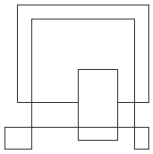


ForecastMonitor

for Forecast System

Tomas Izo, Georgi Stefanov, Ophelia Marott Zhang

Supervisor: Ole Ildsgaard Hougaard

SYSTEMATIC

ICT Engineering

7th semester

10/12/2018

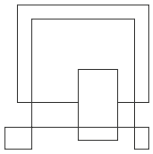


Table of content

Abstract vii

1 Introduction 1

1.1 Background description 1

1.2 Definition of purpose 2

1.3 Problem statement 2

1.4 Delimitation 3

1.5 Choice of models and methods 4

2 Requirements 4

2.1 Functional Requirements 4

2.1.1 Overall structure (OS) 4

2.1.2 Prediction performance (PP) 5

2.1.3 Feature specifications (FS) 5

2.1.4 Model information (MI) 6

2.2 Non-Functional Requirements 6

2.2.1 Technology requirements (TR) 6

3 Analysis 7

3.1 Forecast System monitoring purpose and current approach 7

3.2 “In which format” is the data monitored 9

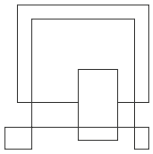
3.3 “How” should the data be monitored 11

3.4 Technology for the purpose 11

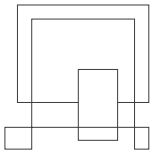
3.5 General Project Development method 12

3.5.1 Feature List 13

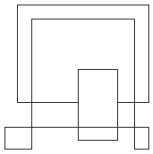
3.5.2 Feature Roadmap 13



4 Design 14
4.1.1 SOLID Principles in Mind 15
4.2 ForecastMonitor Service 16
4.2.1 Service Oriented Architecture (SOA) 16
4.2.2 General Overview 18
4.2.3 Data Access Layer 20
4.2.4 Business Logic Layer 23
4.2.5 Service Layer 26
4.2.6 Configuration 27
4.2.7 Jobs 28
4.3 ForecastMonitor UI 32
4.3.1 Design patterns 33
4.3.2 User Interface 34
5 Implementation 36
5.1 ForecastMonitor Service 36
5.1.1 Startup class 37
5.1.2 Background services & consuming scoped services 38
5.1.3 Job chaining using MediatR 40
5.1.4 Units publishing with SignalR over Web Sockets 41
5.1.5 Mapping 42
5.2 ForecastMonitor UI 43
5.3 Test Projects 46
5.4 Testing framework - NUnit 47
5.4.1 FluentAssertions 47
5.4.2 Moq & AutoFixture 47



5.4.3 WireMock.Net & Handlebars.Net 48
5.4.4 Selenium WebDriver 48
5.4.5 Microsoft.AspNetCore.MVC.Testing 49
6 Test 50
6.1 Test strategy 50
6.2 Test varieties 51
6.2.1 Unit test 52
6.2.2 Integration test 53
6.2.3 End-to-end test 54
6.2.4 UI test 55
6.2.5 Manual test 55
7 Results and Discussion 56
8 Conclusions 58
9 Project future 59
10 Sources of information 60
11 Appendices 1
11.1 A – Feature List 1
11.2 B – Choice of Models and Methods 1
11.3 C – Feature Roadmap 1
11.4 D – Forecast System API Description 1
11.5 E – Test Specification Document 1
11.6 F – ForecastMonitor UI Architecture 1
11.7 G – Use Case Descriptions 1
11.8 H – API Design 1
11.9 I – ForecastMonitor Service Architecture 1



11.10 J – Feature Description Documents 1
11.11 K – Sprint Reports 1
11.12 L – Sprint Planning and Burndown Charts 1
11.13 M – Implementation 1
11.14 N – Process Report 1

List of figures and tables

Figure 1 Forecast System Hierarchy 1
Figure 2 Use Case Diagram 8
Figure 3 Use case description for Monitor Forecast Units 9
Figure 4 Forecast System data schema 10
Figure 5 FDD activities used in this project 12
Figure 6 Example of a feature list item – FDD0001 Display Unit Information 13
Figure 7 ForecastMonitor Design 14
Figure 8 Job types class diagram 15
Figure 9 ForecastMonitor Service - General overview (Service Layer pattern) 18
Figure 10 Forecast System consumer class diagram 21
Figure 11 Memory storage class diagram 22
Figure 12 MemoryDataContext Set operation sequence diagram 22
Figure 13 Data Model 23
Figure 14 PerformanceCalculationLogic class diagram 24
Figure 15 PerformanceCalculationLogic.MAE sequence diagram 24
Figure 16 ClientLogic class diagram 25
Figure 17 UnitPublishingService class diagram 26
Figure 18 UnitPublishingService.PublishUnits sequence diagram 26
Figure 19 Service Layer class diagram 27
Figure 20 API endpoint example 27
Figure 21 AppSettingsManager class diagram 28
Figure 22 Job class diagram 29
Figure 23 Job.ExecuteAsync sequence diagram 30

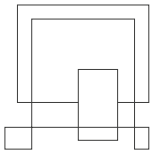
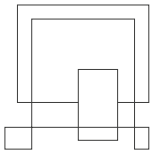


Figure 24 JobNotificationHandler class diagram 30
Figure 25 ScheduledJobService class diagram 31
Figure 26 AdHocJobService class diagram 32
Figure 27 Client Component class diagram 32
Figure 28 Dashboard page 35
Figure 29 Model Performance page 36
Figure 30 .Net Core Startup class overview 37
Figure 31 Service Startup class middleware setup 38
Figure 32 AdHocJobService implementation 39
Figure 33 Consuming scoped services 40
Figure 34 Job class implementation 41
Figure 35 Publishing units with SignalR 42
Figure 36 AutoMapper mapping profile 42
Figure 37 Retrieve unit's data – viewModel 43
Figure 38 Retrieve unit's data – service 43
Figure 39 Part of the client component 44
Figure 40 Part of the unit component 44
Figure 41 Update clients' status function 45
Figure 42 Update installations status 46
Figure 43 FluentAssertions example 47
Figure 44 AutoFixture as auto-mocking container 47
Figure 45 Http responses stubbing 48
Figure 46 TestServer services setup example 49
Figure 47 Test Design Table for feature FDD0001 51
Figure 48 Unit Test 53
Figure 49 Integration Test 54
Figure 50 E2E Test 54
Figure 51 UI Test 55
Figure 52 Feature statuses 57



Abstract

This report describes a solution to the need of the performance monitoring system of Systematic's Forecast System – a service using machine learning for predicting an occupancy for a given hospital section. The solution is a dashboard, intended to be used internally - by machine learning engineers, or on site – by technically minded personnel. It covers the most important monitoring needs and remains open for an extension. To fit within Systematic's healthcare services suite, the design consists of .Net Core 2.1 RESTful service and Angular 6 web application.

1 Introduction

This section covers background information of the project

1.1 Background description

Systematic has several business units, one of them being healthcare. The company offers wide range of e-health solutions - the main idea of all revolves around helping clinical personnel and healthcare professionals to save time on their daily tasks and thus making the work more efficient and reducing patient waiting time, all while saving the money. DABAI - Danish Center for Big Data Analytics Driven Innovation (*Dabai | Dabai*, no date), is used on a project in healthcare, and their primary product is a Forecast System, which is a machine learning system, that predicts the future based on the collected data from the past.

One usage of the forecast system is the product Patient Flow, which is a part of Systematics e-health solutions suite which helps clinicians and hospital staff forecast expected patient flow for a given ward in upcoming days (from one day to a week) by using machine learning techniques. This information helps them to see if the ward's capacity will be sufficient, whether they need some extra staff or whether they can afford to lay someone off.

Currently the Forecast System has only one machine learning system (for Region North) but expecting to expand to two. Two machine learning applications (one for each forecast use case), and several machine learning pipelines (one per forecast use case unit), which each trains a model at the end of the pipeline, then the forecast system will choose the best performing model to use for the real forecast. This hierarchy structure is illustrated in figure 1.

