer from the team has to travel to Herning Vand premises on a weekly basis in order to test it, which will greatly slow down the developing process.

The second approach is of an IoT nature. It is to our knowledge that Herning city has an operating LoRaWAN tower, similar to the one in Horsens. This is a great opportunity to expand on the currently operating systems in Herning Vand, while maintaining the same data protocols that their PLCs are using.

On the right side of the diagram is where the user interfaces and business logic lie. It is shown in the interactions that an Employee is able to extract previously collected sensor information, by the sensors on the left, from the storage. The extraction can be done automatically on a page load, in order to simplify the process to the user.

The second user interaction happens after the user inputs pump information in the system. This triggers a system response in which the system forecasts a result in the form of a label, telling the user whether the system operates in norm or there are abnormalities that need to be analyzed by a supervision employee.

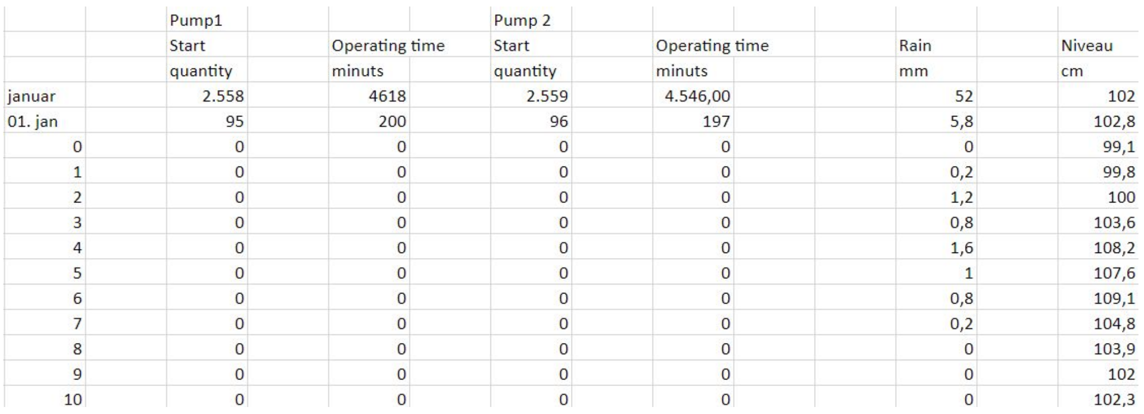
Dataset description

The forecast response will be based on information, granted by the company.
Herning Vand collects sensor readings from a multitude of pumping stations which are later stored in their SCADA system. In order to observe the collected data and deduce a method for forecasting, some sample datasets were requested from the company.
The provided data was collected through the period of 2017 to 2020. It consists of information from a couple of pumping stations, as well as a separate dataset which holds previously labeled breakdowns, manually recorded by their employees. Each pumping station dataset, provided by the company, holds the values of two pumps.

- The pumping stations data readings hold operating time, water quantity, rain quantity and the level of water in the pumps per every hour.

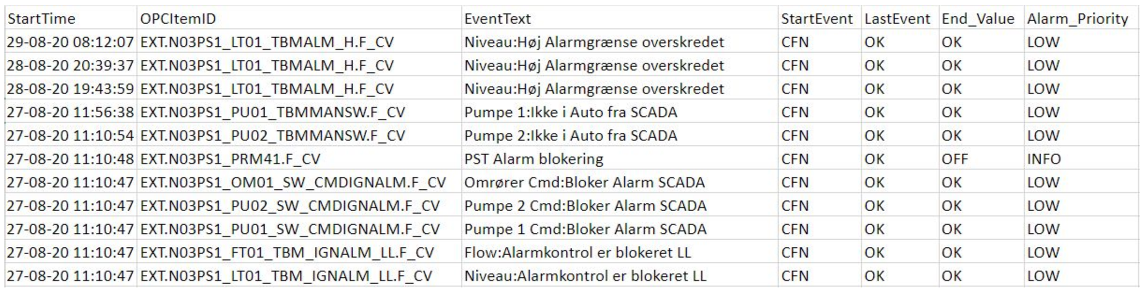
- The provided list of alarms consists of pumps and their respective breakdowns with accurate date and time.

Figure - Herning Vand pump dataset example



	Pump1		Pump 2				
	Start	Operating time	Start	Operating time		Rain	Niveau
	quantity	minuts	quantity	minuts		mm	cm
januar	2.558	4618	2.559	4.546,00		52	102
01. jan	95	200	96	197		5,8	102,8
0	0	0	0	0		0	99,1
1	0	0	0	0		0,2	99,8
2	0	0	0	0		1,2	100
3	0	0	0	0		0,8	103,6
4	0	0	0	0		1,6	108,2
5	0	0	0	0		1	107,6
6	0	0	0	0		0,8	109,1
7	0	0	0	0		0,2	104,8
8	0	0	0	0		0	103,9
9	0	0	0	0		0	102
10	0	0	0	0		0	102,3

Figure - Herning Vand alarm list dataset example



StartTime	OPCItemID	EventText	StartEvent	LastEvent	End_Value	Alarm_Priority
29-08-20 08:12:07	EXT.N03PS1_LT01_TBMALM_H.F_CV	Niveau:Høj Alarmgrænse overskredet	CFN	OK	OK	LOW
28-08-20 20:39:37	EXT.N03PS1_LT01_TBMALM_H.F_CV	Niveau:Høj Alarmgrænse overskredet	CFN	OK	OK	LOW
28-08-20 19:43:59	EXT.N03PS1_LT01_TBMALM_H.F_CV	Niveau:Høj Alarmgrænse overskredet	CFN	OK	OK	LOW
27-08-20 11:56:38	EXT.N03PS1_PU01_TBMMANSW.F_CV	Pumpe 1:Ikke i Auto fra SCADA	CFN	OK	OK	LOW
27-08-20 11:10:54	EXT.N03PS1_PU02_TBMMANSW.F_CV	Pumpe 2:Ikke i Auto fra SCADA	CFN	OK	OK	LOW
27-08-20 11:10:48	EXT.N03PS1_PRM41.F_CV	PST Alarm blokering	CFN	OK	OFF	INFO
27-08-20 11:10:47	EXT.N03PS1_OM01_SW_CMDIGNALM.F_CV	Omrører Cmd:Bloker Alarm SCADA	CFN	OK	OK	LOW
27-08-20 11:10:47	EXT.N03PS1_PU02_SW_CMDIGNALM.F_CV	Pumpe 2 Cmd:Bloker Alarm SCADA	CFN	OK	OK	LOW
27-08-20 11:10:47	EXT.N03PS1_PU01_SW_CMDIGNALM.F_CV	Pumpe 1 Cmd:Bloker Alarm SCADA	CFN	OK	OK	LOW
27-08-20 11:10:47	EXT.N03PS1_FT01_TBM_IGNALM_LL.F_CV	Flow:Alarmkontrol er blokeret LL	CFN	OK	OK	LOW
27-08-20 11:10:47	EXT.N03PS1_LT01_TBM_IGNALM_LL.F_CV	Niveau:Alarmkontrol er blokeret LL	CFN	OK	OK	LOW

Design

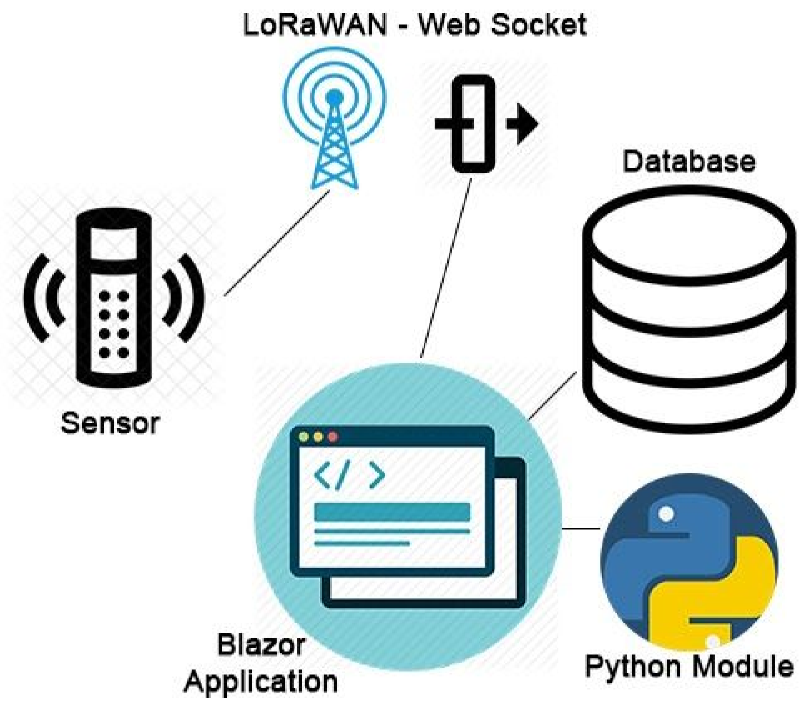


Figure - Architecture Design

System Architecture Description

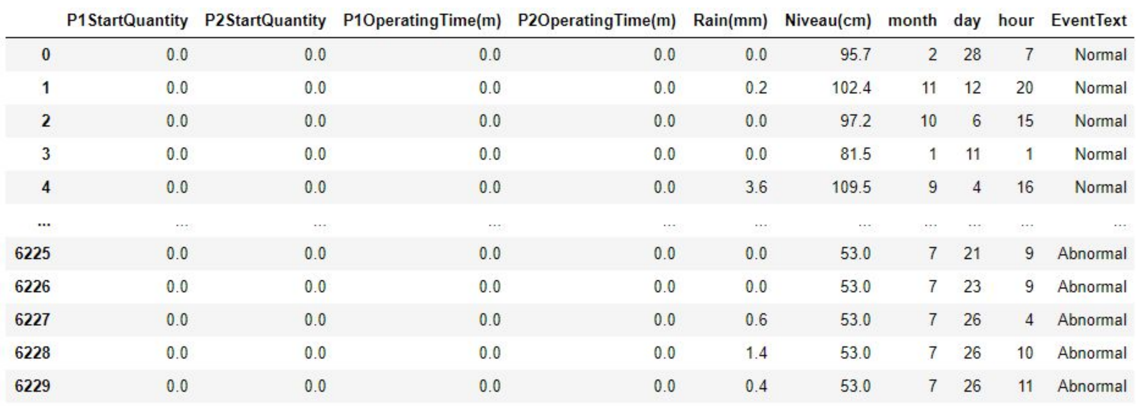
The architecture used for this system will be multi-layered and heterogeneous. Sensors will be sending signals to the local LoRaWAN tower. A Blazor server listens for the sensor readings coming from the LoRa server, stores them in a Postgres database with the help of Entity Framework Core and displays them to a Web Application. The web application establishes a connection to the python module through an API, by passing required values for a prediction. In return the python module sends back a prediction based on the respective values. That prediction is stored in the database on retrieval and displayed to the user through the Web Application.

Fitting a Machine Learning model

Peeking into the datasets

As of currently, Herning Vand collects information about its pumping stations and stores it in their SCADA system. The collected information comes both from sensors and employees. It is not to our knowledge whether their data pipeline continues after the data is stored in a database. For the purpose of this project, the gathered data is not being used after it is collected and stored. As an argument to the assumption please view *(Herning Vand pump dataset example) and (Herning Vand alarm list dataset example).* Some of the attributes in those tables are stored specifically for ease of viewing purposes (f.eks "januar, 01 jan, 00" is not a good way to record datetime if you would want to iterate through the dataset).

To proceed with the design, a thorough data cleaning was conducted, where every attribute was parsed to its respective datatype, every row was assigned Abnormal, if it had an alarm of any type occurring during that time, or Normal if it was operating under norms. Then all of the information was merged into a single dataset, holding attributes which are either measurable or labels.



	P1StartQuantity	P2StartQuantity	P1OperatingTime(m)	P2OperatingTime(m)	Rain(mm)	Niveau(cm)	month	day	hour	EventText
0	0.0	0.0	0.0	0.0	0.0	95.7	2	28	7	Normal
1	0.0	0.0	0.0	0.0	0.2	102.4	11	12	20	Normal
2	0.0	0.0	0.0	0.0	0.0	97.2	10	6	15	Normal
3	0.0	0.0	0.0	0.0	0.0	81.5	1	11	1	Normal
4	0.0	0.0	0.0	0.0	3.6	109.5	9	4	16	Normal
...	...	...	...	...	...	...	...	...	...	...
6225	0.0	0.0	0.0	0.0	0.0	53.0	7	21	9	Abnormal
6226	0.0	0.0	0.0	0.0	0.0	53.0	7	23	9	Abnormal
6227	0.0	0.0	0.0	0.0	0.6	53.0	7	26	4	Abnormal
6228	0.0	0.0	0.0	0.0	1.4	53.0	7	26	10	Abnormal
6229	0.0	0.0	0.0	0.0	0.4	53.0	7	26	11	Abnormal

Figure - The cleaned and labeled dataset

Plotting the data

Another important step in the data preparation process is the balance of the dataset. Out of 250500 rows, 3115 are labeled as alarms. This puts the ratio of *“Alarms : Not alarms”* at *“1 : 80”*.

Due to the high dimensionality of the dataset (holding 9 features) it is impossible to plot the dataset imbalance.

In order to solve that problem, a dimensionality reduction approach will be used called ***PCA***.

- **Principal Component Analysis (PCA)** identifies the axes that account for the largest amount of variance in a dataset. (220/Hands-on MAL)

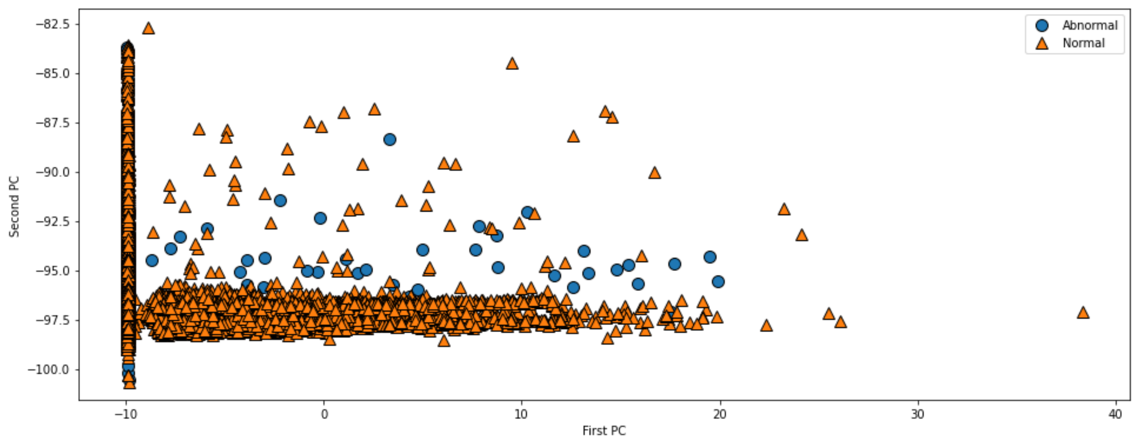


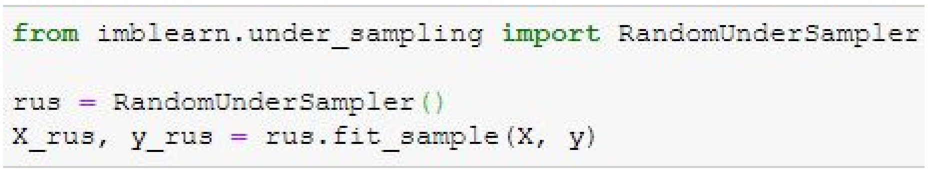
Figure - A two dimensional plot of 2 of the highest variance Principal Components of the dataset. The plot attains 92% of the original variance

Referring to the plot, it can be seen that the label imbalance in the dataset results in a lot of noise, preventing us from being able to put a straight line between points labeled as alarms (Blue/Abnormal) and points labeled as normal system behaviour (Orange/Normal). Balancing the dataset better, can help us visualize the domain of the problem clearer.

After running some calculations, it turns out that most of the alarms have occurred when the system was idle (Operating Time and Pump Quantity holding the values of 0). To be precise 3006 out of 3115 alarms happened during the idle periods of the system.

To properly balance the dataset, there must be equal amounts of rows during idle and operating time, labeled as alarm and normal behaviour respectively.

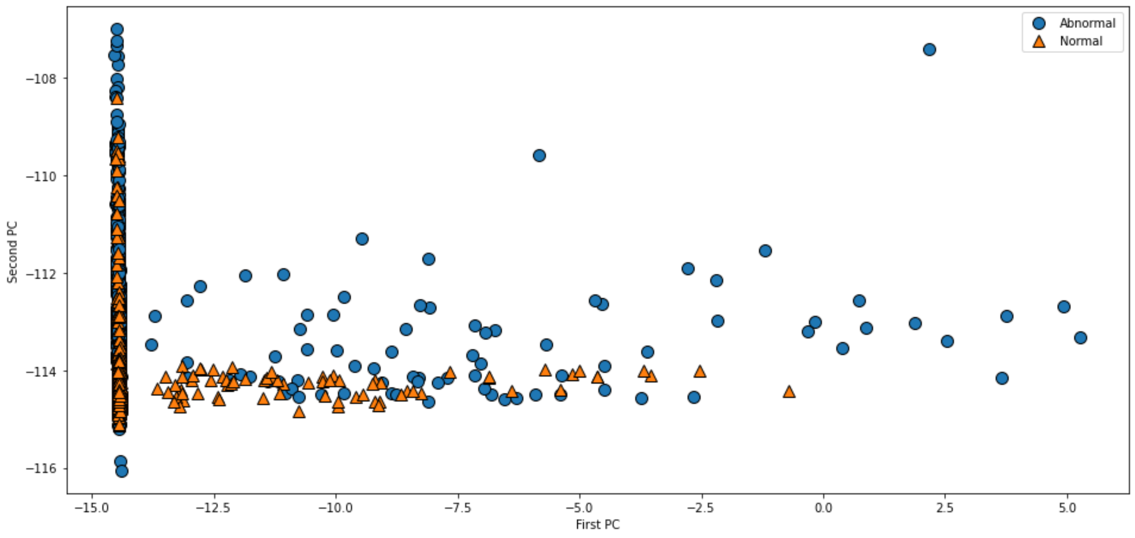
In order to balance the dataset, it has to be undersampled. Python has a library for random undersampling, which automatically balances a given dataset.



```python
from imblearn.under_sampling import RandomUnderSampler

rus = RandomUnderSampler()
X_rus, y_rus = rus.fit_sample(X, y)
```

Figure - Example of the undersampling

The result of the balanced dataset returned a dataset size of 6229. Plotting that data will give us a cleaner picture of the domain of the problem.



(A two dimensional plot of 2 of the highest variance Principal Components of the balanced dataset. The plot attains 93% of the original variance)

Ensemble methods

(In order to read more about the different models trained on the dataset, visit "Fitting Multiple Models" in the Appendix C/Model_Research.pdf.)

So far the performance is good, but there are no other models tested in order to define the best fit model to our case. Using a decision tree model seems like the correct approach upon which we can build on. Another type of machine learning model that we can use, derived from the decision tree model is called *Random Forests*.

- You can train a group of Decision Tree classifiers, each on a different random subset of the training set. To make predictions, you obtain the predictions of all the individual trees, then predict the class that gets the most votes. Such an ensemble of Decision Trees is called a **Random Forest**, and despite its simplicity, this is one of the most powerful Machine Learning algorithms available today. (189/Hands-on MAL)

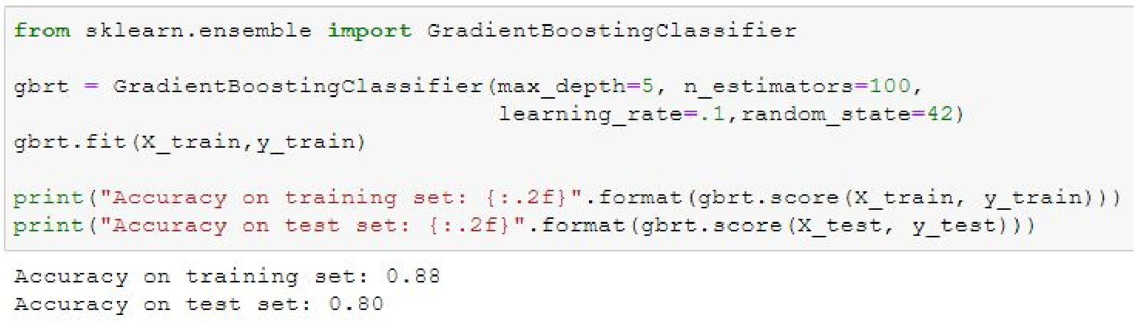
There are a couple of models making use of Random Forests. In order to pick the best one fit for the domain of the problem, we need to look deeper into them. To be precise, it has to be understood what the method of *Boosting* means.

- **Boosting (hypothesis boosting)** refers to any Ensemble method that can combine several weak learners into a strong learner. The general idea of most boosting methods is to train predictors sequentially, each trying to correct its predecessor. (199/Hands-on MAL)

There is a model based on random forests that involves Boosting, called *Gradient Boosting*.