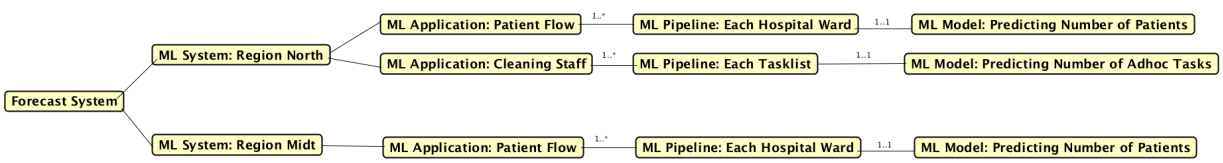
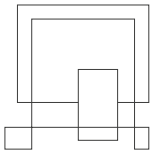


Figure 1 Forecast System Hierarchy



A hospital ward is a division of a hospital shared by patients who needs similar kind of care, e.g. department of cardiology.
A task list is a list containing all the cleaning task that is sorted to a certain group in the hospital.
It is crucial that such model performs on such level that clinicians can trust its forecast, for Patient Flow's case, without risking too few staff number, which will put patients at risk, or too many staff number, which will increase the cost for running the hospital. The current forecasting models already outperforms the hospital staffs' own estimates; however, Systematic engineers are already planning to upgrade the system to achieve even better results - furthermore there are even plans to extend the forecasting platform for use in other areas.
To know models and maintain system’s consistent, high performance, it is important to constantly monitor predictions and model metrics. This leads to the problem, PatientFlow eagerly wants to solve through this project. Currently there is no monitoring system for the forecast system at all, hence it brings great challenge for both debugging the forecast system and invoke model retrain procedure in time, before the model’s bad performance affects the usage and reliability of the system.

1.2 Definition of purpose

The project is aimed to build a tool for monitoring historical performance and status of the existing Forecasting System. Helping the user to keep track of the models used in the separate forecast pipelines and the accuracy of the forecast algorithms.

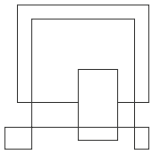
1.3 Problem statement

The main question to be answered in this project is:

- How can the performance of the forecast system be visualized?

To answer the main question, the following questions are to be answered:

1. What is the purpose of monitoring the forecast system?



2. What is the current solution for monitoring the forecast system?
3. What type of data will be monitored?
4. How should the data from the forecast system be retrieved?
5. How often should the new data be retrieved?
6. What type of graphical representation is needed to visualize the performance?
7. How should the performance data be stored?
8. How can the system be designed so it is extensible?
9. How can the system be implemented so it is maintainable?
10. What existing technology of similar purpose is already in use in Systematic?
11. What technology will be most suitable for the purpose?
12. What development method will be useful in this certain development context?

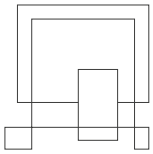
1.4 Delimitation

It is planned, that the current Forecast System is intended to integrate to other existing systematic systems in a close future. After the integration, the Forecast System should be extended to be able to handle forecasts for other type of data as well. According to Systematic, the Forecast System will contain many machine learning systems in a close future.

Due to resource limitation, monitoring of the newly integrated system will not be taken under consideration for this project. That is to say, the project will only focus on monitoring of PatientFlow data. However, the solution system will be designed with the quality of extensibility in mind, so it will be able to monitor the performance of other machine learning systems in the future as well.

Beside above delimitation, the security will not be considered under design and implementation neither, as Systematic has already working solution for authentication and authorization, that can be integrated into the product when needed.

The deployment of the product into production environment is also out of scope, as it is resource costly, which is not aligned with the project resources.



1.5 Choice of models and methods

This section describes the strategy chosen in order to answer the questions raised in the Problem statement (1.3). See Appendix B – Choice of Models and Methods.

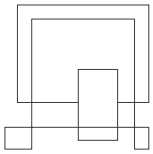
2 Requirements

Requirements were split into 5 categories and are referred to by their initials plus number in all the following section and appendices. (example: OS1: Overall structure requirement #1)

2.1 Functional Requirements

2.1.1 Overall structure (OS)

1. The solution should be designed for use by a Machine learning engineer or system admin in a hospital (technical minded person)
2. The system should have an interactive front-end in form as a web application
3. The solution system should be designed in such way, that it can easily be extended to monitor multiple forecast systems, which each contains multiple forecast pipelines at the same time.
4. The system should provide an overview of all the monitored forecast systems
5. For each monitored system, there should be an overview of all the monitored forecast pipelines
6. The solution should be able to inform about significant changes in prediction performance. The significant change is defined by:
   6.1. Comparing the historical value of MAE against current value
7. The system should be able to provide a way (possibly via button click) to trigger a new training process for one specific pipeline

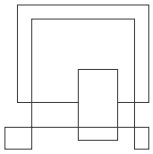


2.1.2 Prediction performance (PP)

8. The solution system should be able to display the Prediction Performance (in form as different evaluation metrics, only MAE score is agreed upon for this project) of the historical forecast compared to the historical result in intervals that is configurable, but two weeks as the default.
9. The system should be able to display an evaluation plot, comparing the forecast result and the actual result in intervals that is configurable, but two weeks as the default, with two different series in one graph.
10. The system should be able to display the feature metrics based on the training data retrieved from the forecast system.
11. The solution system should be able to display performance information in a sorted manner, so the presentation of Region, Client and Unit are ordered by MAE in descending order.
12. The solution system should be able to indicate current prediction performance in color code, so it satisfies the following:
    - 12.1. Each unit has a color indication for its current prediction performance compared to standard deviation of historical MAE distribution, where:
      - 12.1.1. Red - indicates $MAE \geq 3std$
      - 12.1.2. Yellow - indicates $1.5std \leq MAE < 3std$
      - 12.1.3. Green - indicates $MAE < 1.5std$
      - 12.1.4. Gray – indicates not enough data
    - 12.2. Clients has the same color indication as its worst performing unit
    - 12.3. Regions has the same color indication as its worst performing client
13. The solution system will automatically unfold any client with red performance indication, while the rest remains folded.
14. The solution can display performance of all units within a client in heatmap

2.1.3 Feature specifications (FS)

15. The system should be able to visualize feature importance's in a bar plot



16. The system should be able to visualize different feature value in either continuous or discrete format against frequency (distribution of feature value) appeared as histogram, this should be applied to both training data and real-time data

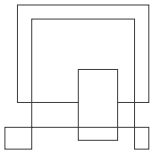
2.1.4 Model information (MI)

17. The system should be able to present the current active model for each pipeline, and it should provide this information about the model:
    17.1. Unit name
    17.2. Creation time
    17.3. Back-testing MAE
    17.4. Number of back-testing data instances
    17.5. Train MAE
    17.6. Number of train data instances
    17.7. Test MAE
    17.8. Number of test data instances

2.2 Non-Functional Requirements

2.2.1 Technology requirements (TR)

18. Every system component in the solution system should be stateless
19. Client should be implemented in a stable version of Angular (*Angular*, no date)
20. Data visualization library/framework used should preferable be free



3 Analysis

Analysis seeks to explain the problem domain at a deeper level to provide a decision base for designing and implementing the solution system for the problem. Answers to following questions from the problem formulation will be presented in the section specified inside the parentheses:

- What is the purpose of monitoring the forecast system? (3.1)
- What is the current solution for monitoring the forecast system? (3.1)
- What type of data will be monitored? (3.2)
- How should the data from the forecast system be retrieved? (3.3)
- How often should the new data be retrieved? (3.3)
- What existing technology of similar purpose is already in use in Systematic? (3.4)
- What technology will be most suitable for the purpose? (3.4)
- What development method will be useful in this certain development context? (3.5)

3.1 Forecast System monitoring purpose and current approach

Currently the forecast system is not monitored in any automated way, whenever monitoring is needed for debugging purpose, manual inspection of the log files and the databases are performed by the developers. This means it is both fault prone and time consuming to monitor a certain model, therefore the current approach is to retrain a new model every day, instead of keeping existing models alive. Though retraining a new model is very CPU resource consuming, and normally one only need to retrain a model, if the data changed significantly or forecast is degrading since last time. Furthermore, the current approach for model retrain is also done manually, hence saving it away will also be cost beneficial.

If the model is to be let live for longer, then the challenge will be to maintain the fine balance of keeping a model alive and retraining new model when needed. Hence the purpose for monitoring the forecast system is to provide fast feedback for how well a model is forecasting, furthermore notify for model retrain as soon as needed.
Based on the purpose studied above and the requirements demanded by the stakeholder, nine use cases are deducted. They can be viewed on the use case diagram in the figure 2.

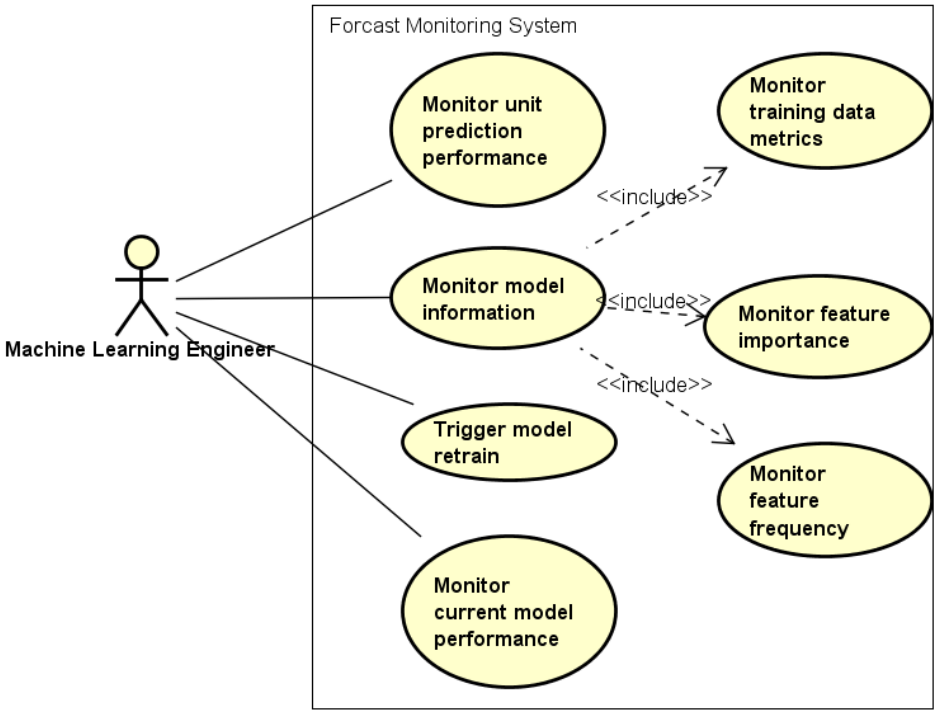
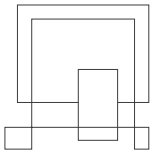


Figure 2 Use Case Diagram

Figure 3 shows the use case description for one of the use cases in the system, Monitor Unit Prediction Performance. The full use case description for all the use cases can be viewed in Appendix G – Use Case Descriptions.



Monitor unit prediction performance / UseCase Description	
ITEM	VALUE
UseCase	Monitor unit prediction performance
Summary	User can monitor predictions and actuals for a certain unit in intervals that is configurable, bu t two weeks as the default backwards as two different series in one graph.
Actor	Machine Learning Engineer
Precondition	
Postcondition	
Base Sequence	1. The user clicks on a certain ML pipeline to access the unit information page. 2. The user can monitor predictions and actuals for a certain unit in intervals that is configura ble, but two weeks as the default backwards as two different series in one graph.
Branch Sequence	
Exception Sequence	
Sub UseCase	
Note	

Figure 3 Use case description for Monitor Forecast Units

3.2 "In which format" is the data monitored

There are two types of data to be monitored: large quantity of numeric data, which consists of: training MAE, the predicted data and the actual historical data; and limited amount of model informational data of: algorithm name, unit name, algorithm age. Though regardless of the quantity, the common for both

types is they are in the JSON, JavaScript Object Notation, format. Figure 4 shows the source data schema from the Forecast System.

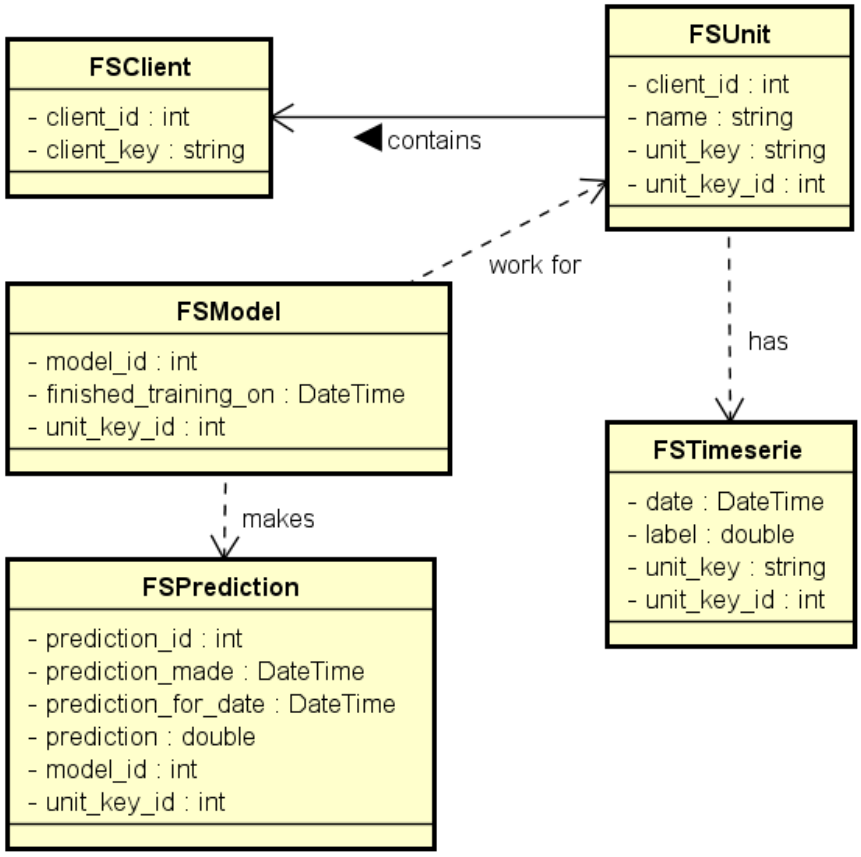
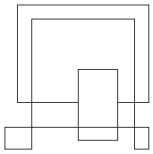


Figure 4 Forecast System data schema

The FSClient object type corresponds to the ML Application in Figure 1 - Forecast System Hierarchy. The object of FSClient can either exists as independent objects queried directly from the Forecast System or as embedded object in the FSUnit. Hence the relationship is described as dependency contains instead of an aggregation. The client_key is a system unique string, containing human readable name of the client, e.g. PatientCapacity, while the client_id is a system unique integer, used as database primary key in the Forecast System.

The FSUnit object type corresponds to the ML Pipeline in Figure 1 – Forecast System Hierarchy. It represents one smallest unit/ward in a hospital. The name property contains the name of the unit, that is understandable to the user, e.g.



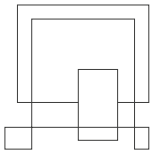
Infektionsmedicinsk - 7Ø, but not necessarily system unique. The unit_key is a system unique string that uniquely represents the unit, e.g. 8389C322-A19B-40A4-9E29-37389353644B_PatientsAtDepartment, while the unit_key_id is an integer used as database primary key in the Forecast System.
The FSModel object corresponds to the ML Model Pipeline in Figure 1 – Forecast System Hierarchy. It has a unique id, model_id, a birth timestamp, finished_training_on, and it has to be linked to a certain FSUnit.
The FSPrediction object represents a prediction made by one FSModel for one specific FSUnit for a specific time - prediction_for_date. It is noteworthy, that the actual prediction value is a double.
The FSTimeserie object represents an actual historical value for a specific FSUnit reported by the hospital staff at a point in time, in current use case this label represents either the amount of occupied capacity or the number of ad-hoc cleaning task. The label value of a FSTimeserie can be updated in the future in case of error correction.

3.3 "How" should the data be monitored

Forecast system currently has a RESTful API for data extraction, which provides some of the data to be monitored, but DABAI team had promised to extend the API to provide all the data needed for this project to happen. For a short resumé of how the Forecast endpoint looks like, see Appendix D – Forecast System API Description. The forecast system will be provided with new data points from the production system every 10 minutes, which means the MAE metric for the certain prediction will also change every 10 minutes. Hence as a real-time monitoring system, the data retrieval from the forecast system should also be every 10 minutes.

3.4 Technology for the purpose

In the health care business unit in Systematic, C# is the most used language for server-side programming, hence C# is a natural choice for the purpose, as the



project has competence in house to maintain and further develop a C# system. AngularJS is currently used for front end of the Forecast system, though an upgrade to Angular has been planned in the predictable future, hence for the GUI programming, Angular will be the most obvious choice. This will be elaborated more in the design section.

3.5 General Project Development method

Systematic has rich history in using Feature Driven Development (Development, 2003) for all their development tasks in general, and FDD has also shown its advantage in quality assuring the development result by ensuring a synchronized cooperation between the stakeholders, developers and tester (Systematic A/S, 2018). Hence using FDD for the project is a very natural choice following the company culture. The activities of FDD conducted in this project and the sequence for how they are used, is illustrated in figure 5. A description of the artifacts of the activities “Build feature list” and “Plan by feature” will be presented in the following section, while the artifacts of the activity “Build by feature” are the in the Appendix J - Feature Description Documents.

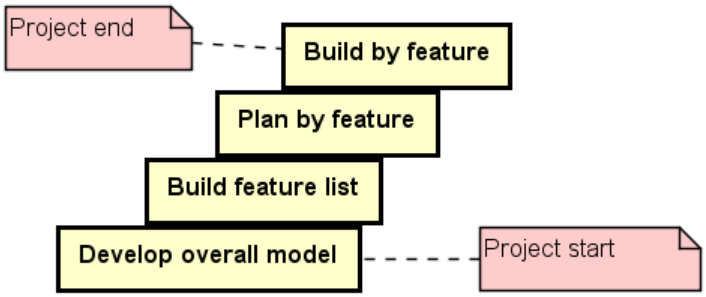
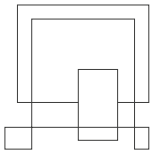


Figure 5 FDD activities used in this project

Furthermore, Scrum (Schwaber and Sutherland, 2013) is chosen to quality assure the development process, to assure a good dynamic in the team and development speed considered unpredictability.