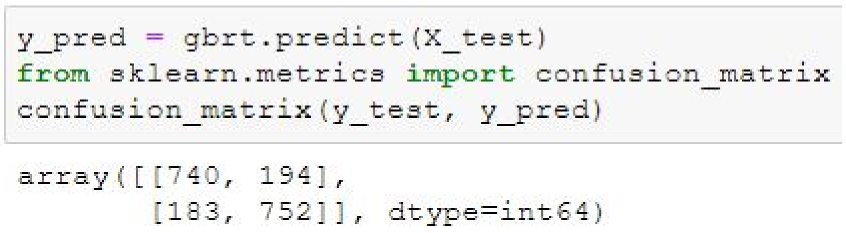
- **Gradient Boosted** trees often use very shallow depths of one to five, which make the model smaller in terms of memory and make predictions faster. The main idea behind **gradient boosting** is to combine many simple models. Each tree can only provide good predictions on part of the data, and so more and more trees are added to iteratively improve performance. **Gradient boosted** trees are frequently the winning entries in machine learning competitions, and are widely used in the industry. (90-91/Introduction to MAL)

Random Forests



```
from sklearn.ensemble import GradientBoostingClassifier

gbrt = GradientBoostingClassifier(max_depth=5, n_estimators=100,
                                  learning_rate=.1,random_state=42)
gbrt.fit(X_train,y_train)

print("Accuracy on training set: {:.2f}".format(gbrt.score(X_train, y_train)))
print("Accuracy on test set: {:.2f}".format(gbrt.score(X_test, y_test)))
```

```
Accuracy on training set: 0.88
Accuracy on test set: 0.80
```



```
y_pred = gbrt.predict(X_test)
from sklearn.metrics import confusion_matrix
confusion_matrix(y_test, y_pred)
```

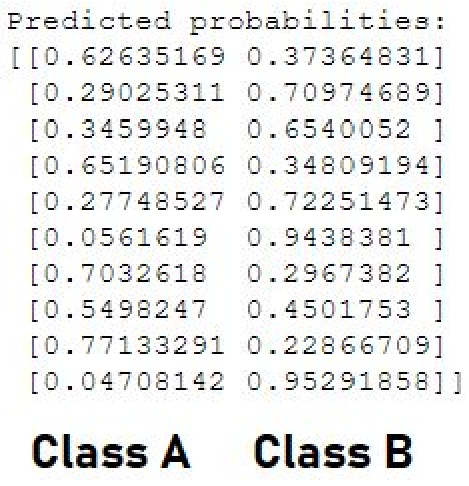
```
array([[740, 194],
       [183, 752]], dtype=int64)
```

Not a big improvement upon the decision tree model, but an improvement nevertheless. So far the accuracy jumped from ⅔ using linear models to ⅘ using gradient boosting model .

The accuracy of 80% is not perfect. In order to prove that, the prediction probabilities need to be observed.

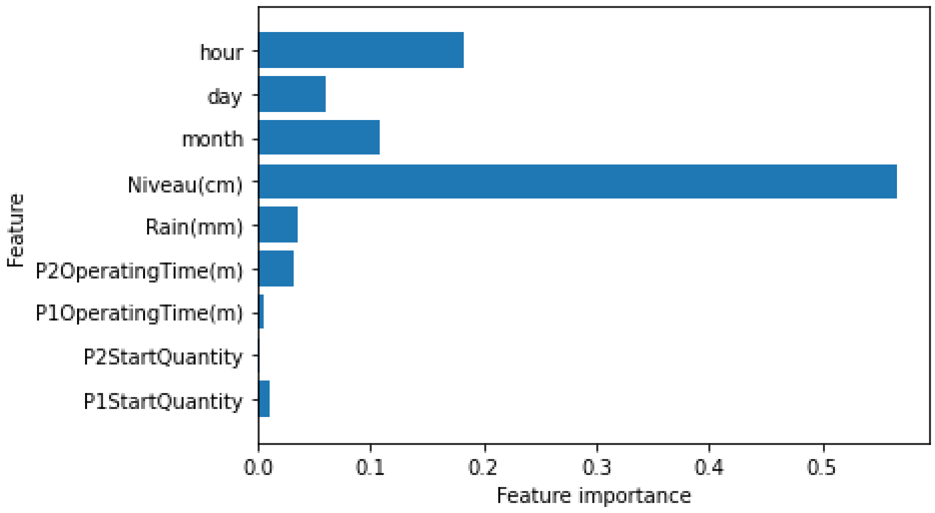
It can be seen that not all rows are a clear prediction for the model. In some cases the model has to pick between 55% of it being an alarm and 45% of not being an alarm. Making such close calls is another proof to how low the variance of the data is.

Random forests derive from decision trees, it is possible to look at the feature importance in the gradient boosted model.



```
Predicted probabilities:
[[0.62635169 0.37364831]
 [0.29025311 0.70974689]
 [0.3459948  0.6540052 ]
 [0.65190806 0.34809194]
 [0.27748527 0.72251473]
 [0.0561619  0.9438381 ]
 [0.7032618  0.2967382 ]
 [0.5498247  0.4501753 ]
 [0.77133291 0.22866709]
 [0.04708142 0.95291858]]
```

It can be seen that the gradient boosted model values Operating time a bit more than the decision tree model. It even puts little emphasis on Start quantity, but the overall feature importance is the same as the one in the decision tree model.

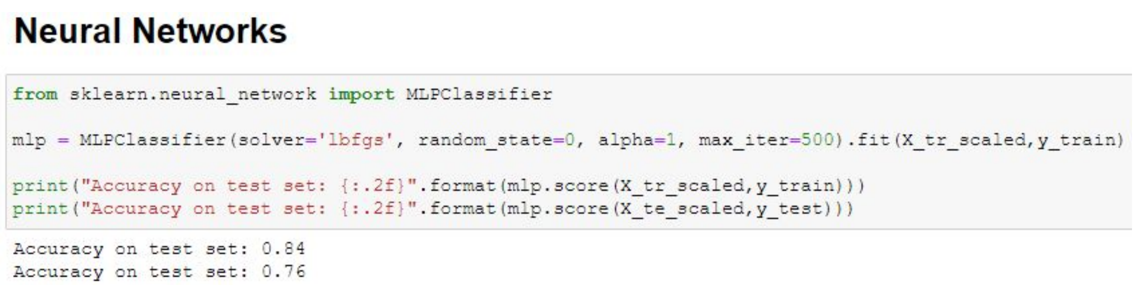


Neural Networks

The potential of neural networks is great, because every algorithm can be tuned by its multitude of hyperparameters, fitting it closely to provide accurate results. To demonstrate the neural networks potential, a neural network algorithm called **MLP** was trained on the data.

- An **MLP** is a class of artificial neural networks and consists of at least three layers - an input layer, a hidden layer and an output layer. Except for the input nodes, each node is a neuron that uses a nonlinear activation function. **MLP** utilizes a supervised learning technique called backpropagation for training.(289/Hands-on MAL)

(Example of a MLP model trained on the pump dataset)



Neural Networks

```
from sklearn.neural_network import MLPClassifier

mlp = MLPClassifier(solver='lbfgs', random_state=0, alpha=1, max_iter=500).fit(X_tr_scaled,y_train)

print("Accuracy on test set: {:.2f}".format(mlp.score(X_tr_scaled,y_train)))
print("Accuracy on test set: {:.2f}".format(mlp.score(X_te_scaled,y_test)))
```

```
Accuracy on test set: 0.84
Accuracy on test set: 0.76
```

With little to no hyperparameter adjustments, it yields accuracy close to the one of the Gradient Boosted Forest model. Exploring the Neural Network algorithms can open up opportunities for more accurate predictions. The topic can be elaborated deeper if *deep learning* was applied as an approach too, adjusting more hidden layers for even more definite results.

But given that this project is a proof of concept, the Gradient Boosted Forest yields a surprising accuracy of 80% and will be used as a machine learning model in our system design.

Flask API

Since the system has no access to industrial apparatus, the input will be manual. The frequency of manual input is not high, which means that an API would fit the purpose. The model discussed above will be serialized, then deserialized into a Flask API server that listens for incoming PUT requests and returns predictions made by the model. Along with the predictions, the server would send the probability of the label, showing how sure the algorithm is of its prediction.

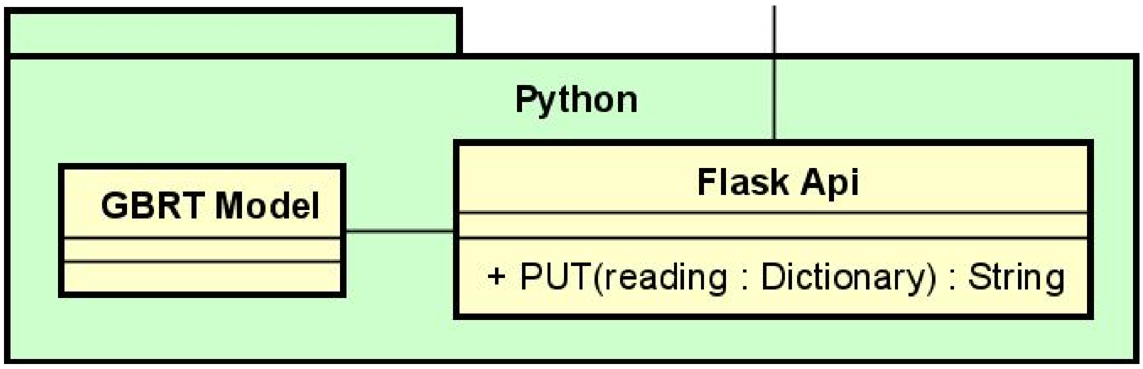


Figure - Class diagram of the API server

Web Application

ASP.NET Core Blazor

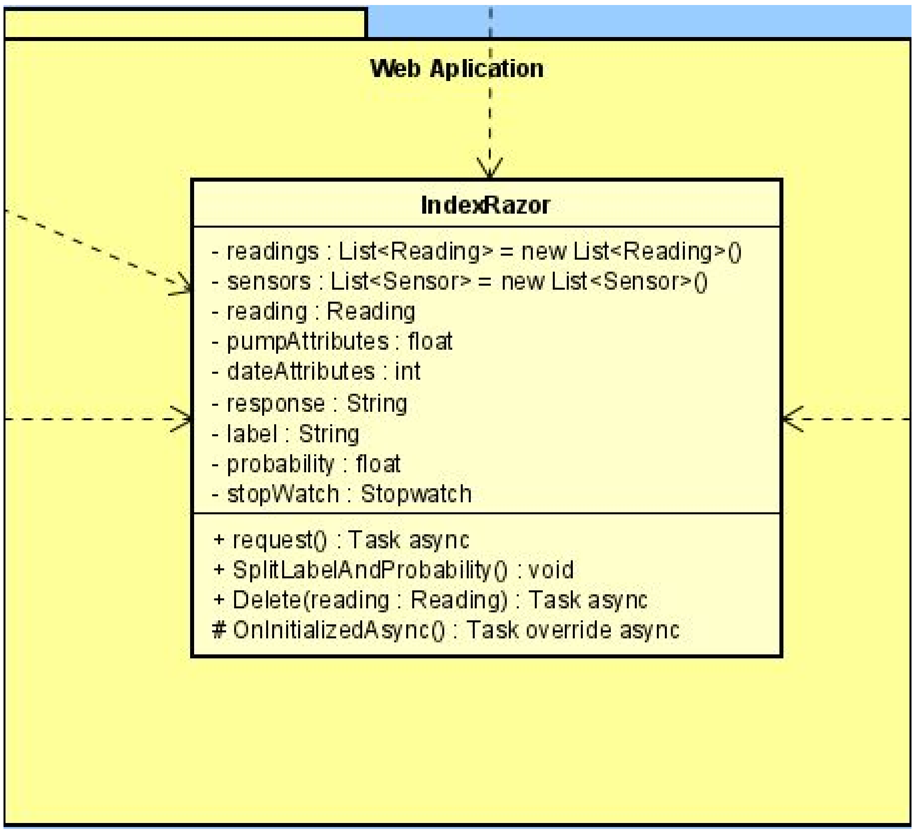

Web Aplication
IndexRazor
- readings : List<Reading> = new List<Reading>()
- sensors : List<Sensor> = new List<Sensor>()
- reading : Reading
- pumpAttributes : float
- dateAttributes : int
- response : String
- label : String
- probability : float
- stopWatch : Stopwatch
+ request() : Task async
+ SplitLabelAndProbability() : void
+ Delete(reading : Reading) : Task async
# OnInitializedAsync() : Task override async

Figure - Class diagram of Web Application

The user interface of this project will be displayed using a web application that will be built using an ASP.NET Core framework called Blazor. As mentioned previously, the decision to create a web application for this proof of concept came from Herning Vand's SCADA that already makes use of multiple web applications to allow users to view different aspects of the various pumping stations, pumps, etc.

Creating the system as a web application would help ensure that if the proof of concept were accepted as a potential product, the end result would most likely work well with the system that Herning Vand already has in place. Another reason for choosing a web application is based on the fact that web applications are flexible in where and how they can be used and viewed. For example, according to (ASP.NET Core Blazor, 2020), creating the system as a web application using Blazor would not only allow the system to be viewed online, it would also have the capability of being viewed on mobile as well due to the Blazor template and would be open to the addition of mobile functionality at a later time.

The choice for using Blazor was made based on a few factors. One was the group having familiarity with the Blazor framework and the C# language, since Blazor uses C# as it's main language instead of JavaScript. Due to Blazor being a part of .NET Core, it can utilize other .NET libraries as needed.

Normally when creating a web application using Blazor, one has a choice in hosting models between Blazor Server or Blazor Web Assembly(WASM) as covered by the hosting model documentation (ASP.NET Core Blazor Hosting Models, 2020). One main difference between the two is that Blazor Server has the server handle most of the processing and allows for a much lighter client, while Blazor WASM eliminates server-side dependency and places almost all responsibility in the client and browser. For this project, Blazor Server was chosen as the hosting model. According to (ASP.NET Core Blazor Hosting Models, 2020), with most responsibility being placed on the server, the web application's download size will be smaller and quicker to load, the framework is more mature and has more support from other .NET libraries, source code will not be downloaded to any clients, and the application should be easier to debug as another result of being a more mature framework.

There are some drawbacks to using Blazor Server however, these are not expected to cause problems with the project. Depending on how updates to the DOM are handled, Blazor Server can make use of a .NET library called Signal R. If used, updates to the client could suffer from latency issues, again according to (ASP.NET Core Blazor Hosting Models, 2020). However, the system is only expected to make calls to the database at least once an hour. This possible latency should not become an issue. Another potential drawback in the Blazor Server's difficulty in handling many clients at once. While it is not expected that the system will bear any load due to multiple users making calls to the database, a potential solution was considered in EF Core.

Entity Framework Core (EF Core)

(For the class design diagram, please refer to the Appendix C/Diagram_Images/Class_Diagram.jpg.)

EF Core is an open-source and cross-platform version of the Entity Framework, an object-relational mapping framework, that carried over its use as an object-relational mapper into .NET, as discussed in (Entity Framework Core, 2020). This library was chosen for use in this project because EF Core allows for the use of several database engines and does support PostgreSQL, a relational database that is planned for use in the system. EF Core also does not require any access code to be hand written for the database according to (Entity Framework Core, 2020). As a result, EF Core makes it easy to establish a database connection because all that is left to be done is to define the DbContext and database entities.

What becomes tricky is using EF Core with Blazor Server due to the way the DbContext works, as explained in (Blazor Server with Entity Framework Core, 2020). The DbContext normally exists for the lifetime of a component and only one is supposed to exist per user per component. However, Blazor Server would cause the DbContext to work as a shared instance and the DbContext is not designed for concurrent use. What is most important is to ensure that only one DbContext exists per user, component and operation, and this can be achieved through the use of a class called DbContextFactory. This class will create a new context per operation that will be destroyed when the operation finishes.

Database

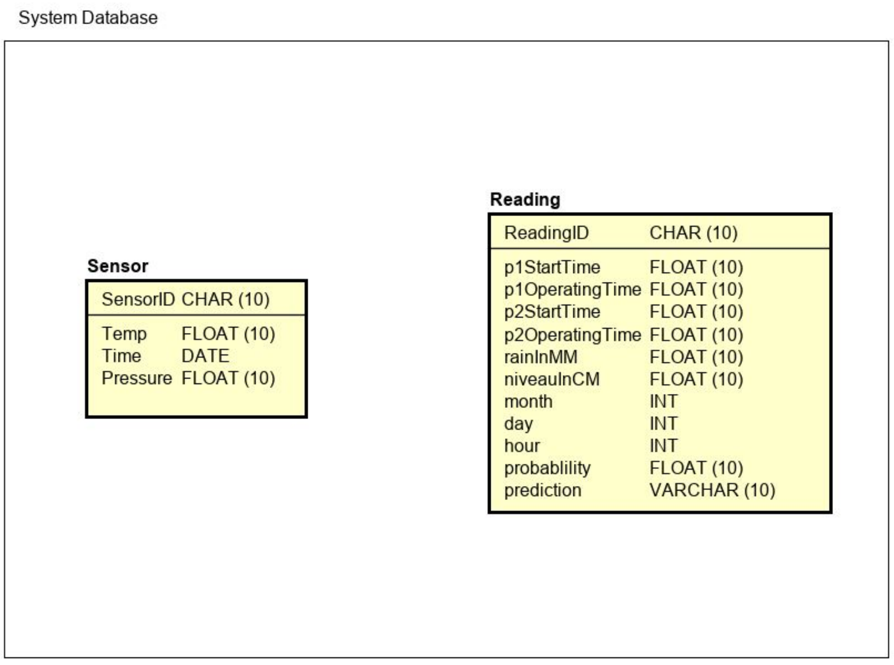


Figure - ER Diagram for the Database

PostgreSQL was chosen as the database query language to be used by the system. This decision was mostly fueled by familiarity with the relational databases and the PgAdmin software. For the system's use case, only two tables were needed without any relationship between them. Despite PostgreSQL being a relational database, it was decided to use the software regardless because the type of database would not change the result. The two tables are never going to have a relationship to connect them.

From the web application, any database access is handled via a .NET version of a framework called the Entity Framework Core, or EF Core. More information about the use of EF Core is detailed in the Entity Framework Core section of design. To sum up, EF Core is used to handle the data access from Blazor to the database.

UI Design

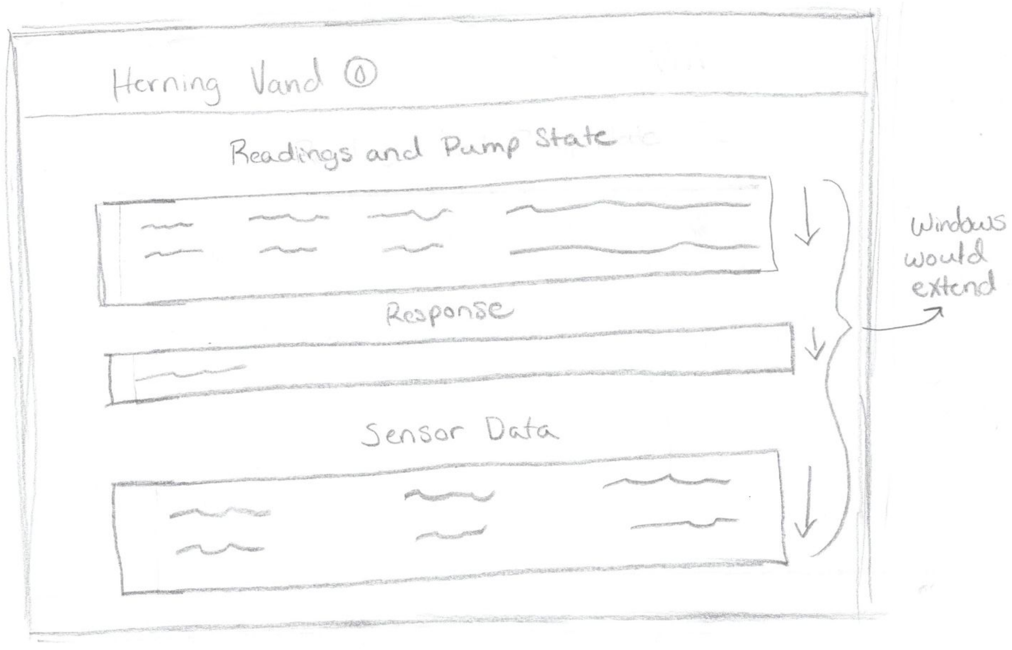


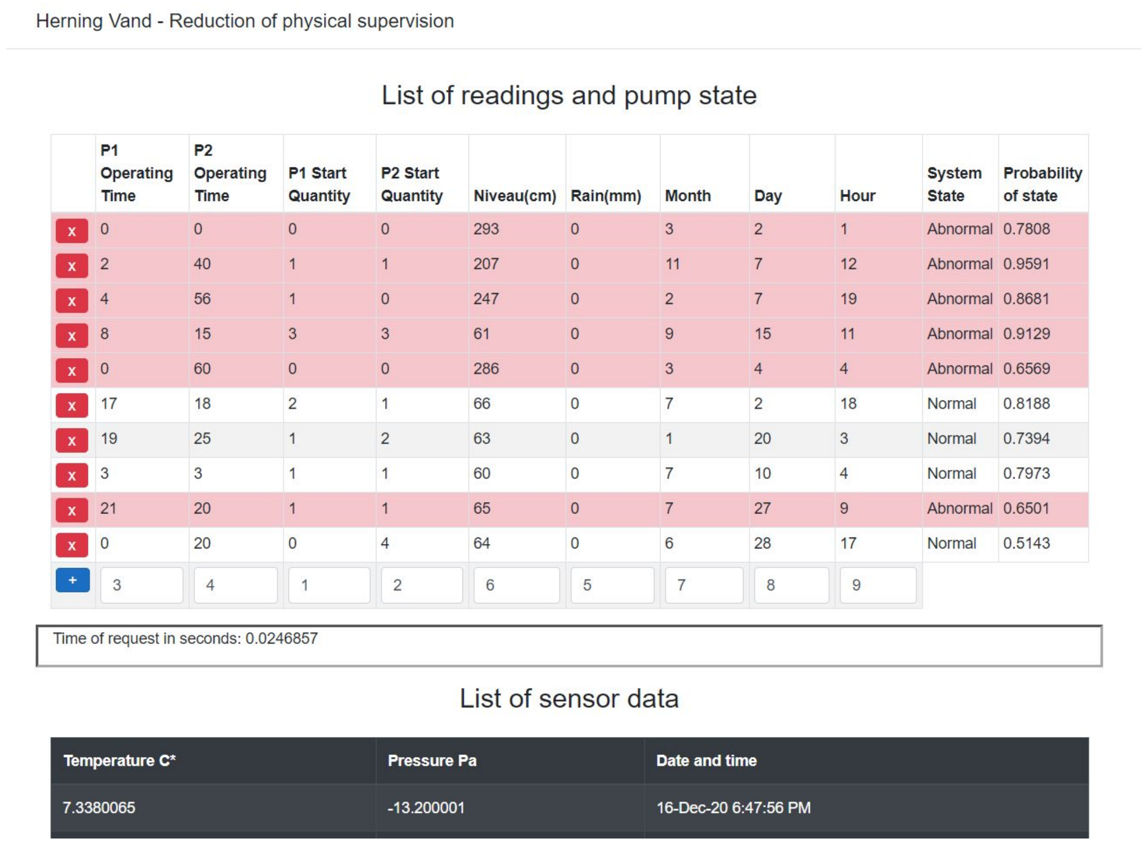
Figure - UI Design Sketch

The design of the system UI was created using Blazor combined with Blazor’s native Bootstrap implementation and a little CSS. Bootstrap is an open-source framework for web and mobile applications. On its own, Blazor doesn’t require Bootstrap however it does exist within the default project files if needed.

Despite Blazor not depending on Bootstrap, there were a couple of reasons to use it. One reason is that Blazor has it by default. The second reason is familiarity with the framework and third is responsiveness. While responsiveness is not a point to be considered for this project, it is a good design practice regardless to ensure the web application's appearance will remain consistent across browsers and screen sizes. Aside from Bootstrap, the UI design was mostly constructed using HTML elements such as tables and divs.

The layout of the UI itself underwent some design considerations as time went on. Given that Herning Vand is a water utility company and the proof of concept nature of the project, the design itself focuses on functionality and simplicity instead of flashiness. There were also concerns regarding whitespace, however, these were addressed in the final design.

The second design was condensed into one page. This was mainly done for simplicity. Again influenced by the proof of concept nature, condensing the design into one page took away some SOLID integrity in exchange for keeping the code in one place. Some parts of the code, such as the LoRaWAN bridge connection, did not need to take up a page of its own. Combining the pages also removed the issue with whitespace.



Herning Vand - Reduction of physical supervision

List of readings and pump state

	P1 Operating Time	P2 Operating Time	P1 Start Quantity	P2 Start Quantity	Niveau(cm)	Rain(mm)	Month	Day	Hour	System State	Probability of state
x	0	0	0	0	293	0	3	2	1	Abnormal	0.7808
x	2	40	1	1	207	0	11	7	12	Abnormal	0.9591
x	4	56	1	0	247	0	2	7	19	Abnormal	0.8681
x	8	15	3	3	61	0	9	15	11	Abnormal	0.9129
x	0	60	0	0	286	0	3	4	4	Abnormal	0.6569
x	17	18	2	1	66	0	7	2	18	Normal	0.8188
x	19	25	1	2	63	0	1	20	3	Normal	0.7394
x	3	3	1	1	60	0	7	10	4	Normal	0.7973
x	21	20	1	1	65	0	7	27	9	Abnormal	0.6501
x	0	20	0	4	64	0	6	28	17	Normal	0.5143
+	3	4	1	2	6	5	7	8	9		

Time of request in seconds: 0.0246857

List of sensor data

Temperature C*	Pressure Pa	Date and time
7.3380065	-13.200001	16-Dec-20 6:47:56 PM

Figure - Final UI Design

Module Interfaces

Due to the system having a multi-layered architecture, interfaces between the layers have to be established. The Blazor application will act as a client, when connecting to the LoRa tower or the python server.

The interface for the arduino to Blazor application will be a continuous socket connection. The decision was made based on how the LoRa server was set up by VIA, which is running a server socket that listens for potential clients *(This type of connection is also recommended by the LORIOT documentation)*.
The data transfer will occur every time the arduino sends a signal to the LoRa tower, which will be picked up by the client socket running in the Blazor app and then stored in the database.

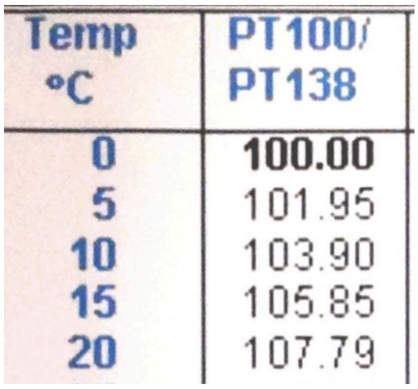
The interface for the Blazor application to the python module will be an API run by the python module. The API will expose a PUT request, that would take in data features, make a prediction on them and then send the prediction back to the Blazor application. The Blazor application then would pick up the response, and save the information in the database.

The choice for an API library in the python module will be Flask, since the team has had previous experience with it and is fast to set up.

The API is the choice of method because it would receive manual input by 1 user only, which does not require a constant connection between the modules.

The temperature sensor

In order to measure temperature a PT100 component was chosen. PT100 is a RTD (resistance temperature detector) sensor. That means that it will change it's resistance based on the temperature that influences it. After contacting Herning Vand a range of temperature to be measured was defined. Jan Ravn said that the temperature varies from 5°C to 15°C. To ensure the temperature will always be in range, the temperature sensor will be implemented to be able to measure from 0°C to 20°C - therefore the allowed error will be +/-5°C. Based on the resistance table (*Figure* bellow) the ΔR will be 7.79Ω.



Temp °C	PT100/ PT138
0	100.00
5	101.95
10	103.90
15	105.85
20	107.79

Figure - PT100 resistance table

In order to precisely detect the resistance changes, a wheatstone bridge construction was chosen. This will remove the offset of 100Ω which the PT100 has at 0°C.
The wheatstone bridge will be built using three 100Ω resistors and the PT100 as the fourth resistor (*Figure* bellow).

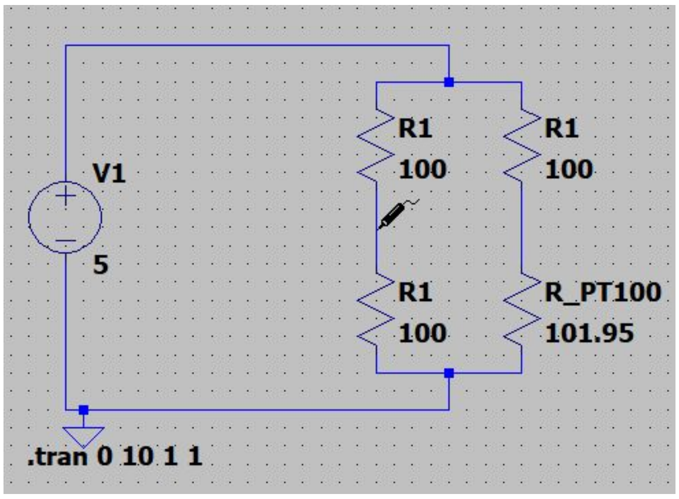


Figure - Wheatstone bridge set up for 5°C

The input voltage that will be used for this circuit will be 5V (supplied by arduino). In order to calculate the voltage between the two arms of the bridge the following formula was used:

$$V_{ab} = V_{in} \times R_{PT100} \div (R_1 + R_{PT100}) - V_{in} \times R_1 \div (R_1 + R_1)$$

$$R_1 \; - \; \textit{the resistance of the remaining resistors in the wheatstone bridge } (100\Omega)$$

The calculations were made for temperatures from 0°C until 20°C for every 5°C. The results of these calculations are presented in the *Table* below. Afterwards stimulations for every temperature were made using LTSpice and the results of calculations were confirmed with the results of the simulations.

Temperature (°C)	R_1 resistance	R_{PT100} (Ω)	V_{ab} (V)
0	100	100.00	0
5	100	101.95	$\approx$0.024
10	100	103.90	$\approx$0.048
15	100	105.85	$\approx$0.071
20	100	107.79	$\approx$0.094

Table - V_{ab} LTSpice Stimulatioins

As the results show the ΔV_{ab} barely reaches 0.01V. The Arduino 2560 that will be used in this project, converts on it's ADC pins voltage between 0-5V. In order to reach 5V at the maximum temperature (20°C) an amplifier will be used that will amplify 0.094V to reach 5V. In order to calculate the required gain, the following formula was used.

$$G = \frac{V_{out}}{V_{in}}$$

$$G = \frac{5}{0.094} \approx 53.19$$

Knowing that the gain will be constant for all temperatures,the output from the amplifier for each temperature can be calculated using the formula shown below. The results of the calculations are shown in *Table* below.

$$V_{out} = V_{in} \times G$$

Temperature (°C)	Gain	V_{ab} (V)	V_{out} (V)
0	53.19	0	0
5	53.19	$\approx$0.024	$\approx$1.277

10	53.19	≈0.048	≈2.553
15	53.19	≈0.071	≈3.776
20	53.19	≈0.094	≈5

Table - V_{out} Calculations

To amplify the V_{ab} an instrumental amplifier will be used. Such an amplifier is a type of a differential amplifier, which means that it amplifies the difference between two voltages. In this case the difference between V_a and V_b , which is calculated before, is being amplified. For building this circuit a value of 10kΩ resistance for the feedback resistors was chosen. The amplifier would reflect the following schematics.

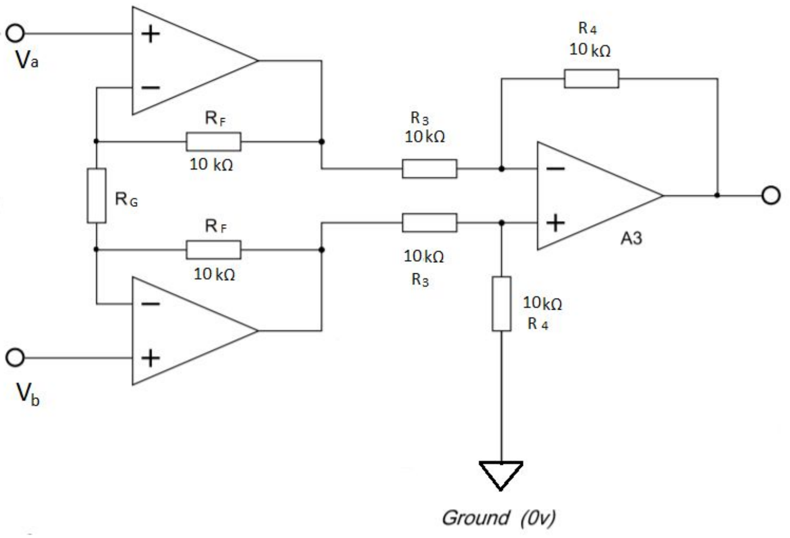


Figure - instrumentation amplifier

Since it is know that the gain should be 53.19 it is possible to calculate the R_G using the following equation:

$$G = \frac{R_4}{R_3} \times \left(1 + \frac{2 \times R_F}{R_G}\right) \Rightarrow R_G = \frac{2 \times R_F}{\frac{R_3}{R_4} \times G - 1}$$

$$R_G = \frac{2 \times 10000}{\frac{10000}{10000} \times 53.19 - 1} = \frac{20000}{52.19} = 383.22\Omega$$

Since a resistor with value 383.22Ω does not exist the calculated value will be rounded to 390Ω. When all resistor values are known it is possible to simulate the circuit and compare the calculated V_{out} values to the stimulated ones.

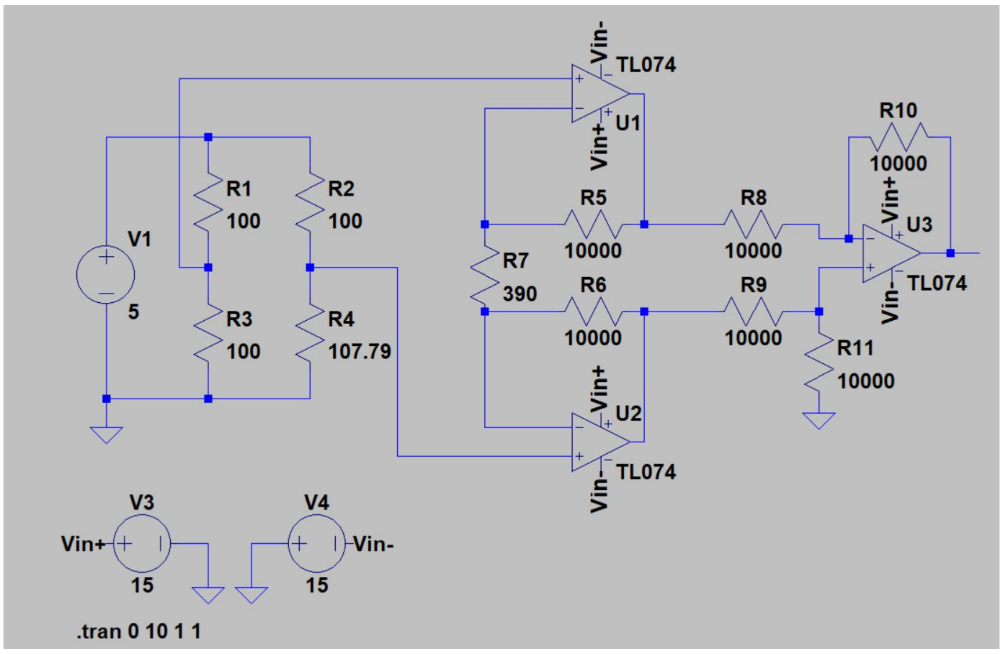


Figure - Instrumentation amplifier stimulation

Temperature (°C)	R_{PT100} (Ω)	V_{ab} (V)	Calculated V_{out} (V)	Simulated V_{out} (V)
0	100.00	0	0	≈0
5	101.95	≈0.024	≈1.277	≈1.262
10	103.90	≈0.048	≈2.553	≈2.499
15	105.85	≈0.071	≈3.776	≈3.713
20	107.79	≈0.094	≈5	≈4.899

Table - Calculated vs Simulated voltage output

The results of simulations differ slightly from the calculated voltage. That is caused by the slightly higher value of the R_G resistor. The simulations with slightly different results are still accurate enough, in order to proceed with the circuit build.

The pressure sensor

The pressure sensor that was picked for the project is SKU237545. This sensor needs a 5V power supply which can be supplied by the Arduino. The pressure range of this sensor is 0-1.2 MPa which is more than enough. This sensor can work between 0°C and 85°C, so it fits in the temperature range of the water. The sensor will output between 0.5-4.5 VDC which is a perfect voltage for the Arduino. No external voltage adjustments are needed, so the sensor will be connected directly to the Arduino module.

Figure - 1 - SKU237545 pressure sensor

Sensors application

The sensor application is implemented in the C programming language. It consists of three periodic tasks. The *Figure* below shows the actions taken by the application when it is running. The application is scheduled by a timer that allows the tasks to take measurements/send data repeatedly.

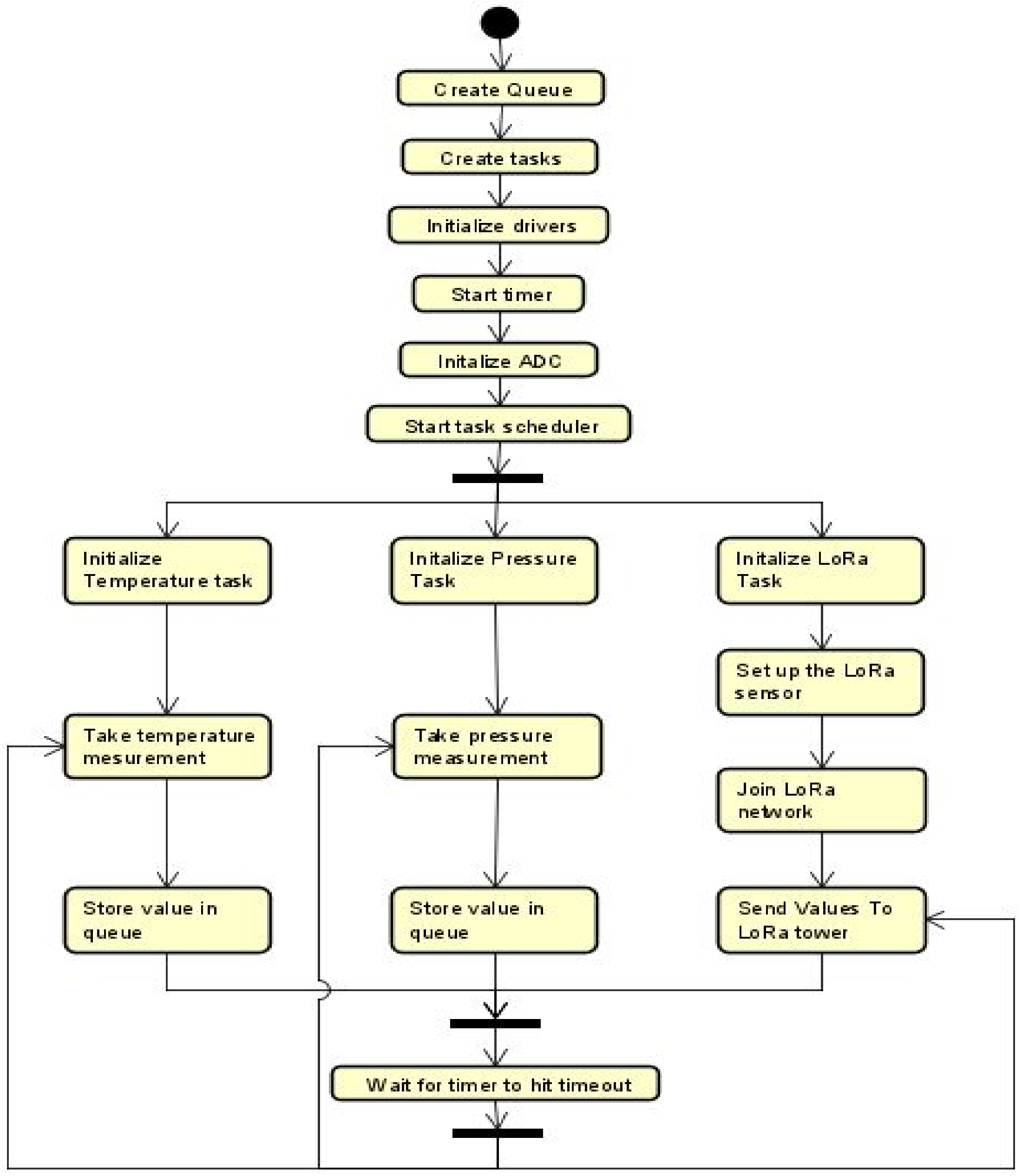


Figure - Sensor application Activity Diagram

The architecture of the application consists of three layers. The first layer is the Hardware Layer. This includes the PT100 (temperature sensor), SKU237545 (pressure sensor), RN2483 (LoRa transmitter) and the Arduino. The next layers are the drivers. The system has a driver for the pressure sensor, temperature sensor, LoRa transmitter, ADC module and the timer. The last layer is the application which uses the drivers that control the hardware. The *Figure* below shows all the layers of the sensor application.

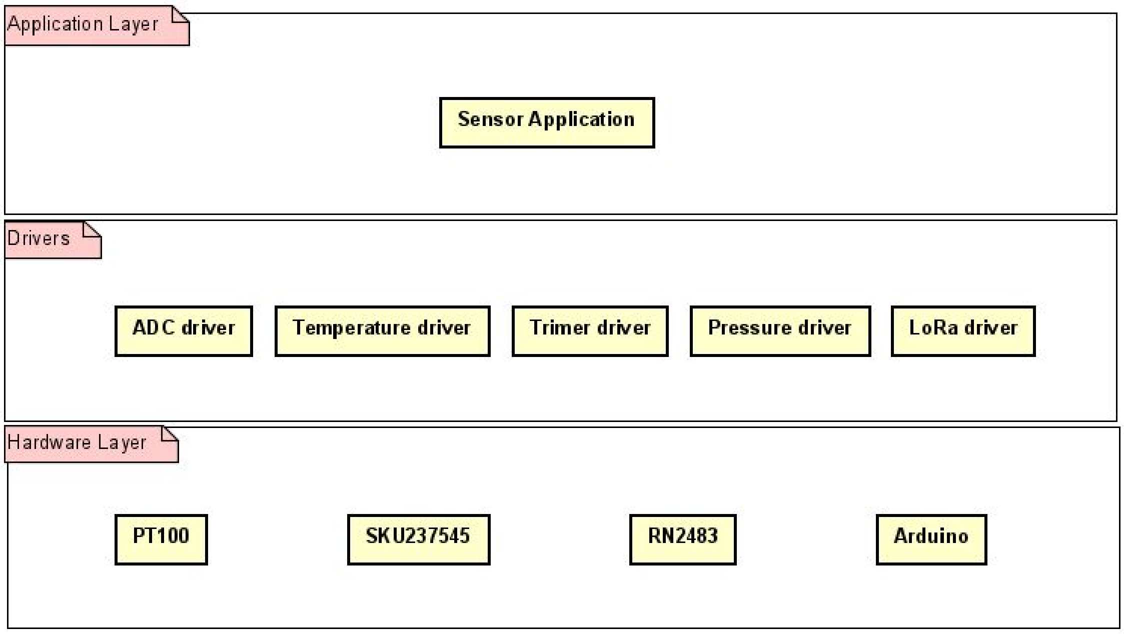


Figure - Sensor Application Architecture

The next diagram shows the structure of the code in depth. Each driver has a header file with the definitions of functions and a corresponding C file that implements the header. The “main.c” includes all the drivers. To the “ADC_driver.c” file access also have the “temperature.c” file and “pressure.c”, since they are using ADC to read the sensor's values. The “reading_struct.h” defines a data type that is used to store the measurements on a queue. Since “pressure.c” and “temperature.c” add measurements to the queue they have access to the header file, as well as the “LoRa_driver.c” which needs to read the values from the queue.