3.5.1 Feature List

Based on the principle of feature driven development, a feature list with ten features is defined based on the knowledge of the problem domain gained throughout the introduction and analysis section. Each item of the feature list consists of a feature description, which describes the overall goal with the certain feature, and a list of requirements, which will be addressed in the feature scope. Figure 6 illustrates, how one of the more complicated features in this project, FDD0001 Display Unit Information, is defined in the feature list, the complete feature list can be found under Appendix A – Feature List.

FDD0001 - Display unit information

- Feature description: User have overview of the given pipeline/model statistics
- OS5: For each monitored system, there should be an overview of all the monitored forecast pipelines
- PP1: The solution system should be able to display the Prediction Performance (in form as different evaluation metrics, only MAE score is guaranteed supported in this project) of the historical forecast compared to the historical result in intervals that is configurable, but two weeks as the default backwards.
- PP4: The solution system should be able to display performance information in a sorted manner, so the presentation of Region, Client and Unit are ordered by MAE in descending order.
- PP5: The solution system should be able to indicate current prediction performance in color code, so it satisfies the following:
  - a. Each unit has a color indication for its current prediction performance, where:
    - i. Red - indicates MAE≥3std
      ii. Yellow - indicates 1.5std≤MAE<3std
      iii. Green - indicates MAE<1.5std
  - b. Clients has the same color indication as its worst performing unit
  - c. Regions has the same color indication as its worst performing client
- PP6: The solution system will automatically unfold any client with red performance indication, while the rest remains folded.

Figure 6 Example of a feature list item – FDD0001 Display Unit Information

3.5.2 Feature Roadmap

Feature roadmap resolves the dependency between the features, and hence support drawing the idea of in which sequence the features can be build.
The roadmap itself can be found in Appendix C – Feature Roadmap.

4 Design

This section documents the considerations behind the design of the solution system. In the health care business unit in Systematic, C# is the most used language for server-side programming, hence C# is a natural choice for the purpose, as the project has competence in house to maintain and further develop a C# system. Angular is chosen for the front end because it is already used for front end of the current Forecast System. (read more at 5.3 ForecastMonitor UI) The design thus reflects upon those technological constraints. Considering also the result of the analysis, the solution is designed as:

- Angular single page application (SPA) - ForecastMonitor UI
- ASP.Net Core RESTful service - ForecastMonitor Service

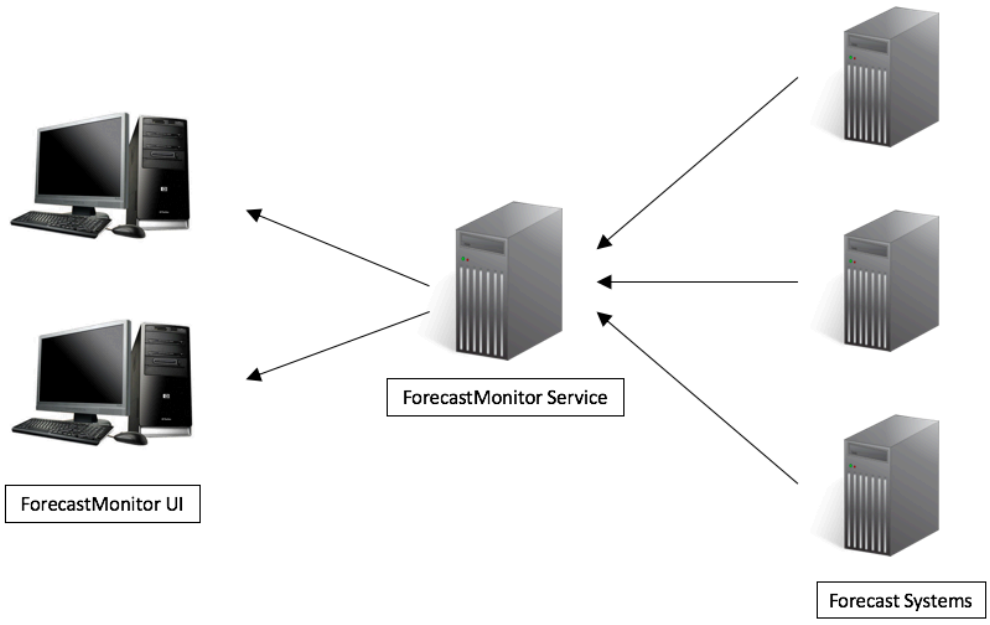
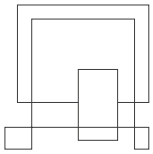


Figure 7 ForecastMonitor Design

The section continues with general design principles followed by detailed descriptions of both ForecastMonitor Service and ForecastMonitor UI, respectively. Note that the sketch lacks a persistent data storage - a memory storage is used instead.



4.1.1 SOLID Principles in Mind

To assure maintainability, extensibility and a generally understandable and clean code, the service had SOLID principles (Robert C. Martin, no date) in mind through the whole design process.
These 5 principles are:

- **Single responsibility principle** – A class should have one, and only one, reason to change. The principle aims to keep the overall design clean and generally easy to understand. An example of usage of the principle could be a design of performance calculation feature (FDD0002), when rather than simply extending the job for Forecast System data retrieval by immediately evaluating unit performance, performance evaluation logic is in another job which spans after data retrieval job completes its work. Both jobs thus serve a single responsibility.
- **Open/Closed principle** – A class should be open to extension by adding new behaviors without a need to modify it. The way the service for scheduled (periodical) job is setup demonstrates the principle. It takes a list of jobs as a constructor argument and executes each jobs ExecuteAsync method when based on its Schedule. The logic for each specific job is executed by that method, so the scheduler can execute any logic regardless the job type. In other words, behavior of the class can be extended by running any job type.
- **Liskov substitution principle** – Functions that use pointers or references to the base classes must be able to use objects of derived classes without knowing it. The principle is used as a safety whenever the design involved inheritance. For example, the figure 8 shows that the Service makes use of two distinct job types – ad-hoc and scheduled job. It would make no sense for an ad-hoc job to have a schedule, however both jobs do inherit the same ExecuteAsync method.

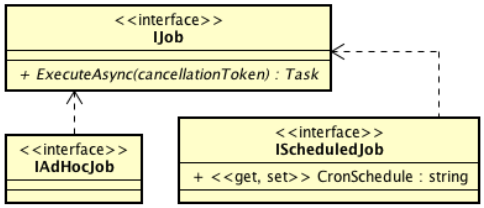
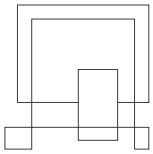


Figure 8 Job types class diagram

- **Interface segregation principle** – Clients should not depend on interfaces that they do not use. It's a principle closely related to the single responsibility one, and



the goal is to define interfaces specific enough and make their implementations use all its methods. Example from the previous principle shows that both ScheduledJob and AdHocJob implementations both satisfy this principle and maintain a high cohesion.

- **Dependency inversion principle** – A class should depend on abstractions rather than concrete implementations**.** Services in ForecastMonitor Service have interface-based constructors and are registered via dependency injection mechanism. The technique adds extra flexibility and is particularly helpful in testing, by allowing to setup custom dependencies and isolate the class under the test.

4.2 ForecastMonitor Service

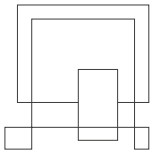
It is a self-contained, logically separated unit, with following responsibilities:

- consume pre-configured FS instances
- store in the memory, and periodically update the data needed for the UI
- perform unit evaluation logic after each update
- provide REST API for the UI

This part of the design was inspired by Martin Fowlers Patterns of Enterprise Application Architecture (Fowler *et al.*, 2002), and unless specified explicitly, patterns described in this section will originate from than publication.

4.2.1 Service Oriented Architecture (SOA)

The solution is to be integrated into an existing software suite which was already designed using SOA (*Service-oriented architecture (SOA)*, no date) and ForecastMonitor Service continues to do so. It's not only due to SOA being a constraint but also because of advantages it offers. The beginning of this section already mentioned a RESTful web service, which is also one of the most common implementations of SOA. Naturally, following SOA advantages and disadvantages are relevant to the RESTful services as well. Each point is also evaluated in connection with the ForecastMonitor.

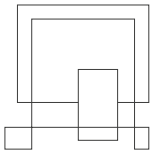


Disadvantages:

- **Extra Network Overhead** – When services interact, they always perform a complete validation of every input parameter. Performance is not an issue for a dashboard application.
- **Extra Investment Cost** – SOA requires extra investments in technology, development and human resource. All the technology was already available, and the solution was designed small enough to fit the development team size.
- **Not Suitable for GUI Heavy Applications** – Increased GUI complexity would require heavy data exchange. ForecastMonitor UI is a dashboard with UI cut to the minimum, which makes the disadvantage irrelevant.