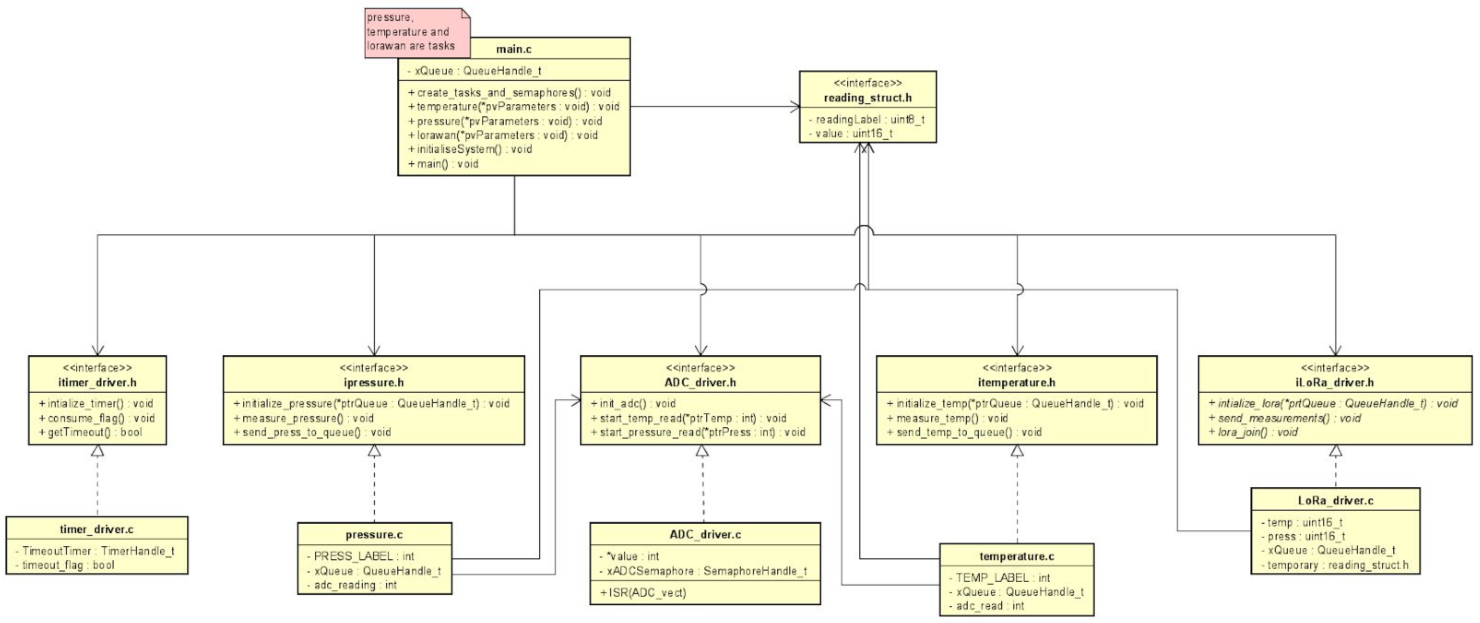

pressure, temperature and lorawan are tasks
main.c
- xQueue : QueueHandle_t
+ create_tasks_and_semaphores() : void
+ temperature(*pvParameters : void) : void
+ pressure(*pvParameters : void) : void
+ lorawan(*pvParameters : void) : void
+ initialiseSystem() : void
+ main() : void
<<interface>>
reading_struct.h
- readingLabel : uint8_t
- value : uint16_t
<<interface>>
itimer_driver.h
+ intialize_timer() : void
+ consume_flag() : void
+ getTimeout() : bool
<<interface>>
ipressure.h
+ initialize_pressure(*ptrQueue : QueueHandle_t) : void
+ measure_pressure() : void
+ send_press_to_queue() : void
<<interface>>
ADC_driver.h
+ init_adc() : void
+ start_temp_read(*ptrTemp : int) : void
+ start_pressure_read(*ptrPress : int) : void
<<interface>>
itemperature.h
+ initialize_temp(*ptrQueue : QueueHandle_t) : void
+ measure_temp() : void
+ send_temp_to_queue() : void
<<interface>>
iLoRa_driver.h
+ intialize_lora(*prtQueue : QueueHandle_t) : void
+ send_measurements() : void
+ lora_join() : void
timer_driver.c
- TimeoutTimer : TimerHandle_t
- timeout_flag : bool
pressure.c
- PRESS_LABEL : int
- xQueue : QueueHandle_t
- adc_reading : int
ADC_driver.c
- *value : int
- xADCSemaphore : SemaphoreHandle_t
+ ISR(ADC_vect)
temperature.c
- TEMP_LABEL : int
- xQueue : QueueHandle_t
- adc_read : int
LoRa_driver.c
- temp : uint16_t
- press : uint16_t
- xQueue : QueueHandle_t
- temporary : reading_struct.h

Figure - Sensors Application Class Diagram

The above diagram shows the files that were created specially for the Herning Vand project. This project is also using two external Libraries supplied by Ib Havn from VIA University College. These repositories are used for RTOS scheduling and controlling the RN2483 (LoRa) transceiver. All information about these repositories can be found in the following links:

- FreeRTOS: https://github.com/ihavn/VIA_FreeRTOS_AVRMEGA
- Drivers: https://github.com/ihavn/IoT_Semester_project

The following *Figure* depicts the process of collecting the temperature by the sensor and storing it in a queue ready for the LoRa task to send it further. As it can be seen the main file initializes the temperature task by making a call to “temperature.c”. Next the timer timeout is checked. The task will check the timeout until the returned value is true. Once the timeout is triggered, “main.c” file makes a call to “temperature.c” to take a measurement. Afterwards, the “ADC_driver.c” is used so the conversion is started. When the ADC finishes converting an interrupt is triggered, which allows the “temperature.c” to add the converted value to the queue. Lastly, the task is checking the timeout as long as the returned value is true. Then it starts the cycle again.

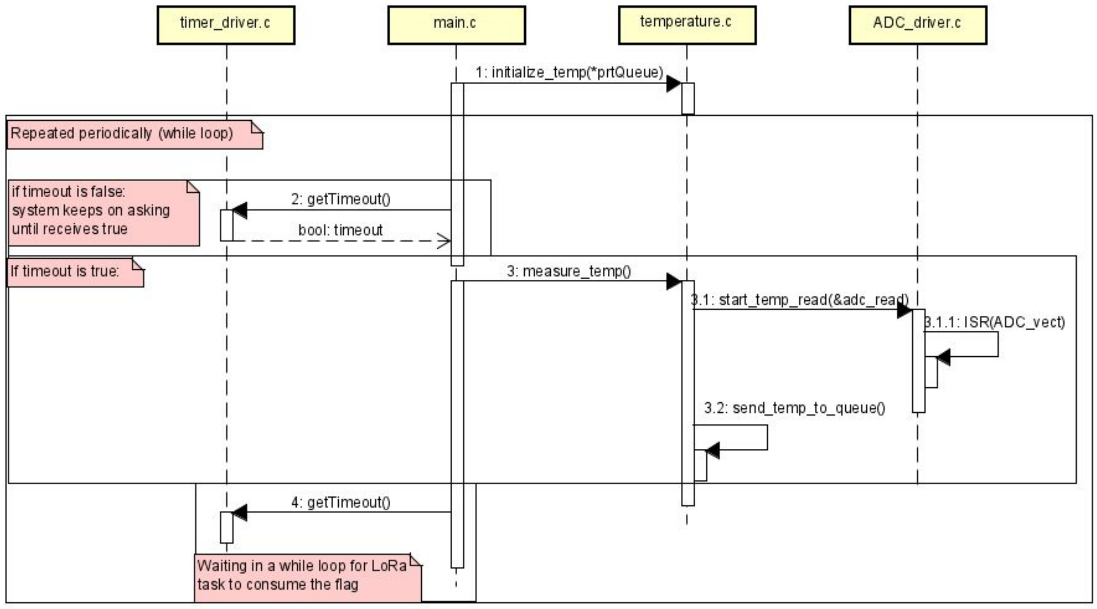

timer_driver.c
main.c
temperature.c
ADC_driver.c
1: initialize_temp(*prtQueue)
Repeated periodically (while loop)
if timeout is false: system keeps on asking until receives true
2: getTimeout()
bool: timeout
If timeout is true:
3: measure_temp()
3.1: start_temp_read(&adc_read)
3.1.1: ISR(ADC_vect)
3.2: send_temp_to_queue()
4: getTimeout()
Waiting in a while loop for LoRa task to consume the flag

Figure - Collecting Temperature Sequence Diagram

The final diagram represents the sequence of actions that are taken in order to send the collected measurements to the LoRa server. This diagram includes one external file “lora_driver.h” which is one of the externally implemented files that were mentioned before. The whole process is started by initializing the LoRa by making a call from “main.c” to “LoRa_driver.c”. Next a whole sequence of actions need to be done so get the transceiver ready to send the data. In order to do so, multiple calls are made to the externa; “lora_driver.h” file. The transceiver needs to be reseted, next all the buffers have to be flushed, factory reset performed, configured to eu868, set OTAA identity and enabled adaptive data rate. Finally the transceiver is ready to join the network, which is done by calling “lora_join()” in the local “LoRa_driver.c” file. This envoces the corresponding method in “lora_driver.h”. After all the preparation is complete, when the timeout has occurred, “send_measurements()” call will be made, which will upload the data to the server. Lastly a call to “timer.c” is made to indicate that the tasks for this cycle have completed their work. Then the cycle is started again.

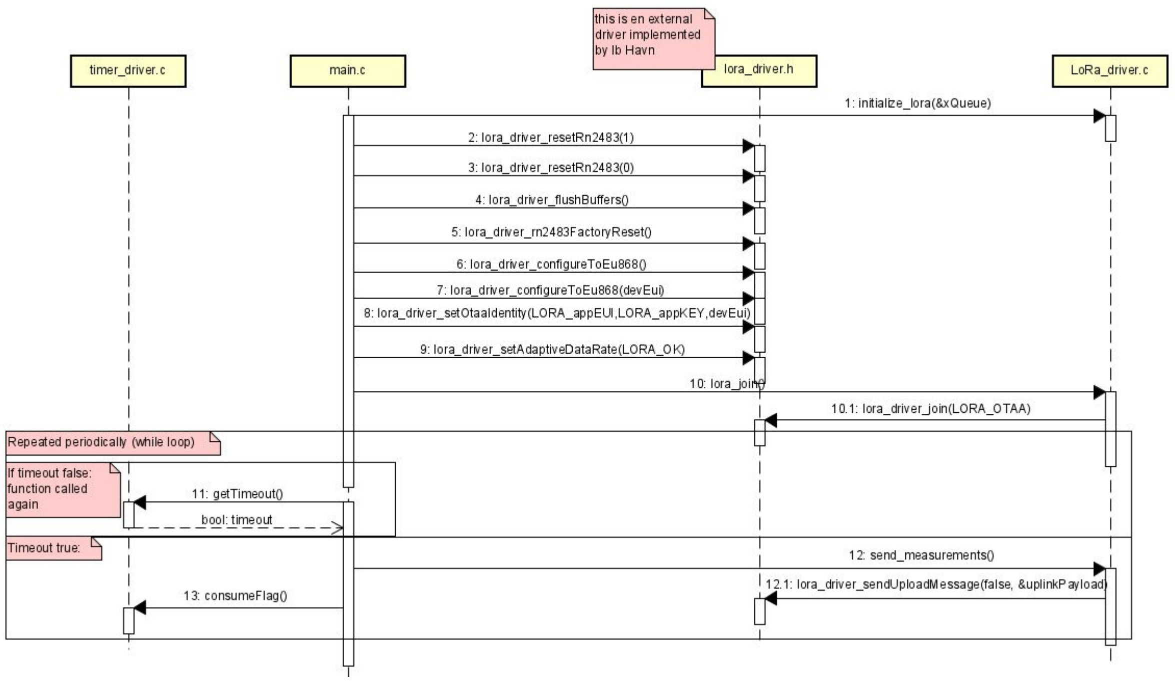
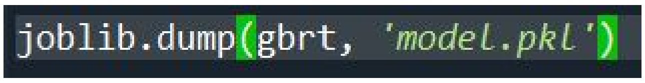

this is en external driver implemented by Ib Havn
timer_driver.c
main.c
lora_driver.h
LoRa_driver.c
1: initialize_lora(&xQueue)
2: lora_driver_resetRn2483(1)
3: lora_driver_resetRn2483(0)
4: lora_driver_flushBuffers()
5: lora_driver_rn2483FactoryReset()
6: lora_driver_configureToEu868()
7: lora_driver_configureToEu868(devEui)
8: lora_driver_setOtaaIdentity(LORA_appEUI,LORA_appKEY,devEui)
9: lora_driver_setAdaptiveDataRate(LORA_OK)
10: lora_join()
10.1: lora_driver_join(LORA_OTAA)
Repeated periodically (while loop)
If timeout false: function called again
11: getTimeout()
bool: timeout
Timeout true:
12: send_measurements()
12.1: lora_driver_sendUploadMessage(false, &uplinkPayload)
13: consumeFlag()

Figure - Sending Data Sequence Diagram

Implementation

Python Module

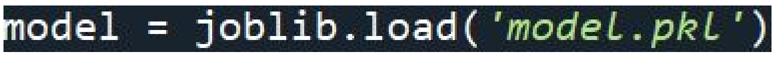
The implementation of the python module is very short and straight to the point. First the algorithm is packed into a pickle file. The pickle format uses a python library to serialize objects into files, which can be deserialized and used later. And so the trained algorithm model is “pickled” using the joblib library.



```
joblib.dump(gbrt, 'model.pkl')
```

Figure - Example of the method that pickles the model

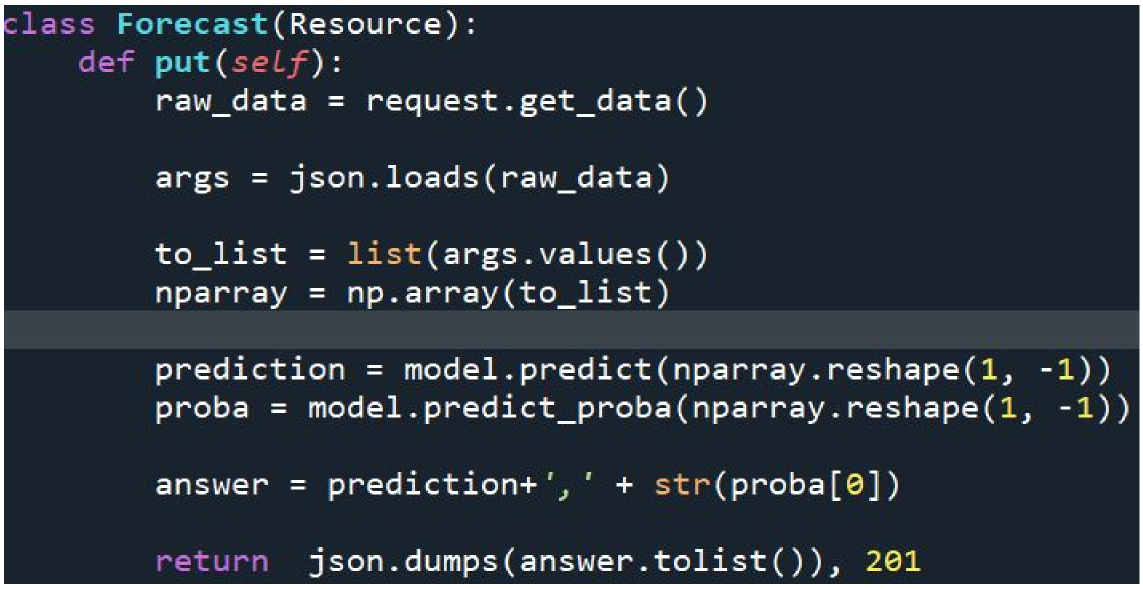
The same model then is “unpickled” into the api code, in order to be used by the application’s requests.



```
model = joblib.load('model.pkl')
```

Figure - Example of deserialization of the model in the api

The API set up in the python module is very simplistic, since it is expected that the Blazor application delivers the exact format of the data features.



```
class Forecast(Resource):
    def put(self):
        raw_data = request.get_data()

        args = json.loads(raw_data)

        to_list = list(args.values())
        nparray = np.array(to_list)

        prediction = model.predict(nparray.reshape(1, -1))
        proba = model.predict_proba(nparray.reshape(1, -1))

        answer = prediction+',' + str(proba[0])

        return  json.dumps(answer.tolist()), 201
```

Figure - PUT request in using Flask

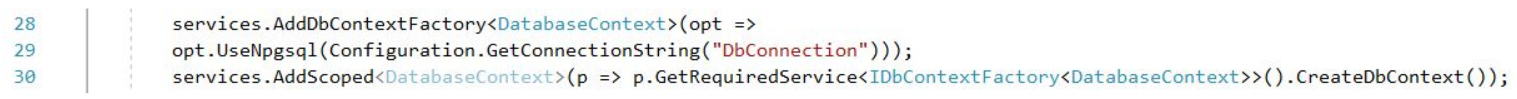
When a request is received, the body is deserialized into a dictionary object. After that the dictionary object gets its values parsed into a list, which then gets sent to the algorithm for predictions. There are 2 methods called on the algorithm model - the label of the prediction and the probability. We ask for probability, because in some cases, the model has to make close calls (45:55) and we want to be able to deliver the accuracy of the prediction as well. After the prediction phase has executed, the result is saved into a string, separated by a coma. That string then is being serialized and sent back to the Blazor application with the response code of 201 (Created).

Web Application

The web application itself is built using Blazor Server. The database connection is established mostly through the use of EF Core. The various interfaces to other modules were established through a variety of means. Websockets were used for the LoRaWaN connection and an API was exposed from the Python Module. Details on these connections are explained below and the Python Module is explained in the previous section.

Database Connection

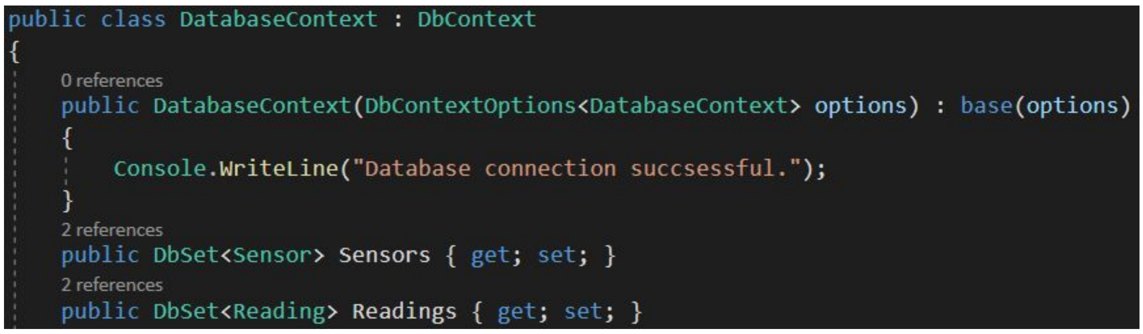
The connection to the database is established using Entity Framework Core, using DbContextFactory as the data access. Normally, EF Core makes use of a class that interacts with the database called the DbContext. However, as mentioned in the Design section of this documentation, Blazor Server operates in such a way that causes the DbContext's lack of thread safety and inability to run concurrently an issue. This can become problematic for system scalability in the case that Herning Vand decides that they want multiple users or components to have the ability to access the database in the future. To work around this problem, it is typically suggested to implement EF Core's DbContext using the factory class, DbContextFactory. This class has thankfully been included in ASP.NET Core as of recent updates.



```
services.AddDbContextFactory<DatabaseContext>(opt =>
opt.UseNpgsql(Configuration.GetConnectionString("DbConnection")));
services.AddScoped<DatabaseContext>(p => p.GetRequiredService<IDbContextFactory<DatabaseContext>>().CreateDbContext());
```

Figure - DbContextFactory Class in Startup.cs

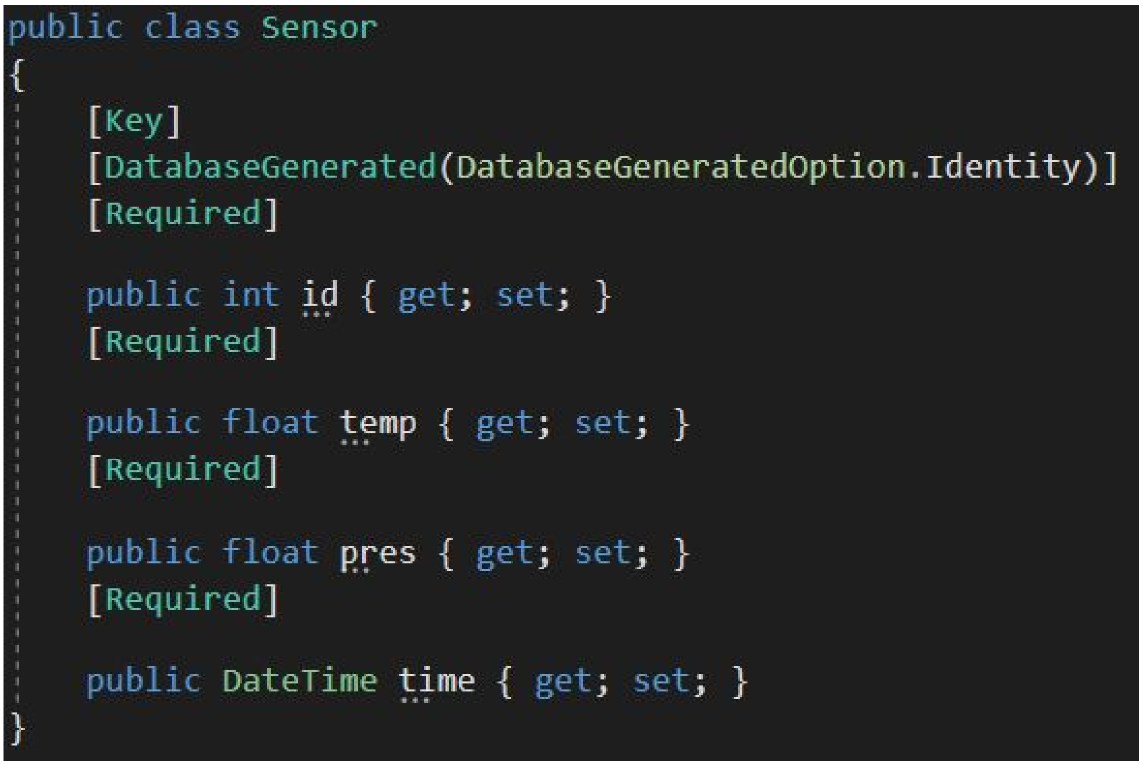
The DbContextFactory class makes requests to the database by creating a DbContext that holds the connection string. This context is created on every request, and after the request has finished, the context is disposed of. This is done to prevent multiple requests to the same context.



```
public class DatabaseContext : DbContext
{
    0 references
    public DatabaseContext(DbContextOptions<DatabaseContext> options) : base(options)
    {
        Console.WriteLine("Database connection succsessful.");
    }
    2 references
    public DbSet<Sensor> Sensors { get; set; }
    2 references
    public DbSet<Reading> Readings { get; set; }
```

Figure - DbContext Class

Following the example, you can see that the DatabaseContext class implements DbContext, passing options in its base.
Since Entity Framework Core makes use of Migrations to modify and update its database, the tables need to be defined in the same class too. In the picture it can be seen that that class defines two unrelated tables - sensors and readings.

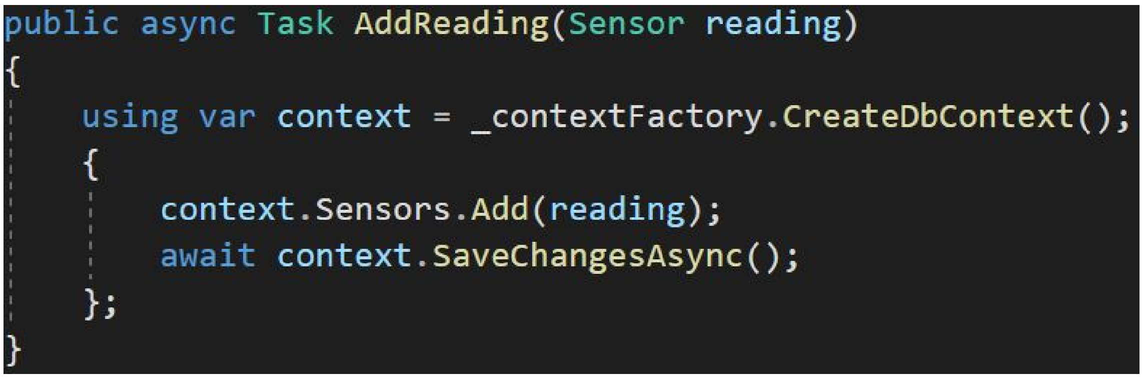


```
public class Sensor
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    [Required]

    public int id { get; set; }
    [Required]

    public float temp { get; set; }
    [Required]

    public float pres { get; set; }
    [Required]

    public DateTime time { get; set; }
}
```

Figure - Example of sensors table data attributes

The framework fits the database tables based on predefined classes as the one you can see in the picture. The class makes use of annotations to specify keys, not nulls, etc. Since we only have two sensors for temperature and pressure, the table will record the two respective values plus the date and the time of the arduino signal. An “id” is added as an auto generated integer by the database.

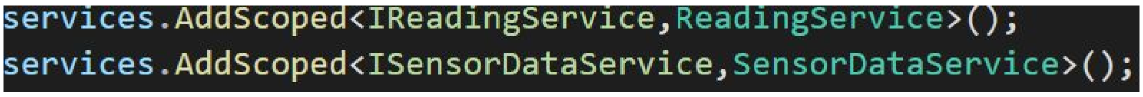
The way entity framework core makes requests to the database is very abstract, which is one of the reasons it was chosen in the design.



```
public async Task AddReading(Sensor reading)
{
    using var context = _contextFactory.CreateDbContext();
    {
        context.Sensors.Add(reading);
        await context.SaveChangesAsync();
    };
}
```

Figure - Example of adding data to the sensors table

After the sensor information has been retrieved and packed into an object, it can be easily added to the database table with this method. The method calls the context factory to build a context object for the request. The sensor object is then added to the database and the changes are saved before the context is destroyed. This method is part of a service class that makes database calls to the sensors table. This class is added to the application’s services as a scoped service. Scoped services are the same objects within a request, but different between multiple requests.

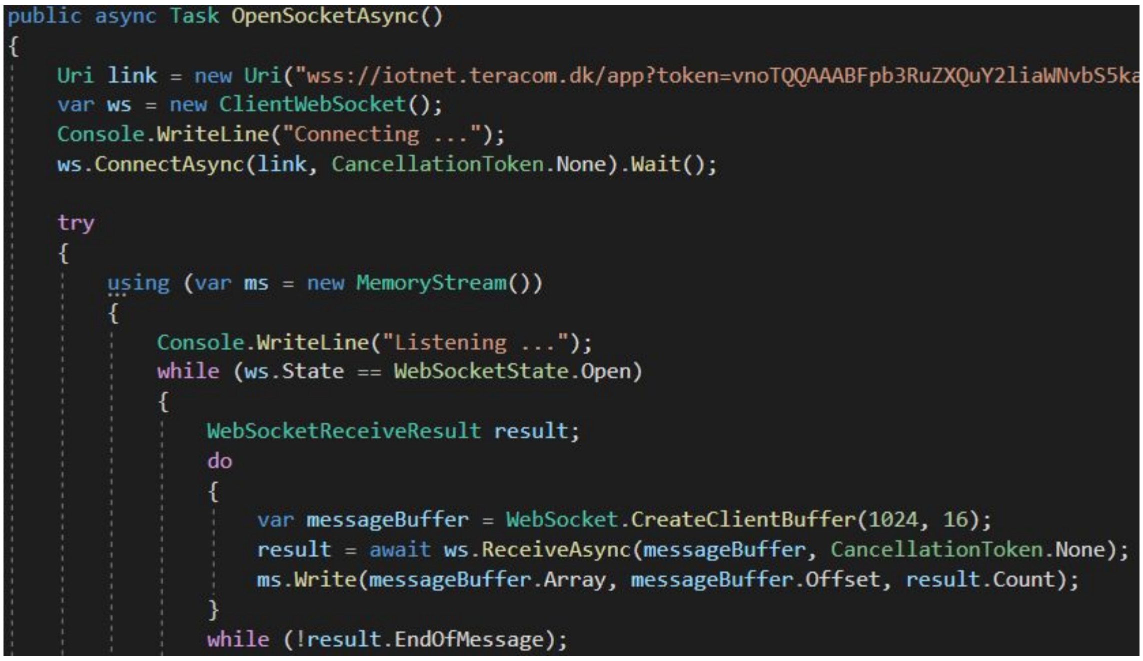


```
services.AddScoped<IReadingService,ReadingService>();
services.AddScoped<ISensorDataService,SensorDataService>();
```

Figure - Example of database interfaces being added as scoped services

LoRa Socket Interface

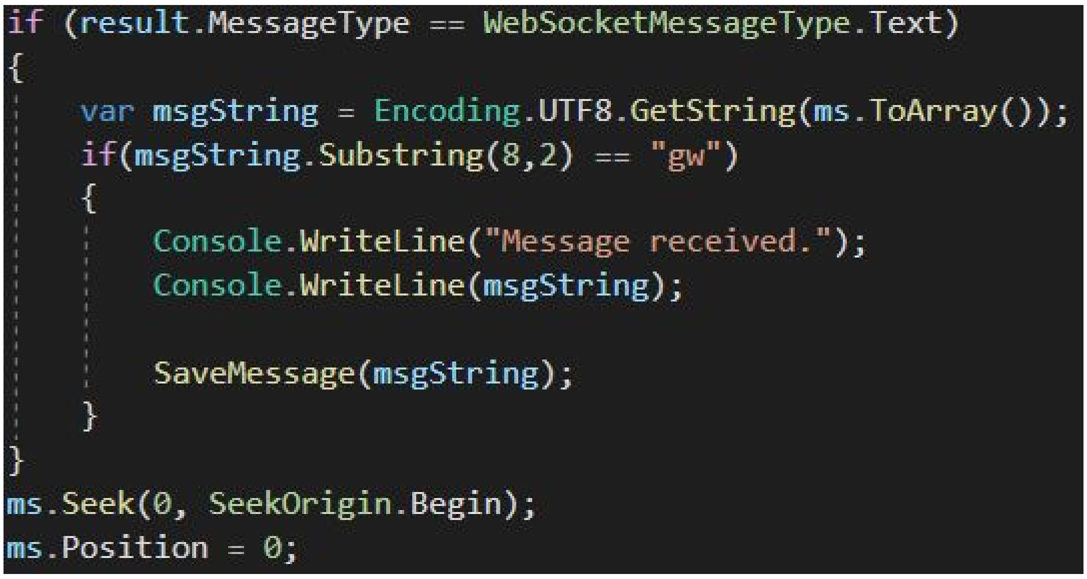
The socket interface is pretty straightforward. The class is added as a singleton to the application's services in order to ensure that only one socket is listening all the time.



```
public async Task OpenSocketAsync()
{
    Uri link = new Uri("wss://iotnet.teracom.dk/app?token=vnoTQQAAABFpb3RuZXQuY2liaWNvbS5ka
    var ws = new ClientWebSocket();
    Console.WriteLine("Connecting ...");
    ws.ConnectAsync(link, CancellationToken.None).Wait();

    try
    {
        using (var ms = new MemoryStream())
        {
            Console.WriteLine("Listening ...");
            while (ws.State == WebSocketState.Open)
            {
                WebSocketReceiveResult result;
                do
                {
                    var messageBuffer = WebSocket.CreateClientBuffer(1024, 16);
                    result = await ws.ReceiveAsync(messageBuffer, CancellationToken.None);
                    ms.Write(messageBuffer.Array, messageBuffer.Offset, result.Count);
                }
                while (!result.EndOfMessage);
```

Figure - Socket Client

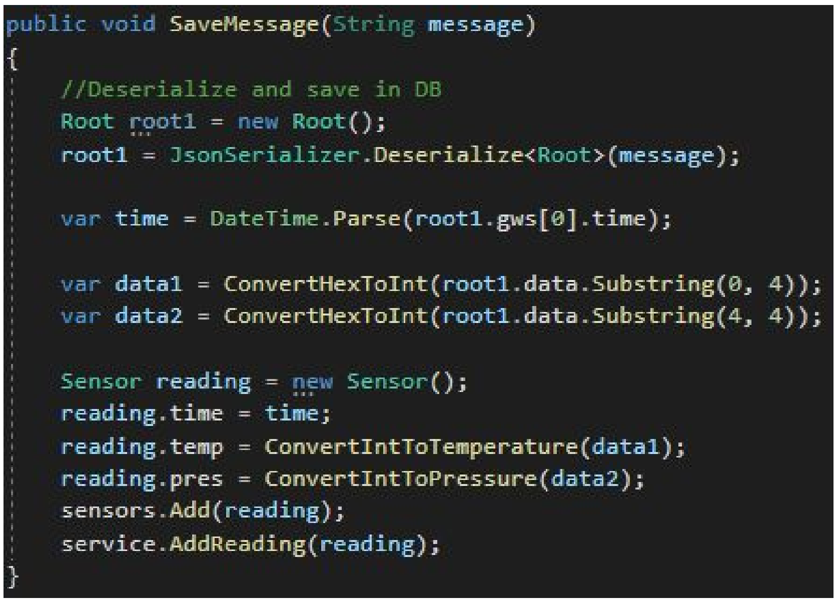
The client opens up a connection to the LoRa server with a predefined token, then starts listening for incoming messages. After it receives a message, it loops through the message's byte array until it reaches the end of it.
The client constantly loops while the socket connection is open and when the connection gets closed, it opens up a new connection (this is done instead of pinging in order to simplify the connection to the LoRa server by only listening for signals and not pinging/ sending data back.



```
if (result.MessageType == WebSocketMessageType.Text)
{
    var msgString = Encoding.UTF8.GetString(ms.ToArray());
    if(msgString.Substring(8,2) == "gw")
    {
        Console.WriteLine("Message received.");
        Console.WriteLine(msgString);

        SaveMessage(msgString);
    }
}
ms.Seek(0, SeekOrigin.Begin);
ms.Position = 0;
```

Figure - Example of a deserialization of the message

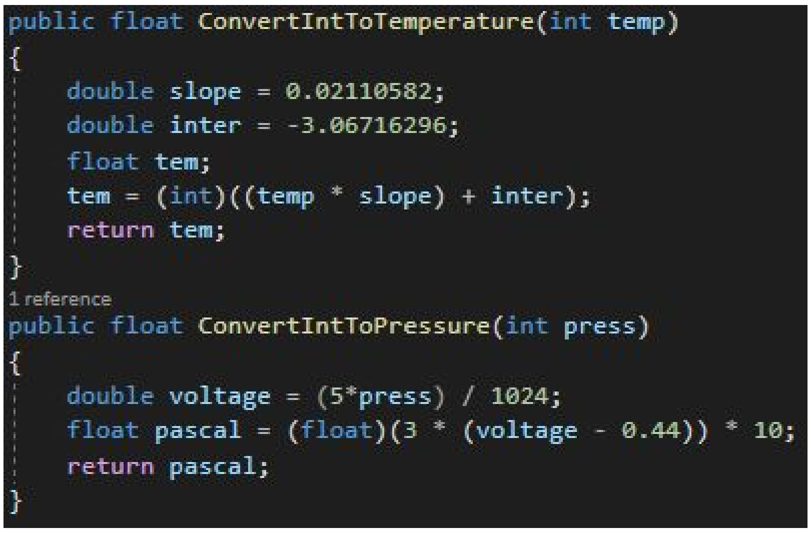
If the message has text, the text gets deserialized into UTF8 format. After the deserialization process is done, a check is made in order to conclude whether the message is of the correct type. It checks for messages of type “gw” because it holds both the data and the time of the signal, which are our sensor’s table attributes. Finally, when it is sure that the received message is of the correct type, it passes the string to a save method, which splits the string into datetime and integer variables, deserializes the ints into floats, creates a sensor object and passes it to the sensor service class in order to save the information into the database.



```
public void SaveMessage(String message)
{
    //Deserialize and save in DB
    Root root1 = new Root();
    root1 = JsonSerializer.Deserialize<Root>(message);

    var time = DateTime.Parse(root1.gws[0].time);

    var data1 = ConvertHexToInt(root1.data.Substring(0, 4));
    var data2 = ConvertHexToInt(root1.data.Substring(4, 4));

    Sensor reading = new Sensor();
    reading.time = time;
    reading.temp = ConvertIntToTemperature(data1);
    reading.pres = ConvertIntToPressure(data2);
    sensors.Add(reading);
    service.AddReading(reading);
}
```

Figure - Example of saving the message in the database

The data received from the arduino is in the form of two integers. These integers are serialized floats, which need to be deserialized accordingly. The socket class implements two methods for deserialization of these integers into floats. The implementation of these methods shows the functions used for serialization.



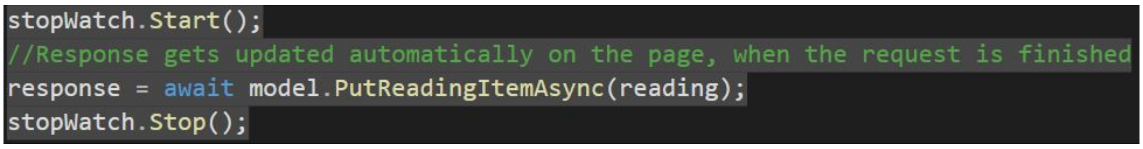
```
public float ConvertIntToTemperature(int temp)
{
    double slope = 0.02110582;
    double inter = -3.06716296;
    float tem;
    tem = (int)((temp * slope) + inter);
    return tem;
}
1 reference
public float ConvertIntToPressure(int press)
{
    double voltage = (5*press) / 1024;
    float pascal = (float)(3 * (voltage - 0.44)) * 10;
    return pascal;
}
```

Figure - Example of deserialization of ints into floats

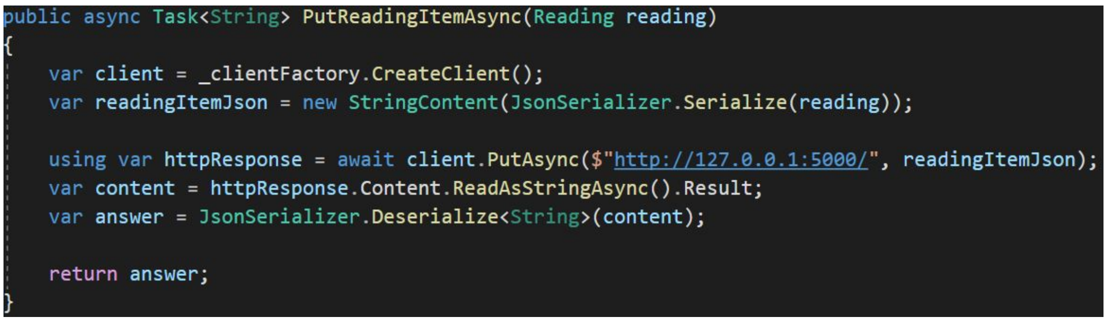
Python Module Interface

Since the python API implements a singular PUT request, the interface that talks to it won't be substantial in size too. Before the request is made, a user input is taken from the web application. That input is then added to a reading object, after that it gets serialized into a json string and sent to the python module. A stopwatch object is used to record the time of the requests.

Figure - Example of a put request in the Blazor page

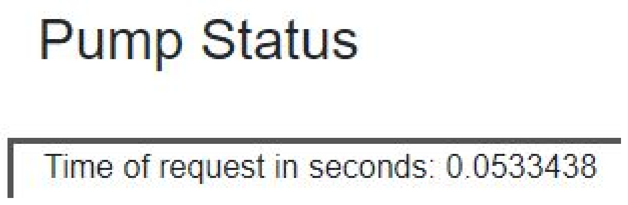


```
stopWatch.Start();
//Response gets updated automatically on the page, when the request is finished
response = await model.PutReadingItemAsync(reading);
stopWatch.Stop();
```



```
public async Task<String> PutReadingItemAsync(Reading reading)
{
    var client = _clientFactory.CreateClient();
    var readingItemJson = new StringContent(JsonSerializer.Serialize(reading));

    using var httpResponse = await client.PutAsync($"http://127.0.0.1:5000/", readingItemJson);
    var content = httpResponse.Content.ReadAsStringAsync().Result;
    var answer = JsonSerializer.Deserialize<String>(content);

    return answer;
}
```

Figure: Example of the Blazor API Client class's PUT method

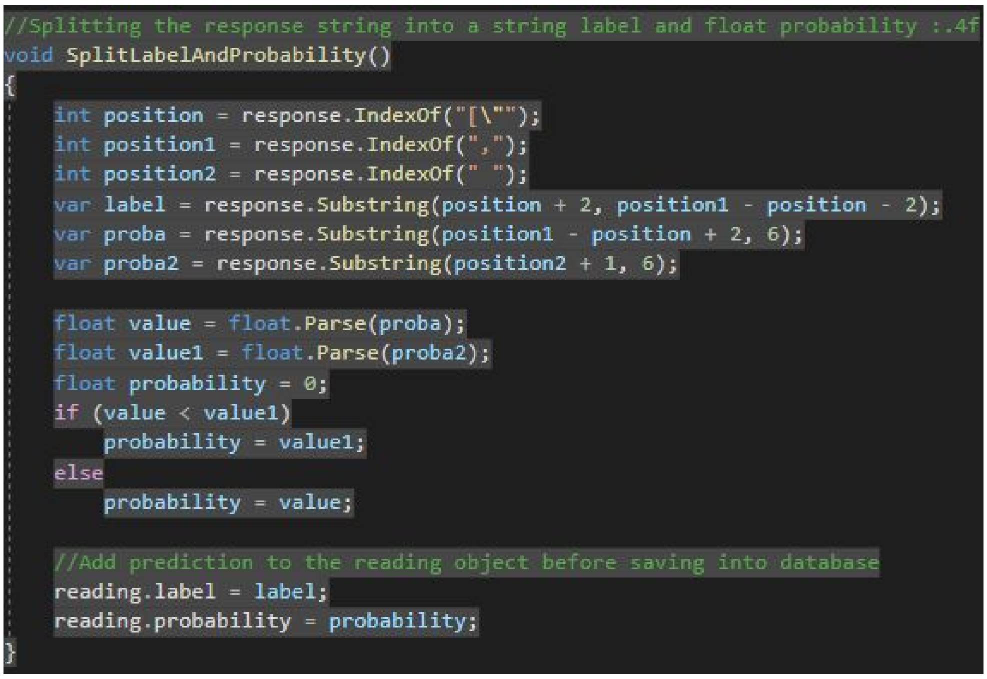


Pump Status

Time of request in seconds: 0.0533438

Figure: Response time of the api client request

The application uses an API client to make requests to the python server. That client class is added as a singleton in the Blazor server services in order to ensure that a single instance of the client is running at all time. Making the client a singleton service allows us to inject it into the web application and make requests directly from the frontend. After receiving a response from the python server, the client class passes the response string back to the web application, where it gets split into a label string and a probability float. The two values are then added to the object, initially sent with the request, and the object is then saved to the readings table in the database, through service interface dependency injection.



```
//Splitting the response string into a string label and float probability :.4f
void SplitLabelAndProbability()
{
    int position = response.IndexOf("[\"");
    int position1 = response.IndexOf(",");
    int position2 = response.IndexOf(" ");
    var label = response.Substring(position + 2, position1 - position - 2);
    var proba = response.Substring(position1 - position + 2, 6);
    var proba2 = response.Substring(position2 + 1, 6);

    float value = float.Parse(proba);
    float value1 = float.Parse(proba2);
    float probability = 0;
    if (value < value1)
        probability = value1;
    else
        probability = value;

    //Add prediction to the reading object before saving into database
    reading.label = label;
    reading.probability = probability;
}
```

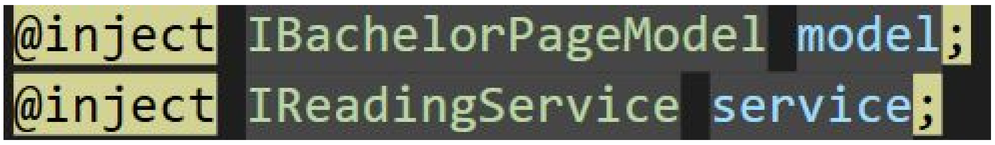
Figure : Splitting Response Method

User Interface : Blazor

The user interface of the web application consists of a single view. As mentioned in the Design section of this report, the UI was kept as a single page because the amount of functionality and whitespace per page was not justifiable. The layout itself was mostly created using a combination of HTML and Bootstrap. Most data in the razor page is organized into HTML tables.

It is important to note that there is a difference between the razor pages that are used in MVC and razor pages that are used in Blazor. On a very fundamental level they are the same, both utilizing the HTML and C# pattern, but it is how they are set up and used within their respective frameworks that makes the difference. In the general Blazor framework, a component is presented using a regular Razor markup but the component has a .razor extension according to(Microsoft, 2020. *Introduction to ASP.NET Core Blazor*) instead of a cshtml or cshtml.cs extension like in MVC.