Advantages:

- **Reusability** - SOA allows to assemble small, loosely coupled pieces of functionality. Therefore, services can be reused in different applications. ForecastMonitor Service is intended to be used in single application at the moment but might be useful in case a new UI is planned.
- **Maintainability** – SOA is an independent entity, so it can be extended, updated, or changed without having to change any other healthcare services. This adds a real value since not all members had access to Systematics source codes and any changes (apart from promised extensions to the Forecast System) to any existing systems were undesirable.
- **Reliability** – SOA applications are more reliable since smaller independent services yield less code and can be easier tested and debugged. In the case of ForecastMonitor, having UI separated from business logic has shown to be a great advantage especially in the testing – which is covered in later sections. Having UI separated also helps in the overall readability.
- **Location Independence** – SOA can be published to any directory where consumers can access them. Both ForecastMonitor Service and UI are deployed on the same IIS but having an option to run them locally is helpful, particularly during the development.



- **Scalability & Availability** – multiple instances of SOA applications can run on different servers. ForecastMonitor Service is implemented as a REST service, thus is stateless, and a single installation can serve multiple users, but the solution being a dashboard, this advantage does not play a significant role.
- **Platform Independence** – SOA facilitates a product by integrating different technologies, from different vendors. Even though Forecast System is implemented in Python, ForecastMonitor Service has no problems consuming it. The same goes for JavaScript ForecastMonitor UI and Service

4.2.2 General Overview

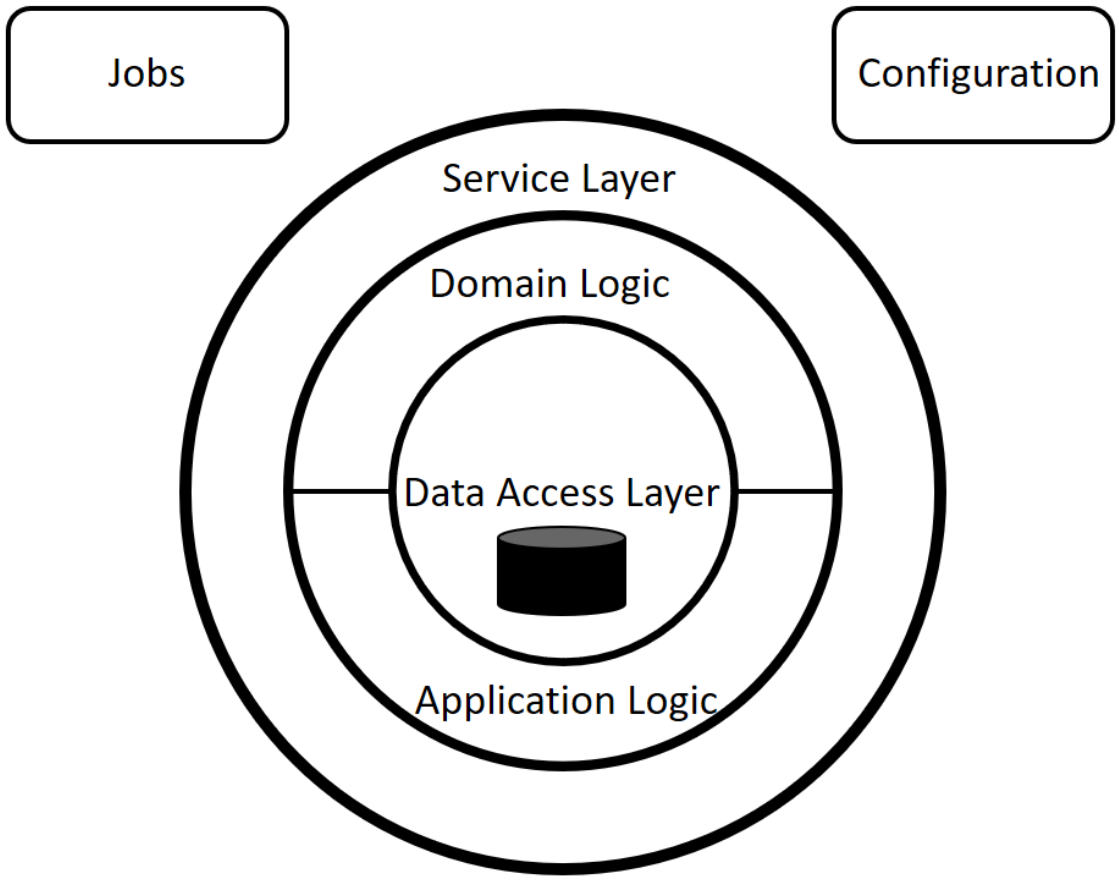
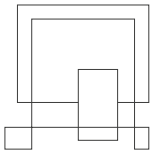


Figure 9 ForecastMonitor Service - General overview (Service Layer pattern)



The general structure is dictated by Service Layer pattern. The Service contains logic for different purposes, and the pattern groups it into different layers. The naming convention for interfaces (and the logic they implement) intended to interface across layers or used in DI uses a “service” suffix – to maintain the same naming convention with .Net Core DI container. There is a brief description of each layer (each layer has its own subsection later in the section, starting by innermost layer in outward direction):

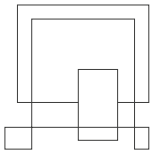
- **Data Access Layer** – contains services for data access, including Forecast System consumer, and implementations of Data access objects
- **Business Logic Layer** – as many prefer, including Martin Fowler himself, layer is split into two separate sub-layers:
    - Domain Logic – logic purely concerning the problem domain - unit performance evaluation strategies
    - Application logic – application responsibilities logic, such as service publishing new units to the UI or providing plot data
- **Service Layer** – defines an application’s boundary and its set of available operations (by using services from Application Logic), or in other words, a layer with Web API Controllers and Data transfer objects

The two extra areas are:

- **Jobs** –tasks intended to run in the background and services responsible for executing them, with both scheduled (periodical) tasks such as CacheUpdateJob, or ad-hoc tasks which run upon request, such us publishing of units once newly arrived data went through performance evaluation process
- **Configuration** – contains logic for middleware setup and a service for reading configuration file

The reason why the latter two are in its own area rather than being assigned to one of the layers is that both are either used in or handling logic of multiple layers.

There are two known implementation variations: domain façade, or operation script approach, and the Service uses the former one – Service Layer implemented as a set



of thin facades without any business logic. Opposed to the latter (thicker Service Layer with implementation of business logic), It keeps the layer more cohesive and understandable.

4.2.3 Data Access Layer

Consists of Forecast System consumer, memory-based data storage and its access services, including ORM model.

4.2.3.1 Forecast System Consumer

Consumer has three parts:

- ForecastSystemClient – Http client with endpoints to the Forecast System
- ForecastSystemService – wrapper of the client with enhanced retrieval logic (used by CacheUpdateJob)
- Data transfer objects – types received from Forecast System

Note that each installation needs a compile time registered instance of the ForecastSystemService – this can be achieved via a DI container. Splitting retrieval logic between two interfaces gives an advantage in cases of:

- If Forecast System changes its API or endpoints, only the client class needs to be changed
- If the solution would need a different way of interaction with Forecast System, it’s enough to change the service

The figure below shows simplified diagram (full diagram present in Appendix I - ForecastMonitor Service Architecture) for both client and service.

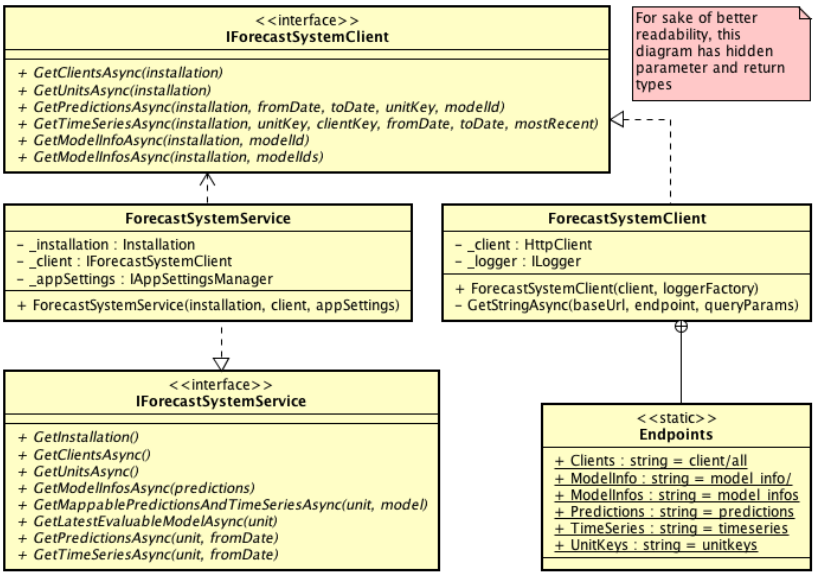


Figure 10 Forecast System consumer class diagram

4.2.3.2 Memory Data Storage

The Service uses memory storage instead of a persistent one and takes advantage of .Net Core's out of the box support of key-value based memory caching via IMemoryCache (*Cache in-memory in ASP.NET Core | Microsoft Docs*, no date) interface.