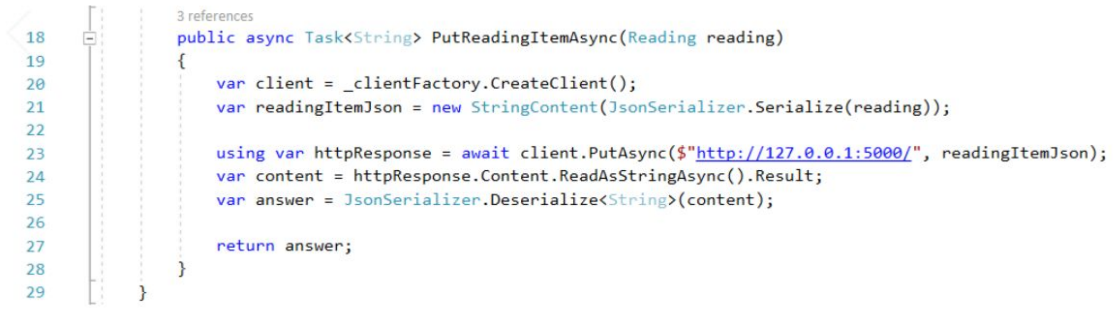
As a result of the delimitation that prevents the use of Herning Vand's own systems, the view implements the "Readings and Pump State" table using input fields that act as a stand-in for automated input from SCADA. The index razor also displays sensor data that is retrieved from the database and displayed.



```
@inject IBachelorPageModel model;
@inject IReadingService service;
```

Figure : Dependency Injection of Client Model and Database Service

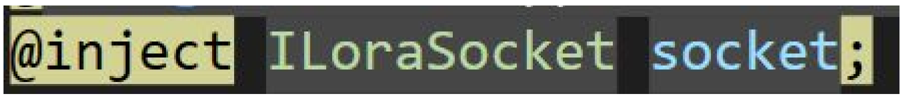
The index razor page makes use of dependency injection which allows access to various service classes and methods within Blazor Server. Starting with the "Readings and Pump State" section. When the user enters information into the UI and clicks the "+" button, the respective methods are called from the C# code in the razor page's @code section, namely the Request() method. This method creates a new Reading object from the user input and then the PutReadingItemAsync method is called with the argument of the Reading object. This method sends the Reading object as a PUT HTTP request to the Python server. The method awaits until the Python server sends its response. The response is then sent back to the Reading section of the razor page and displayed within the appropriate window. The response is also saved in the PostgreSQL database via the AddReading async method.



```
        3 references
        public async Task<String> PutReadingItemAsync(Reading reading)
        {
            var client = _clientFactory.CreateClient();
            var readingItemJson = new StringContent(JsonSerializer.Serialize(reading));

            using var httpResponse = await client.PutAsync($"http://127.0.0.1:5000/", readingItemJson);
            var content = httpResponse.Content.ReadAsStringAsync().Result;
            var answer = JsonSerializer.Deserialize<String>(content);

            return answer;
        }
    }
```

Figure : PutReadingItemAsync method

The Sensor data behaves in a similar way sans the user input, however the razor listens in on the socket connection using the functionality that was injected into the page. When sensor data is received from the Arduino it is properly parsed via the respective LoRa Socket methods and saved to the database. Once these actions complete the result is displayed in the "Sensor Data" section of the web page. Keep in mind that the socket interface is added as a singleton to the Blazor Server services in the Startup.cs and will remain the same object throughout the lifecycle of the application.



```
@inject ILoraSocket socket;
```

Figure : Dependency Injection of Socket Interface

The Temperature Sensor

In order to build the wheatstone bridge with the instrumentation amplifier multiple components were used. The full list can be found in Appendices in the *"circuit_components.pdf"* file.

Firstly the wheatstone bridge was built. Three 100Ω resistors were used. Not all resistors had the perfect value of 100Ω therefore the true value is shown in the table below:

Name	Theoretical Resistance (Ω)	Actual Resistance (Ω)
R_1	100	96.7
R_2	100	99,5
R_3	100	99,3
R_{PT100}	varying	varying

Table - actual resistances (Wheatstone Bridge)

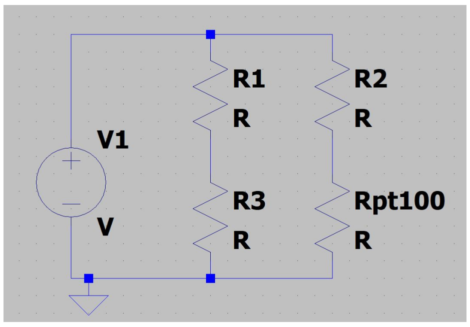


Figure - wheatstone bridge schematics

With such resistor values, the Wheatstone Bridge arm with resistors R_1 and R_3 has the voltage 2.39V. The resistance on the second arm was changing depending on R_{PT100} therefore it was measured at different temperatures, shortly after the voltage was checked. The results are shown in the table below.

Temperature (°C)	R_{PT100} (Ω)	V_{out} (V)
0	100.6	2.39
5	102.6	2.43
10	104.7	2.48
15	106.7	2.53
20	108.5	2.56

Table - actual voltage and resistance depending on temperature

Since the values differ from the results established in the design section the calculations had to be adjusted. The maximum V_{ab} at 20°C will be 0.1V. Therefore the actual gain and R_G are calculated below.

$$G = \frac{5}{0.1} \approx 50$$

$$R_G = \frac{2\times10000}{\frac{10000}{10000}\times50-1} = \frac{20000}{49} = 408.16\Omega$$

In the second part of the circuit, which is the instrumentation amplifiers, the calculated value of R_G did not bring the required result of amplifying to 5V. Another resistor had to be chosen instead. By testing different resistances, the value of 375V was chosen. Below are shown the actual resistor values and the schematics of the instrumentation amplifier.

Name	Theoretical Resistance (Ω)	Actual Resistance (Ω)
R_5	10 000	9 920
R_6	10 000	9 930
R_7	375	373

R_8	10 000	9 950
R_9	10 000	9 920
R_{10}	10 000	9 910
R_{11}	10 000	9 910

Table - actual resistances (Instrumentation Amplifier)

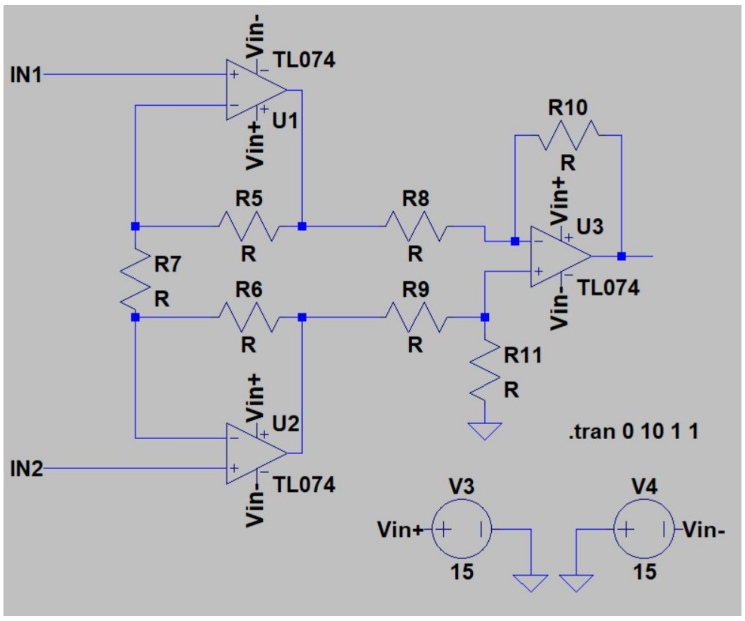
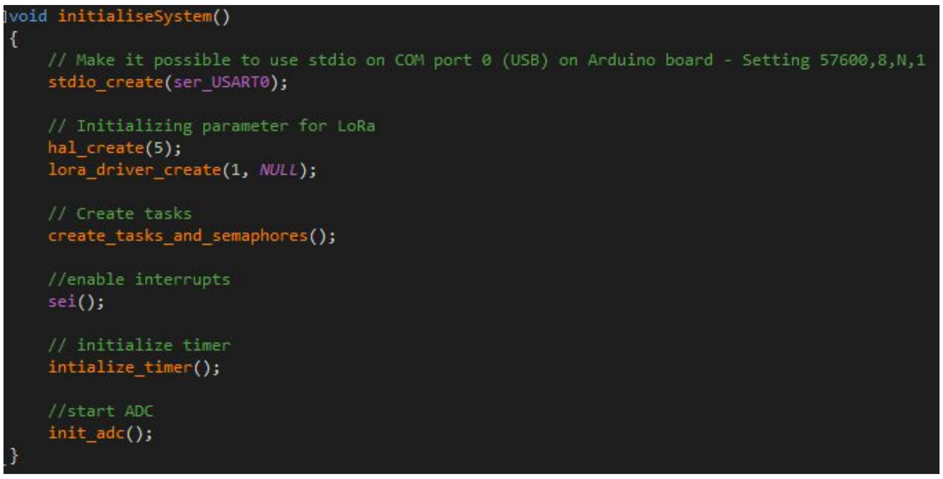


Figure - Instrumentation Amplifier schematics

The resistance of 373Ω was achieved by soldering two resistors of value 330Ω and 47Ω in a parallel connection. Such a setup was able to amplify the V_{ab} as it was required. At the temperature of 20°C the output was 5.1V and at 0°C it was 0.48V. This voltage is connected to the Arduino to pin PK0, which is later used by the sensor application.

Sensor Application

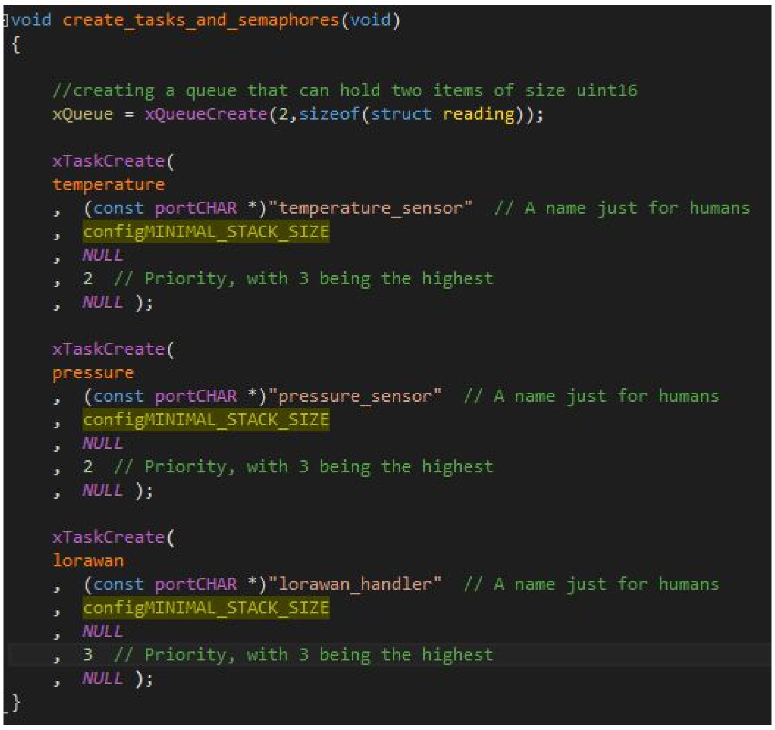
In order to start the sensor application the following initialization is completed: setting up com port, initializing the LoRa transceiver, creating tasks and semaphores, enabling interrupts and semaphores, initializing timer and initializing ADC.



```
void initialiseSystem()
{
    // Make it possible to use stdio on COM port 0 (USB) on Arduino board - Setting 57600,8,N,1
    stdio_create(ser_USART0);

    // Initializing parameter for LoRa
    hal_create(5);
    lora_driver_create(1, NULL);

    // Create tasks
    create_tasks_and_semaphores();

    //enable interrupts
    sei();

    // initialize timer
    intialize_timer();

    //start ADC
    init_adc();
}
```

Figure - Sensor Application System Initialization Implementation

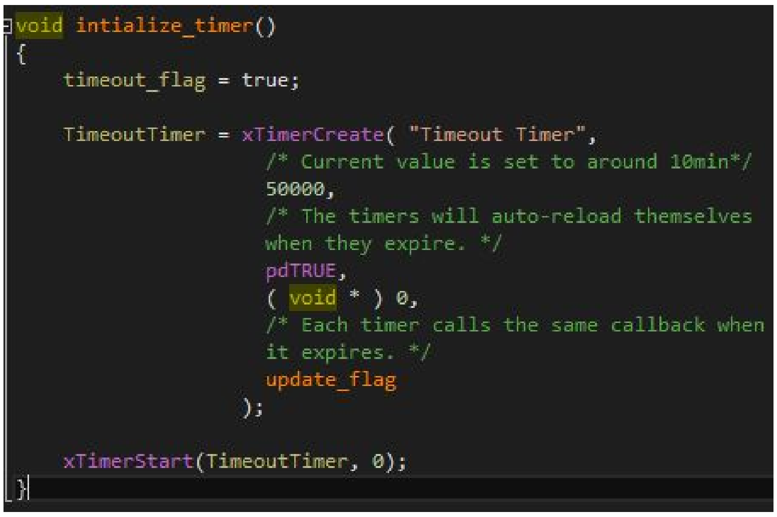
In the "*create_tasks_and_semaphores()"* method three tasks are created. A task for the temperature, pressure and LoRa. Each task has a name, stack size and a priority defined. The LoRa task has the highest priority since it needs the most computation task to complete.



```
void create_tasks_and_semaphores(void)
{

    //creating a queue that can hold two items of size uint16
    xQueue = xQueueCreate(2,sizeof(struct reading));

    xTaskCreate(
    temperature
    ,  (const portCHAR *)"temperature_sensor"  // A name just for humans
    ,  configMINIMAL_STACK_SIZE
    ,  NULL
    ,  2  // Priority, with 3 being the highest
    ,  NULL );

    xTaskCreate(
    pressure
    ,  (const portCHAR *)"pressure_sensor"  // A name just for humans
    ,  configMINIMAL_STACK_SIZE
    ,  NULL
    ,  2  // Priority, with 3 being the highest
    ,  NULL );

    xTaskCreate(
    lorawan
    ,  (const portCHAR *)"lorawan_handler"  // A name just for humans
    ,  configMINIMAL_STACK_SIZE
    ,  NULL
    ,  3  // Priority, with 3 being the highest
    ,  NULL );
}
```

Figure - Creating Tasks And Semaphores

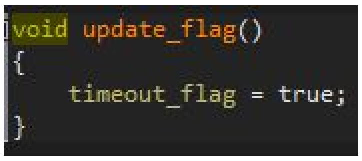
In the "*initialize_timer"* method, the timeout flag is set to true, in order for the sensors to be allowed to send their initial reading. Next the timer is created. The current timeout is set to 10min. The timer is set to reload itself every time it expires. And the callback function it will call on each timeout is the "*update_flag"* function. Finally the timer is started.



```
void intialize_timer()
{
    timeout_flag = true;

    TimeoutTimer = xTimerCreate( "Timeout Timer",
                        /* Current value is set to around 10min*/
                        50000,
                        /* The timers will auto-reload themselves
                        when they expire. */
                        pdTRUE,
                        ( void * ) 0,
                        /* Each timer calls the same callback when
                        it expires. */
                        update_flag
                      );

    xTimerStart(TimeoutTimer, 0);
}
```

Figure - Timer Initialization

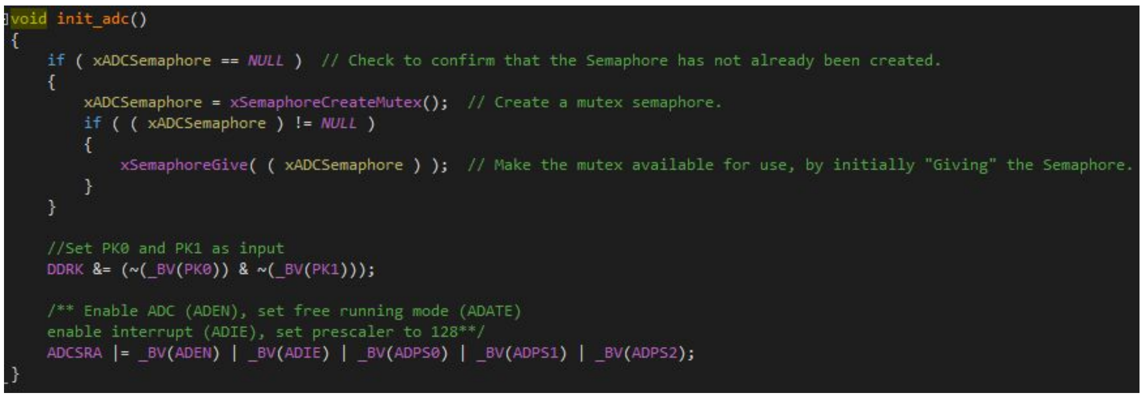
The following figure shows the implementation of the timer callback. The callback sets the timeout_flag to true which later will be consumed by the LoRa task, which will set it to false.



```
void update_flag()
{
    timeout_flag = true;
}
```

Figure - Timer Callback

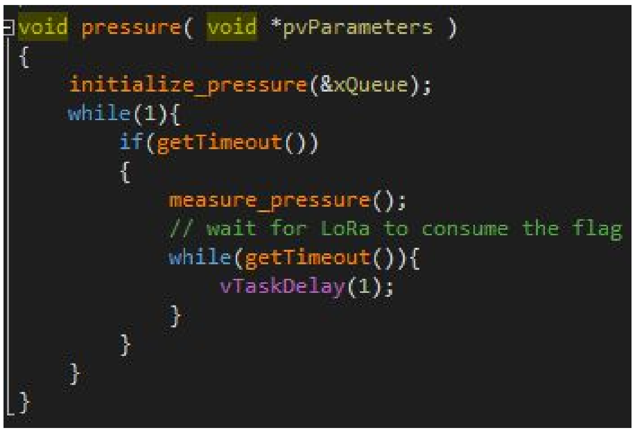
Lastly the ADC was initialized. The *"init_adc*" method from *"ADC_driver.c*" creates a semaphore for the ADC usage. Since both the temperature task and the pressure task need to use the ADC in order to convert the values a semaphore was created. The semaphore will prevent the tasks to use the ADC at the same time which could corrupt the readings. Next in the Data Direction port the two bits on which the temperature and pressure are received on are set to input. Lastly the ADC is enabled, set to free running mode, the conversion complete interrupt enabled and the prescaler is set to 128.



```c
void init_adc()
{
    if ( xADCSemaphore == NULL )  // Check to confirm that the Semaphore has not already been created.
    {
        xADCSemaphore = xSemaphoreCreateMutex();  // Create a mutex semaphore.
        if ( ( xADCSemaphore ) != NULL )
        {
            xSemaphoreGive( ( xADCSemaphore ) );  // Make the mutex available for use, by initially "Giving" the Semaphore.
        }
    }

    //Set PK0 and PK1 as input
    DDRK &= (~(_BV(PK0)) & ~(_BV(PK1)));

    /** Enable ADC (ADEN), set free running mode (ADATE)
    enable interrupt (ADIE), set prescaler to 128**/
    ADCSRA |= _BV(ADEN) | _BV(ADIE) | _BV(ADPS0) | _BV(ADPS1) | _BV(ADPS2);
}
```

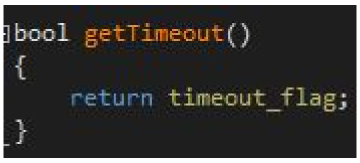
Figure - ADC initialization

The following code shows how the measurement of pressure is conducted. Since the temperature is measured in a similar way only the pressure collection process is presented.
The code shown below is the task definition. When the application is initialized the task will call the "*initialize_pressure*" method from the *"pressure.c"* file. Next the task will enter it's loop. The task is only allowed to measure the pressure when the timeout has occurred, therefore the task is constantly checking the timeout flag, by calling the "*getTimeout"* in the "*timer_driver*".



```c
void pressure( void *pvParameters )
{
    initialize_pressure(&xQueue);
    while(1){
        if(getTimeout())
        {
            measure_pressure();
            // wait for LoRa to consume the flag
            while(getTimeout()){
                vTaskDelay(1);
            }
        }
    }
}
```

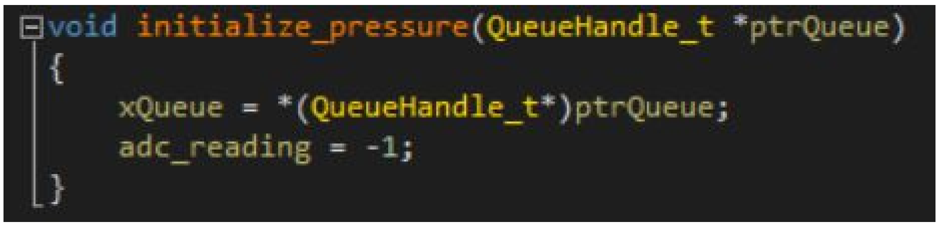
Figure - Pressure Task



```c
bool getTimeout()
{
    return timeout_flag;
}
```

Figure - Get Timeout Method

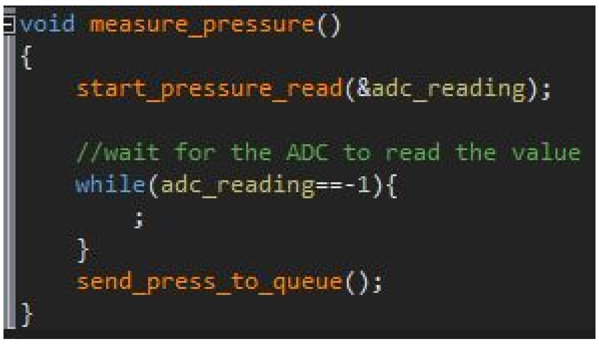
In the "*initialize_pressure*" the queue pointer is dereferenced and the "*adc_reading"* variable is initialized to -1.



```
void initialize_pressure(QueueHandle_t *ptrQueue)
{
    xQueue = *(QueueHandle_t*)ptrQueue;
    adc_reading = -1;
}
```

Figure - Initialize Pressure Task

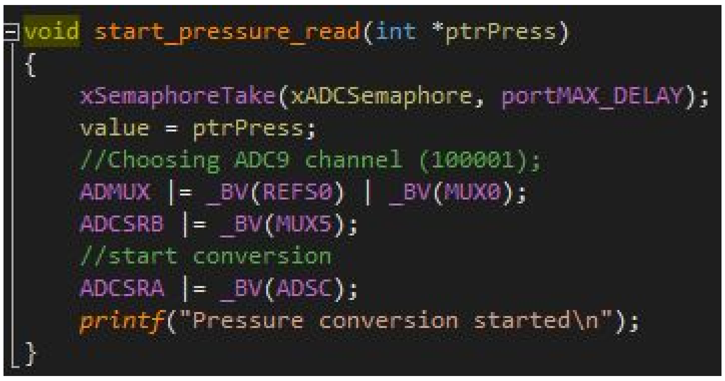
Once the returned value of the timeout is *true* the task is able to continue with collecting the measurement. The code proceeds to the “*pressure.c*” file. From there a function in the “ADC_driver.c” file is called, and a pointer to the reading variable is provided, so the ADC can store the converted value in it.



```
void measure_pressure()
{
    start_pressure_read(&adc_reading);

    //wait for the ADC to read the value
    while(adc_reading==-1){
        ;
    }
    send_press_to_queue();
}
```

Figure - Measure Pressure Method

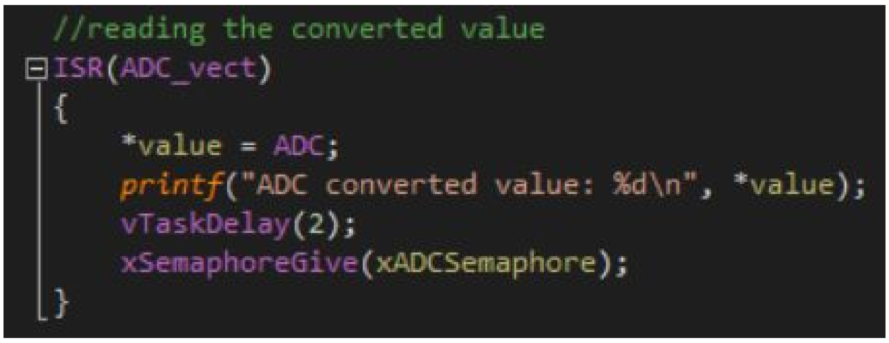
In the beginning of “*start_pressure_read*” the semaphore is taken in order for the task not to be interrupted while ADC is converting the value. Next the pointer that was provided by the pressure driver is dereferenced. Later the channel of the ADC has to be switched so the voltage from the correct pin is converted. Since the ADC is able to convert one value at a time the pressure and temperature tasks need to take turns in using the ADC. Finally the “*ADSC”* bit is set and the conversion is started.



```
void start_pressure_read(int *ptrPress)
{
    xSemaphoreTake(xADCSemaphore, portMAX_DELAY);
    value = ptrPress;
    //Choosing ADC9 channel (100001);
    ADMUX |= _BV(REFS0) | _BV(MUX0);
    ADCSRB |= _BV(MUX5);
    //start conversion
    ADCSRA |= _BV(ADSC);
    printf("Pressure conversion started\n");
}
```

Figure - Start Pressure Read Method

Since the Conversion Complete interrupt was enabled in the "*init_adc()*" when the ADC finishes converting the interrupt is triggered. In the interrupt handler the converted value is read and stored in the variable provided by "*pressure.c*". Finally the semaphore can be returned, so the other task can convert the voltage on its own pin.