A permanent storage was ruled from following reasons:

- Stakeholders expressed a wish to do not make a "copy" of Forecast System data
- Forecast System already contains all the data needed for visualization, the Service only needs to transform it
- The amount of data that the Service needs to store, and access allows memory storage

Caching into the memory also saves a significant amount of development time. Memory part of the data storage is abstracted to the point that a different choice in the future remains open. Class diagram below shows how are data services, IDataContext and IMemoryCache related.

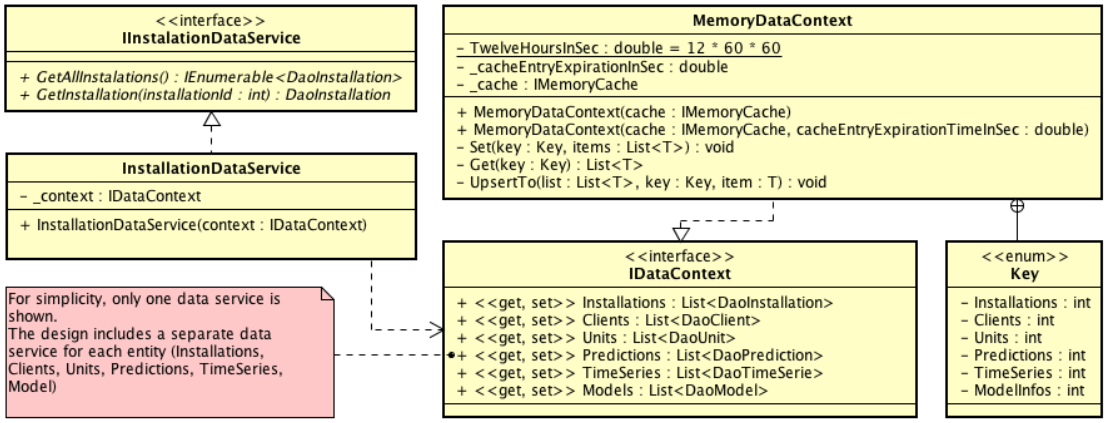


Figure 11 Memory storage class diagram

MemoryDataContext implements getters for all properties by use of its generic private getter and a key – dictionary-like nature of IMemoryCache requires one. Setting is done with use of UpsertTo (update or insert) method, so no data loss occurs during the Set operation. All three generic methods are with constraint to IDao interface (more about IDao in Data Model subsection). The cache also invalidates each entry by its expiration time (12 hours by default) and no clean-up is required. Sequence diagram below shows the Set method.

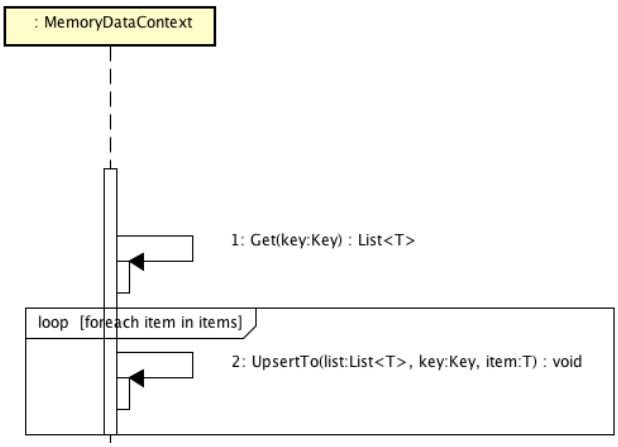
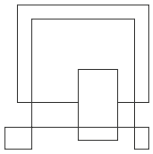


Figure 12 MemoryDataContext Set operation sequence diagram

4.2.3.3 Object Relational Mapping

IMemoryCache can't create a relational data model, but data access objects are still related and need some object relational mapping. The IDao interface provides it by method KeysEqual and a class diagram below enriched by ER multiplicity shows how



to implement it. This way IDataContext dependent services can act as if IDataContext was an ORM provider. KeysEqual method is also intended to be used during Upserting.

Note that Dao prefix is simply a naming convention (used in Systematic) and it is not intended to follow the DataAccesObject pattern. Each object is simply a POCO class without any logic, apart from the method in IDao interface.

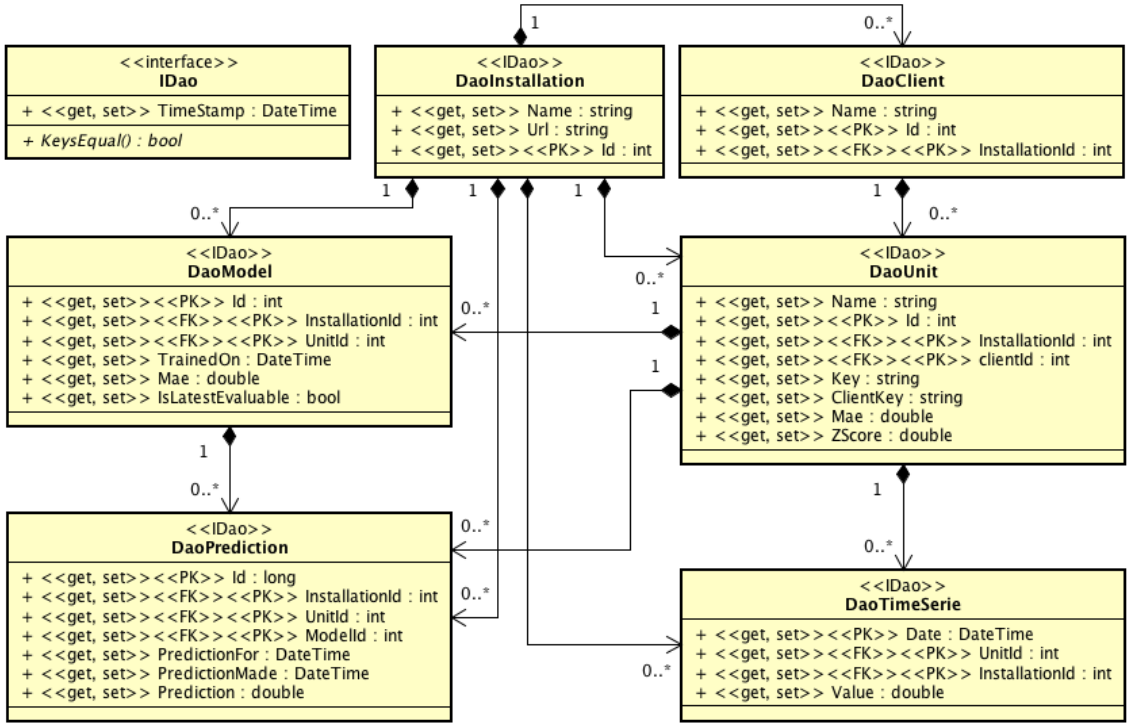


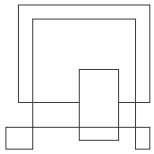
Figure 13 Data Model

4.2.4 Business Logic Layer

As mentioned earlier, business layer is split into two subdomains. The benefit is that area which creates the value for the product is separated from application's internal logic and makes the design more organized and well-structured.

4.2.4.1 Domain Logic

Contains logic concerning the problem domain - the performance calculation logic.



It implements logic for unit and model evaluation and is used within PerformanceCalculationJob. Figures below show PerformanceCalculationLogic class diagram and sequence diagram for MAE (ZScore is available in Appendix I).

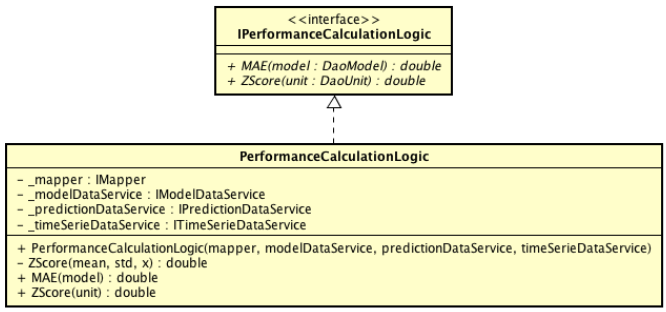


Figure 14 PerformanceCalculationLogic class diagram

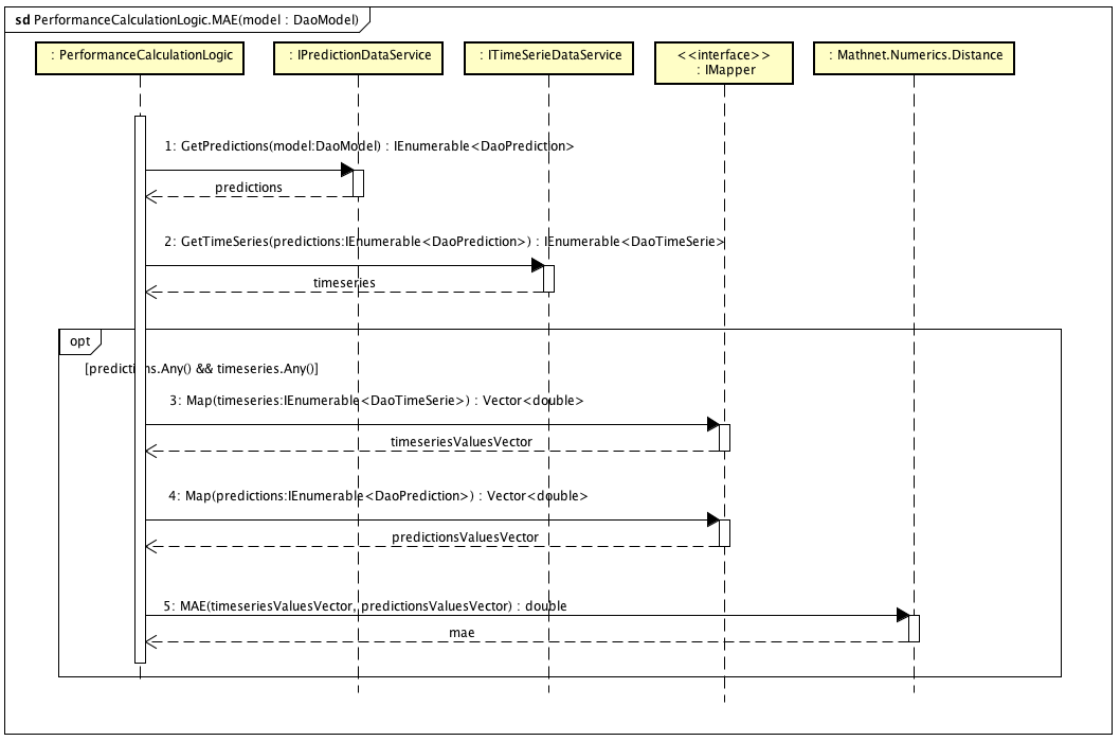
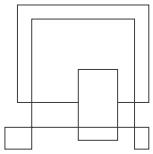


Figure 15 PerformanceCalculationLogic.MAE sequence diagram



4.2.4.2 Application Logic

Holds application responsibilities logic. It holds logic for each controller from the Service Layer and a SignalR depending service for publishing unit updates over web sockets (*Introduction to ASP.NET Core SignalR | Microsoft Docs*, no date).
It was mentioned earlier that the solution only needs to retransform data arriving from Forecast System and perform some evaluation logic. Evaluation logic is performed every time (after CacheUpdateJob completes) by PerformanceEvaluationJob and results are stored in the cache, so the only remaining responsibility is to communicate with DAL and map its responds into data transfer objects. Transforming is done via AutoMapper (IMapper interface). There are three application logic interfaces:

- IClientLogic
- IInstallationLogic
- IUnitLogic

The example below shows ClientLogic class diagram, rest is in Appendix I – ForecastMonitor Service Archtitecture.

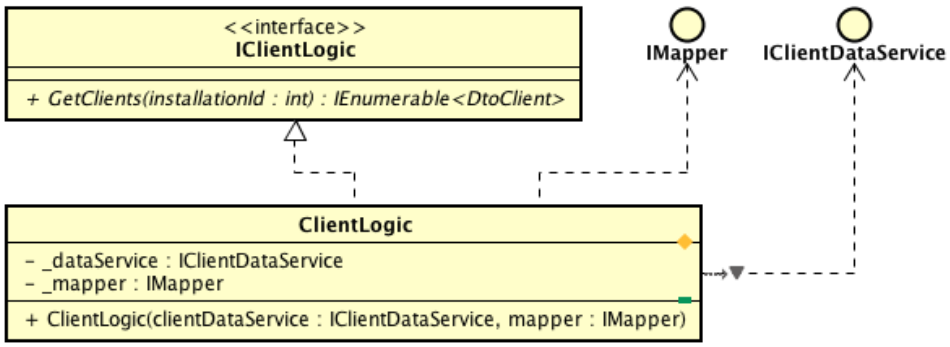


Figure 16 ClientLogic class diagram

Beginning of the section mentioned SignalR component – UnitPublishingService. SignalR makes developing real-time web functionality easy and allows to push content to connected clients instantly as it becomes available. Having a real-time functionality in a dashboard app is not a requirement, especially when the backend has no real-time data access. However, it's API is so simple that using it saves more time than the other alternative – making client pooling for new data in some fixed, pre-defined interval. Figures below shows both UnitPublishingService class diagram and its publishing logic sequence diagram – PublishUnits method.

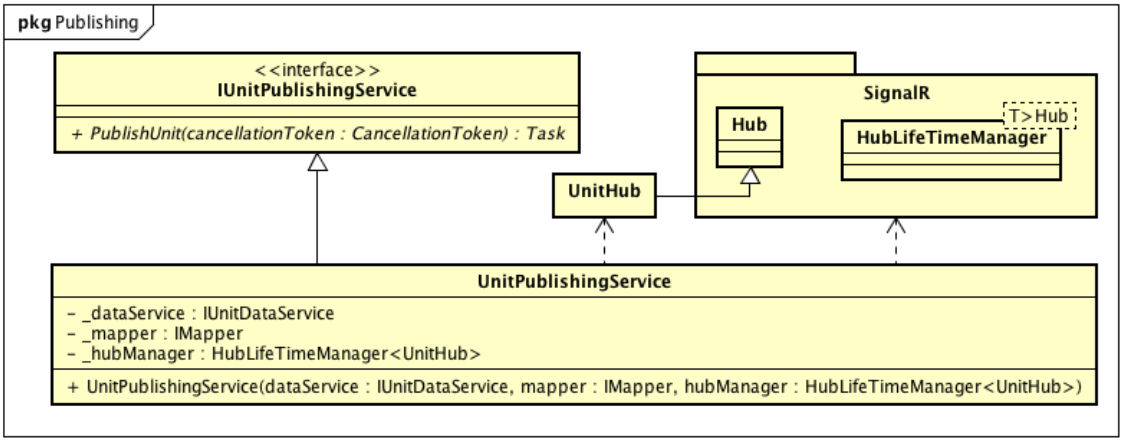


Figure 17 UnitPublishingService class diagram

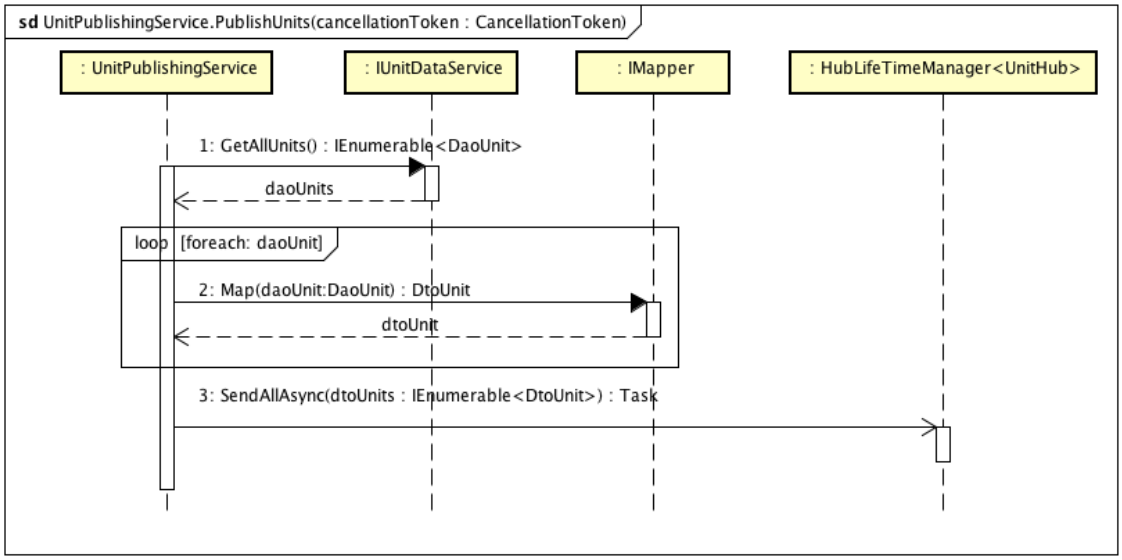


Figure 18 UnitPublishingService.PublishUnits sequence diagram

4.2.5 Service Layer

Since the pattern is implemented in a domain façade variation, the layer is kept logic-less and contains only controllers with route (API) definitions. Each controller inherits from ControllerBase class and uses application logic interfaces to perform its operations. Serialization is done automatically via middleware.