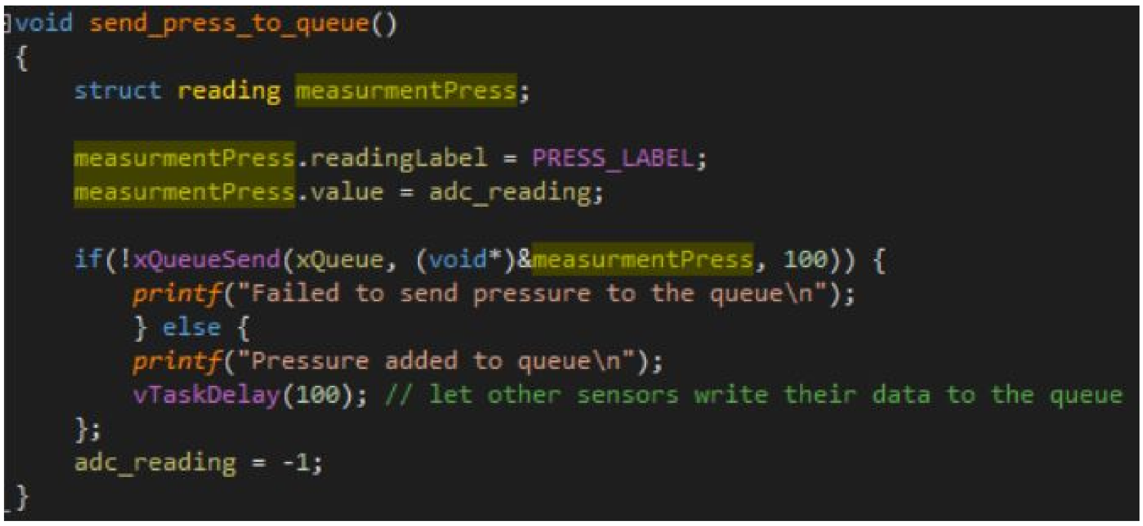


```
//reading the converted value
ISR(ADC_vect)
{
    *value = ADC;
    printf("ADC converted value: %d\n", *value);
    vTaskDelay(2);
    xSemaphoreGive(xADCSemaphore);
}
```

Figure - Interrupt Handler

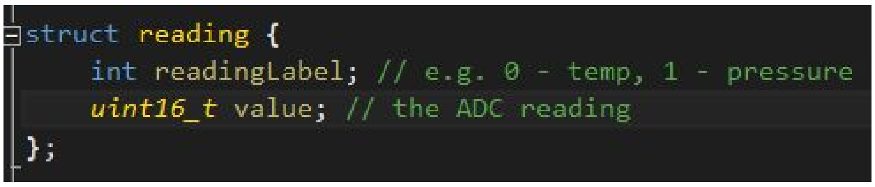
Whereas the ADC is converting the value the pressure task is continuously checking if the ADC driver has already overwritten the "*adc_variable*", which was initialized to -1. Once the value is changed the task continues to the "*send_press_to_queue()*". The described code snippet can be seen in the "*Measure Pressure Method"* figure.

In the "*send_pressure_to_queue()*" an instance of the struct defined in "*reading_struct.h*" is created. In order to distinguish the temperature and pressure data when reading from the queue the struct includes a label. Next the value is sent to the queue. Lastly the "*adc_reading*" variable is restored to its original value so the next cycle is conducted correctly.



```
void send_press_to_queue()
{
    struct reading measurmentPress;

    measurmentPress.readingLabel = PRESS_LABEL;
    measurmentPress.value = adc_reading;

    if(!xQueueSend(xQueue, (void*)&measurmentPress, 100)) {
        printf("Failed to send pressure to the queue\n");
        } else {
        printf("Pressure added to queue\n");
        vTaskDelay(100); // let other sensors write their data to the queue
    };
    adc_reading = -1;
}
```

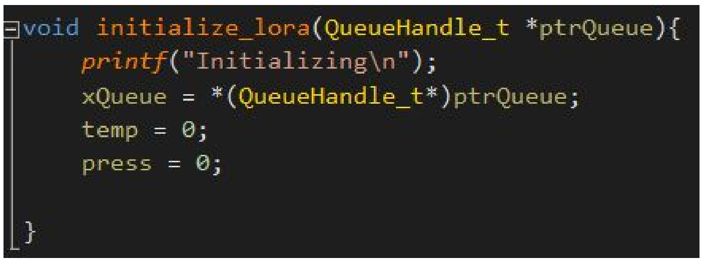
Figure - Adding Pressure to Queue



```
struct reading {
    int readingLabel; // e.g. 0 - temp, 1 - pressure
    uint16_t value; // the ADC reading
};
```

Figure - Reading Struct

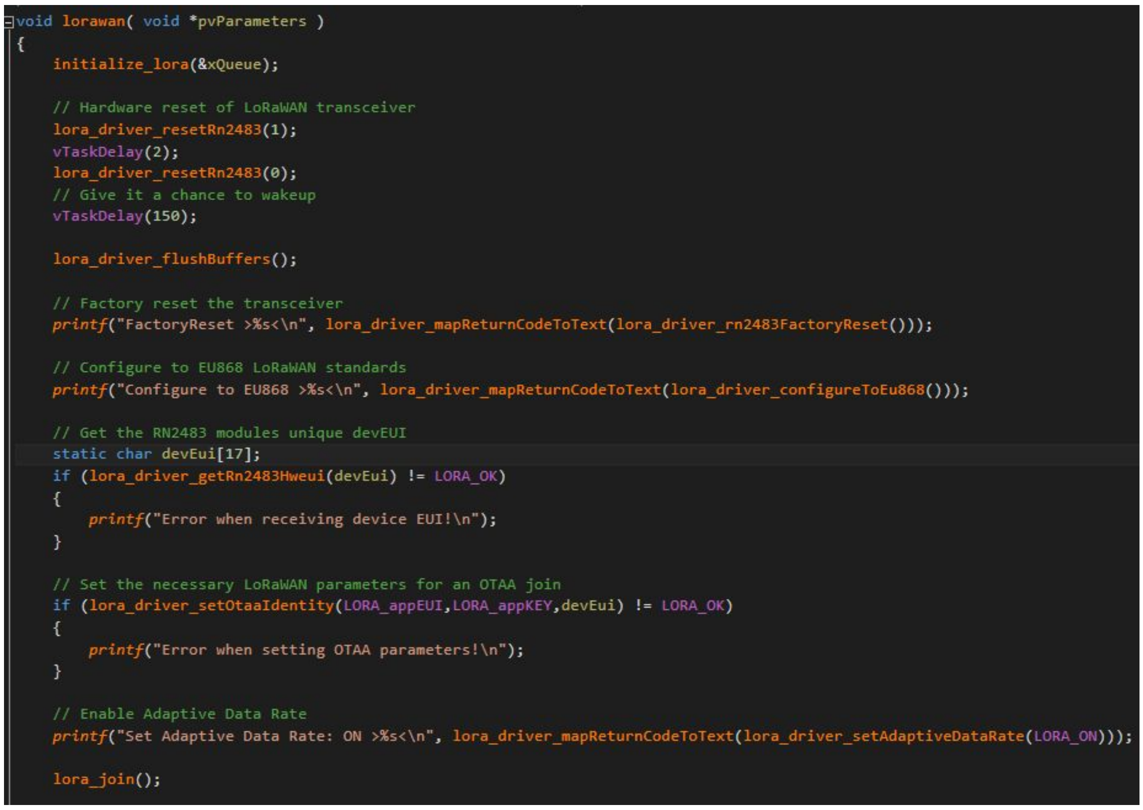
Once the measurements are in the queue the system will send them. This functionality is implemented in the LoRa task. First the task is initialized by passing a pointer to the queue so the driver can access the values stored in it, and the required variables initialized.



```
void initialize_lora(QueueHandle_t *ptrQueue){
    printf("Initializing\n");
    xQueue = *(QueueHandle_t*)ptrQueue;
    temp = 0;
    press = 0;

}
```

Figure - Initializing LoRa task

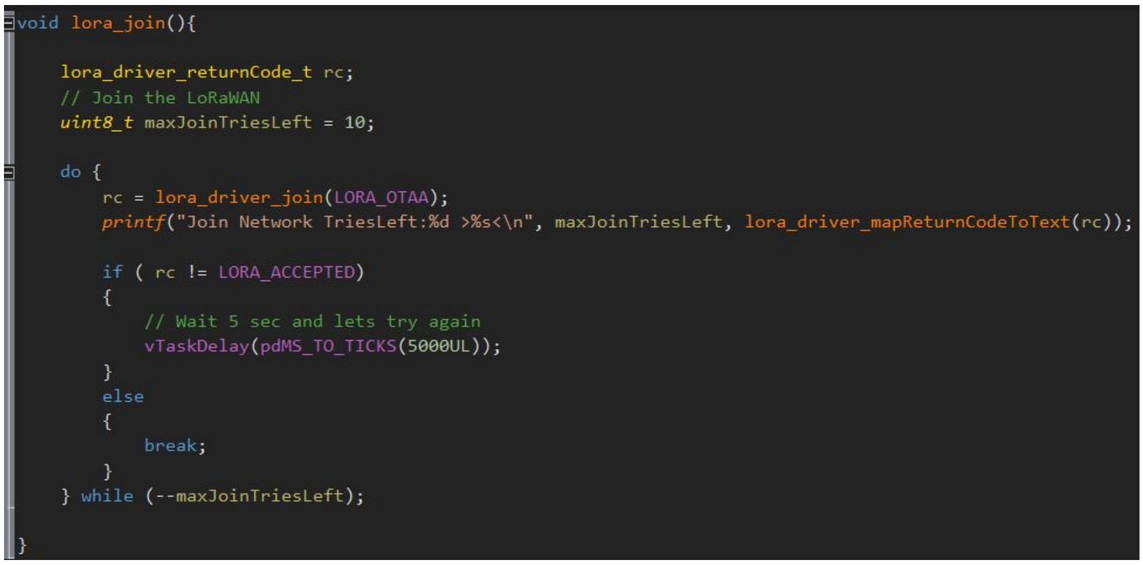
Next the LoRa transceiver needs to be prepared for usage. In order to do that the hardware needs to go through a sequence of actions. Firstly it is reseted, next the buffers flushed, factory reset performed, EU868 standard configured, the device EUI received, OTAA parameters set and adaptive data rate set. All these functions are implemented in an external LoRa driver implemented by Ib Havn. Finally the transceiver joins the network.



```c
void lorawan( void *pvParameters )
{
    initialize_lora(&xQueue);

    // Hardware reset of LoRaWAN transceiver
    lora_driver_resetRn2483(1);
    vTaskDelay(2);
    lora_driver_resetRn2483(0);
    // Give it a chance to wakeup
    vTaskDelay(150);

    lora_driver_flushBuffers();

    // Factory reset the transceiver
    printf("FactoryReset >%s<\n", lora_driver_mapReturnCodeToText(lora_driver_rn2483FactoryReset()));

    // Configure to EU868 LoRaWAN standards
    printf("Configure to EU868 >%s<\n", lora_driver_mapReturnCodeToText(lora_driver_configureToEu868()));

    // Get the RN2483 modules unique devEUI
    static char devEui[17];
    if (lora_driver_getRn2483Hweui(devEui) != LORA_OK)
    {
        printf("Error when receiving device EUI!\n");
    }

    // Set the necessary LoRaWAN parameters for an OTAA join
    if (lora_driver_setOtaaIdentity(LORA_appEUI,LORA_appKEY,devEui) != LORA_OK)
    {
        printf("Error when setting OTAA parameters!\n");
    }

    // Enable Adaptive Data Rate
    printf("Set Adaptive Data Rate: ON >%s<\n", lora_driver_mapReturnCodeToText(lora_driver_setAdaptiveDataRate(LORA_ON)));

    lora_join();
```

Figure - LoRa Task Set Up

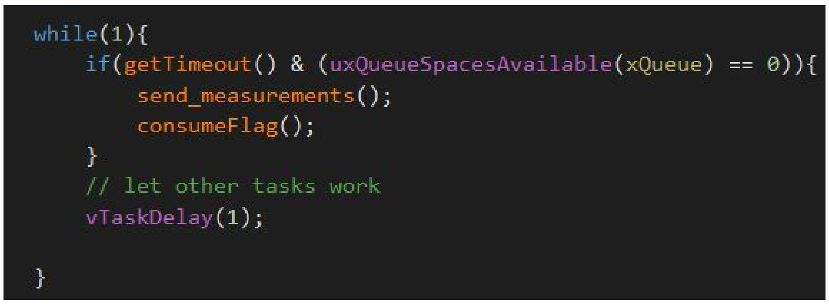
In order to join the LoRa network the attempt is repeated 10 times. When the attempt fails the task will wait for 5 seconds before repeating the action. In case when the joining has succeeded the loop will break and the function will be completed.



```
void lora_join(){

    lora_driver_returnCode_t rc;
    // Join the LoRaWAN
    uint8_t maxJoinTriesLeft = 10;

    do {
        rc = lora_driver_join(LORA_OTAA);
        printf("Join Network TriesLeft:%d >%s<\n", maxJoinTriesLeft, lora_driver_mapReturnCodeToText(rc));

        if ( rc != LORA_ACCEPTED)
        {
            // Wait 5 sec and lets try again
            vTaskDelay(pdMS_TO_TICKS(5000UL));
        }
        else
        {
            break;
        }
    } while (--maxJoinTriesLeft);

}
```

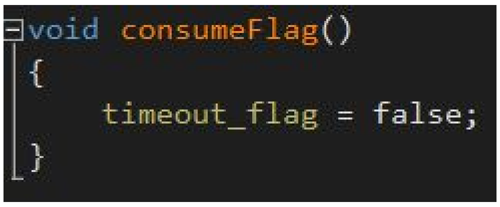
Figure - LoRa Joining Network

When all the setup is completed the task can serve its main functionality, which is send the measurements. The sending will only be performed when the timeout indicates so and the pressure and temperature tasks have filled up the queue. Whenever both conditions are met the task will send the measurements and consume the timeout flag.



```
while(1){
    if(getTimeout() & (uxQueueSpacesAvailable(xQueue) == 0)){
        send_measurements();
        consumeFlag();
    }
    // let other tasks work
    vTaskDelay(1);

}
```

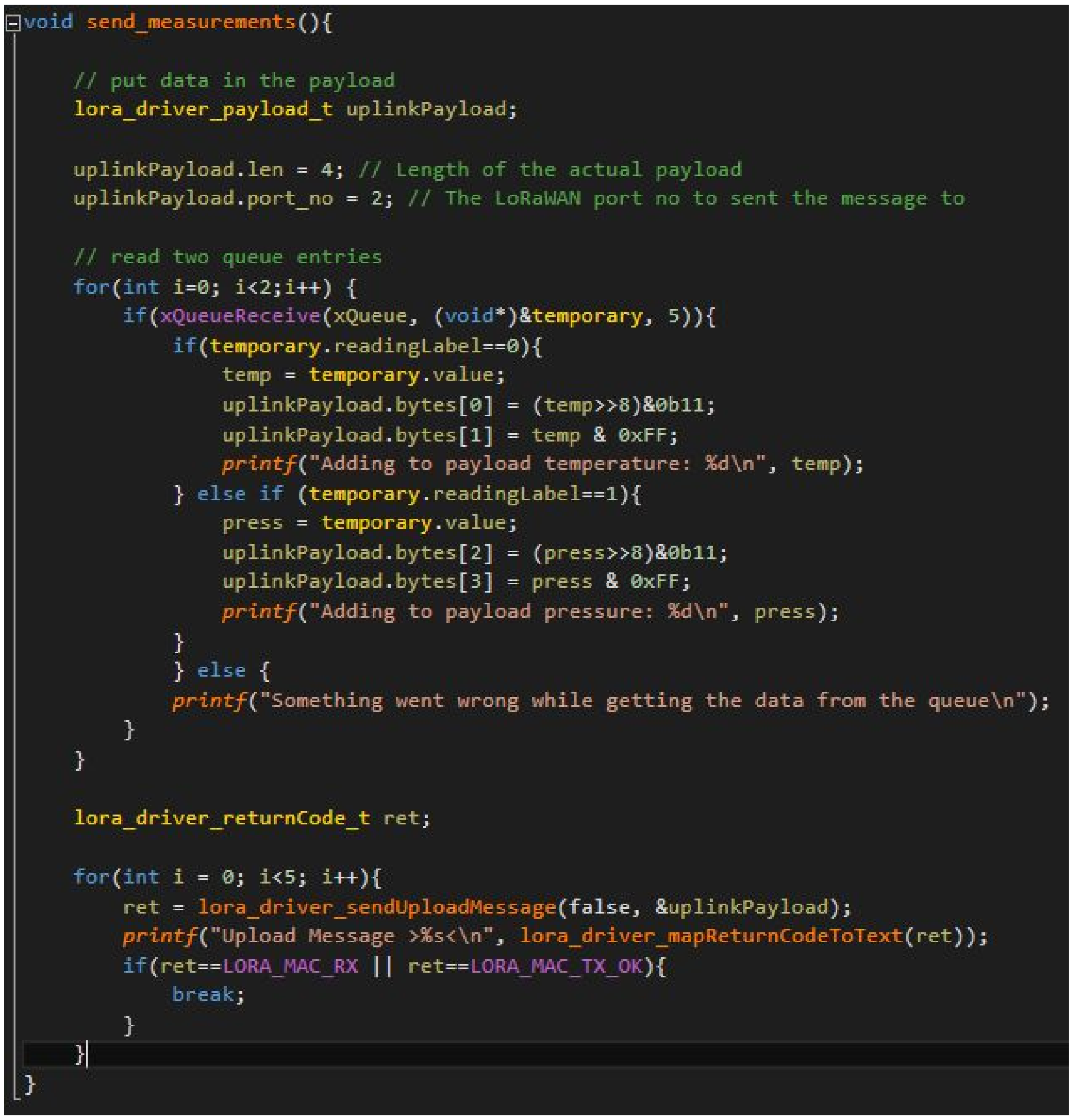
Figure - LoRa Task Main Loop



```
void consumeFlag()
{
    timeout_flag = false;
}
```

Figure - Consume Flag

In order to send a payload to LoRa an instance of *“lora_driver_payload_t”* is created. The length of the payload is 4 since 2 bytes are needed to send the temperature measurements as well as 2 bytes for the pressure. Next there is a loop to iterate over all the measurements in the queue. All measurements are received and next the label is checked. If the label indicates it is a temperature reading then the values are stored in the first two bytes of the payload, and if the label implies its pressure the the value is stored in the last two bytes of the payload. Finally when the payload is ready to be sent out, the message is uploaded within 5 tries until it receives a success response. Once the data is in the LoRa server the LoRa Socket Interface takes care of receiving it.



```
void send_measurements(){

    // put data in the payload
    lora_driver_payload_t uplinkPayload;

    uplinkPayload.len = 4; // Length of the actual payload
    uplinkPayload.port_no = 2; // The LoRaWAN port no to sent the message to

    // read two queue entries
    for(int i=0; i<2;i++) {
        if(xQueueReceive(xQueue, (void*)&temporary, 5)){
            if(temporary.readingLabel==0){
                temp = temporary.value;
                uplinkPayload.bytes[0] = (temp>>8)&0b11;
                uplinkPayload.bytes[1] = temp & 0xFF;
                printf("Adding to payload temperature: %d\n", temp);
            } else if (temporary.readingLabel==1){
                press = temporary.value;
                uplinkPayload.bytes[2] = (press>>8)&0b11;
                uplinkPayload.bytes[3] = press & 0xFF;
                printf("Adding to payload pressure: %d\n", press);
            }
            } else {
            printf("Something went wrong while getting the data from the queue\n");
        }
    }

    lora_driver_returnCode_t ret;

    for(int i = 0; i<5; i++){
        ret = lora_driver_sendUploadMessage(false, &uplinkPayload);
        printf("Upload Message >%s<\n", lora_driver_mapReturnCodeToText(ret));
        if(ret==LORA_MAC_RX || ret==LORA_MAC_TX_OK){
            break;
        }
    }
}
```

Figure - Uploading Payload to LoRaWAN

Test

Testing for the system was carried out during implementation, but more towards the end of the project. This wasn't the initial plan as shown in the Project Description(see Appendix: Time Schedule). Time constraints, critical report sections and unexpected delays in acquiring critical data contributed to the need to shuffle the schedule in order to ensure that the project would meet the specified deadline and be ready to hand over to Herning Vand.

As a result of these difficulties and how the system was implemented Unit testing had to be meshed into Integration testing. However, the strategy to run the tests through Integration Testing, System Testing and finally Acceptance Testing continued.

Test Specifications

The general test specifications for the system to be checked during the various testing phases are as follows.

1. Accurately label pump state given pump attributes
2. Store and display continuously collected sensor signals
3. Expose an API from python
4. Store information in a database
5. Host a web application

Unit Testing

Unit testing was not prioritized for this project. The system implementation operates in such a way that testable functionality was abstracted away by various libraries and frameworks needed for the heterogeneous system to work as intended. As a result testing continued with Integration testing.

Integration Testing

Integration testing was the most intensive due to the nature and implementation of the system. The integration testing was performed in a grey box format, a combination of white and black box testing. This means that the team tested the system with only a partial understanding of the system and reports any failures or anomalies.

For the system to be brought to completion, it depended on the individual modules ability to communicate and pass, store, retrieve and display an input. This integration testing was generally carried out by observing the running system from the outside to ensure that the dummy data would appear in the UI as expected.

The different modules of the system were tested incrementally, meaning that we performed integration tests with the different modules as they became available. There was no further strategy used for the overall process during the integration tests.

Critical Modules:

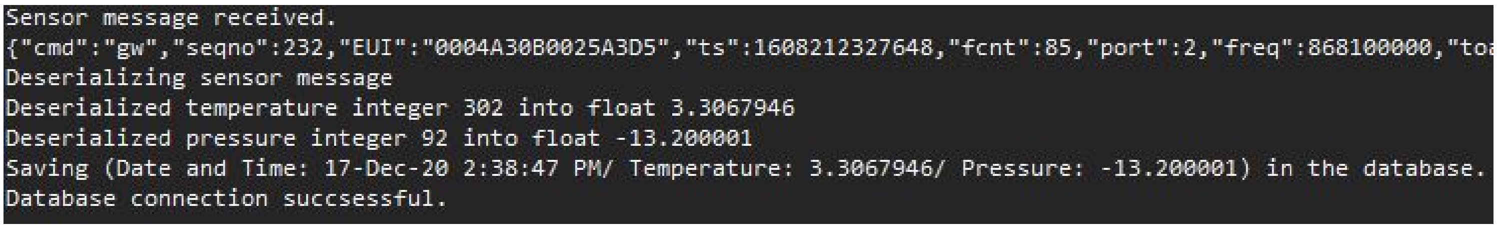
- Web Application
- Python Server
- Database and EF Core
- LoRa socket
- Sensor application

Test Cases:

Test Case	Requirement	Expected Result
1. Print the request and the response by the python api server.	Check the request/response PUT HTTP connection between the web application and the python server	The web application receives a prediction based on data entry.
2. Store a prediction done by the python api server.	Check the connection between the web application and the database via EF Core	The system saves and views data from the database.
3. Listen, receive and print a sensor message sent by the LoRa server.	Check the connection between the Arduino and the LoRaWAN server	The LoRa server receives timed signals from the arduino.
4. Listen, receive and print a sensor message sent by the LoRa server.	Check the connection between the LoRaWAN server and the socket interface.	The socket interface listens and receives signals, delivered to the LoRa server.

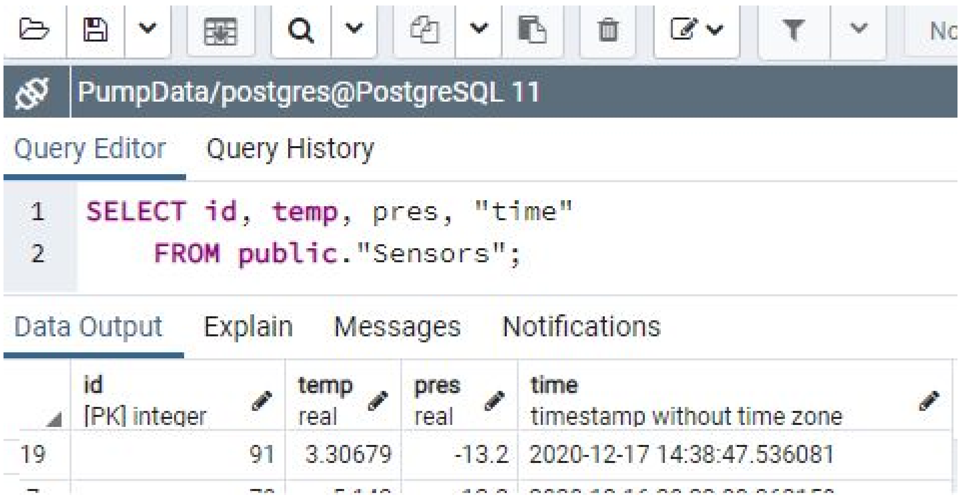
Testing the retrieval of sensor messages by the socket

A message from the LoRa server was awaited and tested to verify the validity of the socket interface and the data consistency throughout the signal path.



```
Sensor message received.
{"cmd":"gw","seqno":232,"EUI":"0004A30B00025A3D5","ts":1608212327648,"fcnt":85,"port":2,"freq":868100000,"toa
Deserializing sensor message
Deserialized temperature integer 302 into float 3.3067946
Deserialized pressure integer 92 into float -13.200001
Saving (Date and Time: 17-Dec-20 2:38:47 PM/ Temperature: 3.3067946/ Pressure: -13.200001) in the database.
Database connection succsessful.
```

Figure - System print of the received sensor message



PumpData/postgres@PostgreSQL 11

Query Editor Query History

```
SELECT id, temp, pres, "time"
    FROM public."Sensors";
```

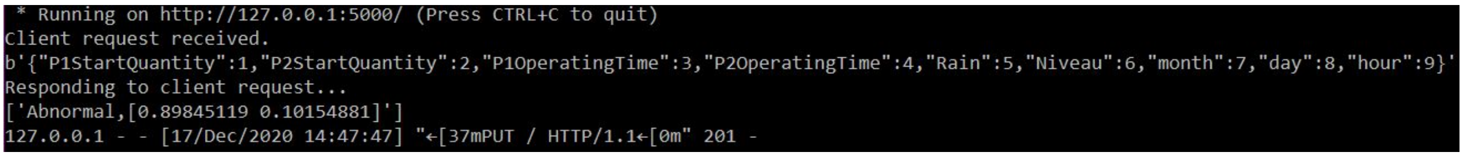
Data Output Explain Messages Notifications

	id [PK] integer	temp real	pres real	time timestamp without time zone
19	91	3.30679	-13.2	2020-12-17 14:38:47.536081

Figure - Database table displaying the stored message

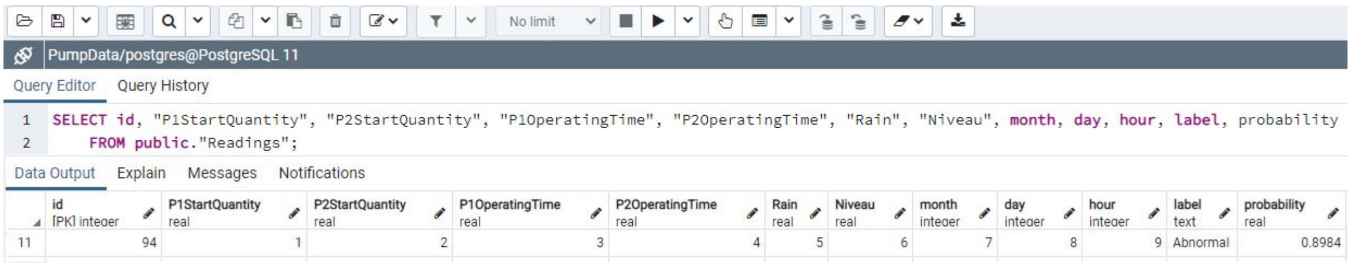
Testing the requests and responses of the python api server

A test signal was sent to the python server in order to verify that the interface is expecting requests and returning responses.



```
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
Client request received.
b'{"P1StartQuantity":1,"P2StartQuantity":2,"P1OperatingTime":3,"P2OperatingTime":4,"Rain":5,"Niveau":6,"month":7,"day":8,"hour":9}'
Responding to client request...
['Abnormal,[0.89845119 0.10154881]']
127.0.0.1 - - [17/Dec/2020 14:47:47] "←[37mPUT / HTTP/1.1←[0m" 201 -
```

Figure - System print of the received request and the sent response by the python server



```
SELECT id, "P1StartQuantity", "P2StartQuantity", "P1OperatingTime", "P2OperatingTime", "Rain", "Niveau", month, day, hour, label, probability
    FROM public."Readings";
```

	id [PK] integer	P1StartQuantity real	P2StartQuantity real	P1OperatingTime real	P2OperatingTime real	Rain real	Niveau real	month integer	day integer	hour integer	label text	probability real
11	94	1	2	3	4	5	6	7	8	9	Abnormal	0.8984

Figure - Database table displaying the stored request

System Testing

System testing encompasses the entire system, but there were some restrictions that needed to be considered. System testing was conducted as grey box testing. This decision was made based upon travel restrictions due to an ongoing pandemic and the sensitivity of the equipment necessary to transport and set up at Herning Vand to conduct the system tests properly. Finally, due to a delimitation of the project restricting the use of the internet as much as reasonably possible, it was not feasible to deploy any modules to a cloud-based service as a solution to transport issues.

The goal of the grey box based system test was to ensure that all inputs into the system from the users, in this case the developers, would produce the desired outputs in the web page UI. The system test would also be a way to ensure that all components were functioning as intended.

Acceptance Testing

Due to the pandemic and the inability to transport the necessary equipment, acceptance testing was conducted remotely. The conducted black box testing was executed throughout the company's meeting and the results were discussed with the project lead at Herning Vand.

The requirements as discussed by the test specifications were gone over and it was shown that they were met in the proof of concept system. The company has acknowledged the results that the outputs to the web page UI were as expected. Anomalies were displayed. At the conclusion of the test, the Herning Vand project lead discussed the potential future of the proof of concept as a product and asked for a reference to the report.

(To read more on Black Box testing, please refer to Appendix F/Black_Box_Testing.pdf)

Hardware Testing

Numerous tests were performed to ensure the correct functioning of the sensors. Must tests were performed for the temperature sensor since it required external circuits.

The first component that was tested was the Wheatstone Bridge. Only one test was performed for it:

- For the temperature 0°C the V_{ab} = 0V
  - The measurements on both arms of the bridge were 2.39V
  - Result: PASSED

Next the instrumentation amplifier was tested. The amplifier had to meet two requirements. The voltage at 0°C had to be 0V and at 20°C it had to be 5V, therefore the two tests were performed:

- For the temperature 0°C the V_{out} is 0V
  - The measurement was: 0.48V
  - Result: PASSED
- For the temperature 20°C the V_{out} is 5V
  - The measurement was: 5.1V
  - Result: PASSED

The measurements were not exactly the same as specified in the test case, but since the error was not significant enough to influence the system, therefore the tests were considered passed.

Finally accuracy of the temperature conversion was tested. The following table shows the results of the experiment. The temperature is shown in °C.

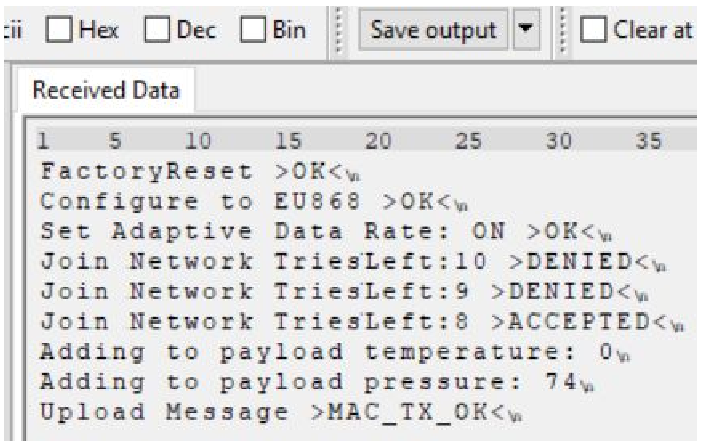
External Thermometer	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
PT100 measurement	0.1	0.9	1.4	3.2	4.3	5.1	5.8	6.5	7.5	8.6	10	11.1	12.2	13.1	14.2	15	16	17	18	18.5	18.5
Error	0.1	0.1	0.6	0.2	0.3	0.1	0.2	0.5	0.5	0.4	0	0.1	0.2	0.1	0.2	0	0	0	0	0.5	1.5

Table - Temperature Sensor Test

As it can be seen the temperature received from the PT100 is slightly different then the external indicates. The average error is calculated to 0.267. The reason for the slight difference in the calculated temperature is due to the time consuming process of making tests for the 0-20°C temperature range. If more tests were performed the ADC conversion to temperature formula could be improved. The error is small enough to satisfy the project.
Due to the lack of an external water pressure sensor the calibration of the SKU237545 pressure sensor could not be tested.

Sensor Application Testing

The sensor application is using the FreeRTOS. Because of that most of the code is scheduled by the external functionality of the RTOS. Also the application is using numerous external drivers. Because a lot of important functionality was implemented by external sources there was not much to test using unit testing. Therefore a decision was made that testing using *"printf"* statements will be used. This way the system can be tested if everything is working as planned.
In order to view that the system is working correctly the most important steps are printed to the console. The first messages that are printed out are the LoRa transceiver status messages. The messages inform whether the hardware has been correctly initialized. Next the application is trying to join the network. On the following snippet it can be seen that it succeeded to join the LoRa network on the third try. Next it is important to know if the correct data is uploaded to the payload, therefore it is displayed what is being sent. This also enables to compare the data with the received values in the LoRa Socket Interface. Lastly the application log shows whether the message has been successfully uploaded to the server.
The snippet below shows that the application is able to successfully read the values from the pins, add them to payload and upload them to the LoRa server.



```
1    5    10    15    20    25    30    35
FactoryReset >OK<\n
Configure to EU868 >OK<\n
Set Adaptive Data Rate: ON >OK<\n
Join Network TriesLeft:10 >DENIED<\n
Join Network TriesLeft:9 >DENIED<\n
Join Network TriesLeft:8 >ACCEPTED<\n
Adding to payload temperature: 0\n
Adding to payload pressure: 74\n
Upload Message >MAC_TX_OK<\n
```

Figure - Sensor Application Console Messages

Results and Discussion

The technical challenges encountered were effectively solved into a quick and accurate integration that would satisfy requirements imposed onto the proof of concept. The result of this project achieves exactly what its goals were set on.

The group has executed a thorough analysis of the problem domain. Organized and constructing a system design and the problem was addressed with an IoT system. The working proof of concept can effectively determine the state of Herning Vand's pumps with just 5 data features and a response time of less than a second. Herning Vand does not maintain an incredible amount of sensors in their pumps, but even with the ones they have, they can be used to monitor their pumps and even forecast the future state through patterns in their collected data.

By setting up thermal and pressure sensors, the proof of concept shows an example of a cost effective, fast and accurate IoT solution that is scalable and modular to suit their needs. The integration of sensors through the local LoRaWAN has proven that it is possible to set up a data gathering device onto any industrial machine and continuously collect, monitor and forecast the state of the machine through the data gathering and machine learning processes.

The system is a proof that it is possible to quickly and cheaply set up an application with a sensor interface, that collects and monitors system behaviour, which results in a potential reduction of maintenance costs.

Conclusions

The development concluded successfully, covering all the requirements set by analysis. The choice of practices allow the database to be highly scalable and modifiable without having to take down the system. This would allow the addition of new sensors to the system without crashing it.

The choice for python interface was solely based on the user input requirement set onto the proof of concept. Even though the connection to the python server is not constant, the response times are quick. The usage of Loriot services make the sensor interface abstract, saving the team some time developing complex sensor interfaces.

Although the socket listener for sensor signals is located in the Blazor application, if the system needs to be scaled, it will be a good idea to move the interface to the python module, immediately making a prediction on receiving sensor data. This would eliminate the request/ response protocol between python and .Net, turning it into a singular request protocol. Such connections will be faster and more modular, turning the Blazor Server into a business logic application that listens and records to the database.   The web application made using Blazor Server also hosts a web page based UI for the user to view the sensor and pump data.

Despite some design principles that can be improved if the proof of concept needs to be scaled to a product, the current integration and design completely satisfy the requirements showing the potential of similar systems in the industry.

Project future

Automating the data gathering process can create a data pool, which can be used to further train the machine learning algorithm.

For that case, a machine learning pipeline has to be set up on the python server. That pipeline could take data as input and further train the existing model, applying grid search models in advance to automatically assign the best parameters yielding even more accurate system monitoring.

The introduction of more sensors for various data measurements of the pumps can allow the system to apply multiple labels in its classification, resulting in the precise alarm label of the component that is breaking down. The introductions of more data features will also open up the possibility of fitting accurate forecasting models to the new data, that detect patterns of breakdowns and can predict such in the future. The sensor signals from the microcontroller to the local LoRaWAN tower make it easy to increase the frequency of data gathering as well, capturing the exact data values of an alarm event.

Scaling up the system will require an interface overhaul. The API in the python module has to be replaced by a continuous TCP Socket connection for security and optimization purposes.

The Blazor web application can be replaced by a python web application, merging two modules into one. Doing that will speed up the data manipulation processes. Switching entirely to the python language will open up the possibility of plotting data on the frontend of the web application. Such functions would allow us to graphically represent a timeline of system states, that displays future breakdowns as well.

Replacing the database connection from entity framework core to a custom made adapter will remove any dependency that the system has on the library and will open up the system for maintenance and thorough debugging.

The sensors' accuracy can be improved. Since making precise sensors requires a lot of physical testing, which is very time consuming, it was impossible to make precise sensors. Also the current circuit is created on a breadboard therefore it can be easily broken. The sensor module could be soldered. Another improvement would be using a smaller microcontroller, or even a custom breadboard for the sensors. A lot of Arduino's potential is not used in the application.

Sources of information

Apec Water, 2013. The History of Clean Drinking Water.[online] Available at: <https://www.freedrinkingwater.com/resource-history-of-clean-drinking-water.htm>

United States Environmental Protection Agency, 1998. How Wastewater Treatment Works...The Basics.[online] Available at: <https://www3.epa.gov/npdes/pubs/bastre.pdf>

Danva, 2020. Water in figures(year).[online] Available at: <https://www.danva.dk/publikationer/benchmarking-og-statistik/water-in-figures online/>

Herning Vand, (2020). “*Projects from Herning Vand*” [Powerpoint presentation]. page 8. (Accessed: 27 March 2020).

A, Jeanette., 2020. *Clarification of System Expectations for Requirement Specification*. [email] (Personal Communication, 22 April 2020).

R, Jan., 2020. *Overview of Herning Vand Water Pumping Station, Daily Workings and SCADA*. [meeting] (Personal Communication, 9 September 2020).

Microsoft, 2020. *Introduction to ASP.NET Core Blazor*. [online] Available at: <https://docs.microsoft.com/en-us/aspnet/core/blazor/?view=aspnetcore-3.1#blazor-server> [Accessed 27 October 2020].

Microsoft, 2020. *ASP.NET Core Blazor Hosting Models*. [online] Available at: <https://docs.microsoft.com/en-us/aspnet/core/blazor/hosting-models?view=aspnetcore-3.1> [Accessed 27 October 2020].

Microsoft, 2020. *Entity Framework Core*. [online] Available at: <https://docs.microsoft.com/en-us/ef/core/> [Accessed 27 October 2020].

Microsoft, 2020. *ASP.NET Core Blazor Server with Entity Framework Core (EFCore)*. [online] Available at: <https://docs.microsoft.com/en-us/aspnet/core/blazor/blazor-server-ef-core?

view=aspnetcore-3.1> [Accessed 27 October 2020].

e-statis(StackOverflow user), 2009. *What is an ORM, how does it work, and how should I use one? [closed]*. [online] Available at: <https://stackoverflow.com/questions/1279613/what-is-an-orm-how-does-it-work-and-how-should-i-use-one> [Accessed 27 October 2020].

Larman, C., 2005. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. 3rd ed. Upper Saddle River, NJ: Pearson Education, Inc.

Aurélien Géron, 2019. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems.* 2nd ed. O’REILLY.

Sarah Guido, 2017. *Introduction to Machine Learning with Python: A Guide for Data Scientists.* O’REILLY.

Appendices

The following files can be found in the attached appendices:

- AppendixA - Project Description and Requirement_Specification
  - Project_Description.pdf
  - Requirement_Specification.pdf
- AppendixB - Analysis
  - Conceptual_Domain_Model.asta
  - Conceptual_System_Sequence.asta
  - Use_Cases.asta
  - View_Forecast_Activity.asta
  - View_Sensor_Data_Activity.asta
- AppendixC - Design
  - Diagram_Images
    - Arduino_Architecture.jpg
    - Arduino_Class_Diagram.jpg
    - Class_Diagram.jpg
    - Database_ER_Diagram.png
    - Label_Function_Sequence_Diagram.jpg
    - Sensor_Signal_Sequence_Diagram.jpg
    - System_Architecture.jpg
    - User_Interface.png
  - Model_Research
    - Fitting_Multiple_Models.pdf
  - Circuit_Simulation
    - instrumentation_amplifier.asc
    - wheatstone_bridge.asc
  - Astah_Diagrams
    - Arduino_Activity_Diagram.asta
    - Arduino_Architecture.asta
    - Arduino_Sequence_Diagram.asta
    - Arduino_System_Class_Diagram.asta
    - Class_Diagram.asta
    - DB_ER_Diagram.asta
    - Label_Sequence_Diagram.asta
    - Sensor_Signal_Sequence_Diagram.asta
- AppendixD - Implementation
  - circuit_components.pdf
- AppendixE - Testing
  - Black_Box_Testing.pdf