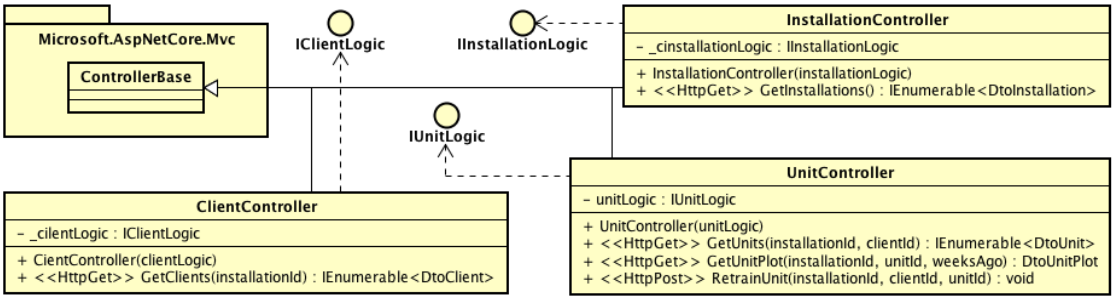


Figure 19 Service Layer class diagram

The Service defines 5 GET requests. The full API design is available as Appendix H – API Design, the figure below shows GET request for unit plot.

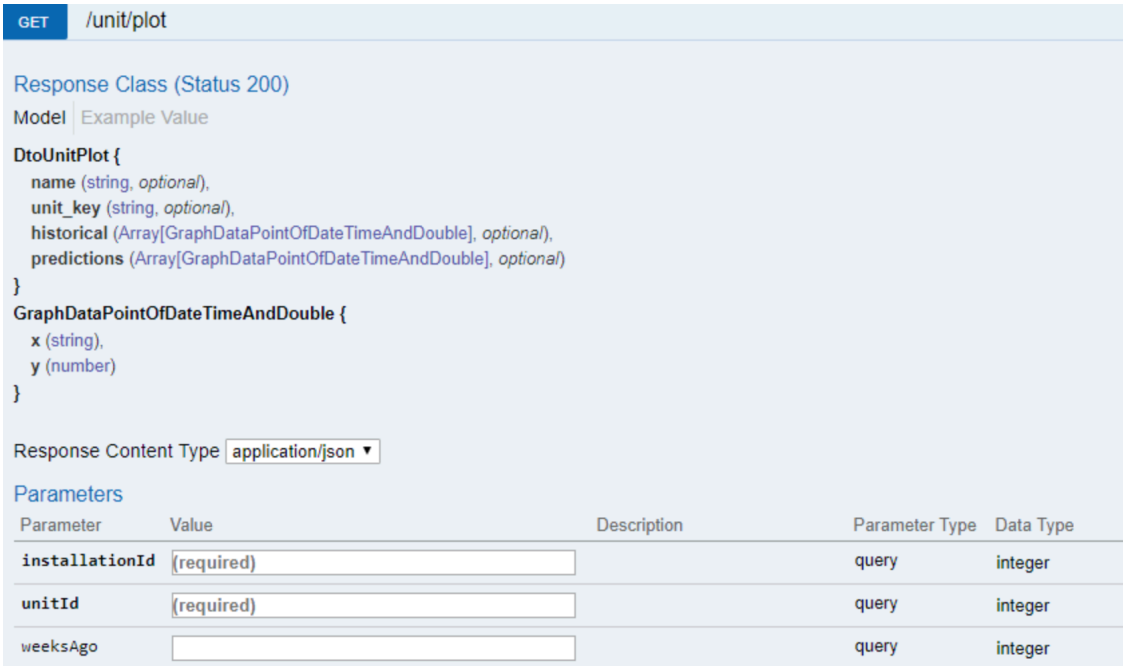
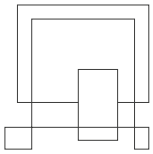


Figure 20 API endpoint example

4.2.6 Configuration

Contains middleware setup and an appsettings.json (.Net Core configuration file) reader service, used across other layers. AppSettingsManager provides a concrete implementation for every configuration entry, by consuming.Net Core's IConfiguration interface, which can read the configuration file section by section and cast results into



objects (*IConfiguration Interface (Microsoft.Extensions.Configuration) | Microsoft Docs*, no date).

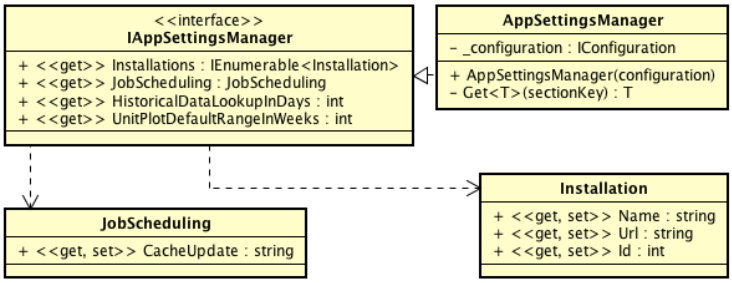


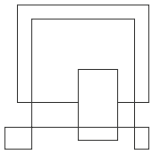
Figure 21 AppSettingsManager class diagram

4.2.7 Jobs

Jobs are classes with some pre-defined execution logic, which is meant to run in the background, without user interaction. There are two types of these jobs (periodical/scheduled & ad-hoc) and the namespace also includes two background services that can execute them:

- ScheduledJobService
- AdHocJobService

Both services are dependent of .Net Core IHostedService interface (*Background tasks with hosted services in ASP.NET Core | Microsoft Docs*, no date) – implementation of which enables to run application logic in the background. Downside of using hosted services is that they (including their jobs) need to be registered by DI container as singletons, which limits their ability to put any lower-scoped services in their constructor and must access them within their own scope. A UML class diagram would not provide enough information on how the logic should be implemented or which services should it use. Each job then needs its own execution sequence diagram.



4.2.7.1 Jobs & job chaining

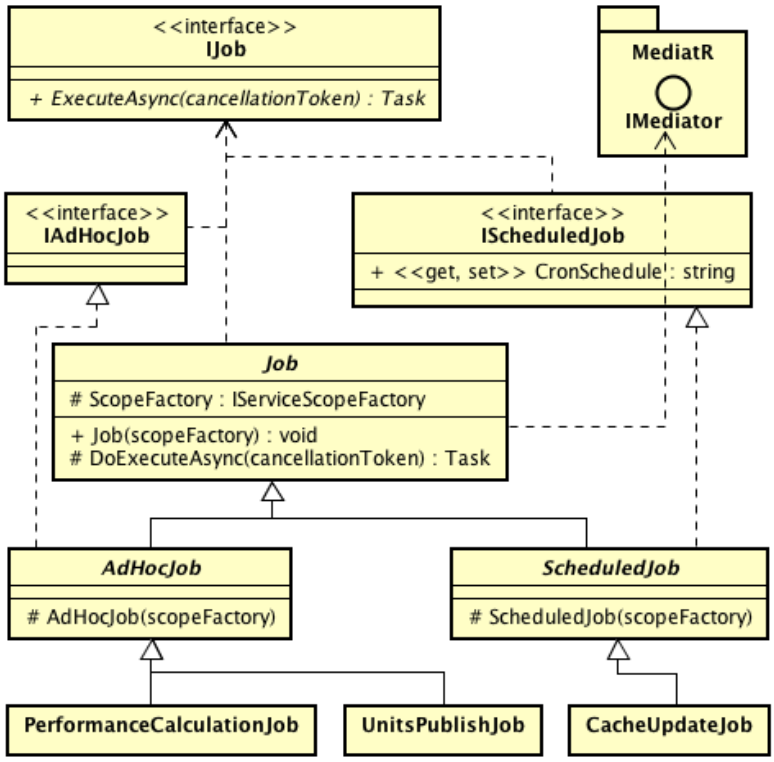


Figure 22 Job class diagram

To keep track of finished jobs, every job inherits from an abstract Job class, whose implementation makes use of Mediator pattern and informs JobNotificationHandler about its completion once its execution logic is done. Handler is then able to execute other logic and the Service uses it to make a job sequence without the need of adding complexity to job services or coupling jobs together.

The sequence is:

1. CacheUpdateJob (ScheduledJob, every 10 minutes)
2. PerformanceCalculationJob (AdHocJob)
3. UnitsPublishJob (AdHocJob)

This way the dashboard gets new units as soon as new data arrives and performance calculation completes. Figures below show a job execution sequence diagram and JobNotificationHandler class diagram.

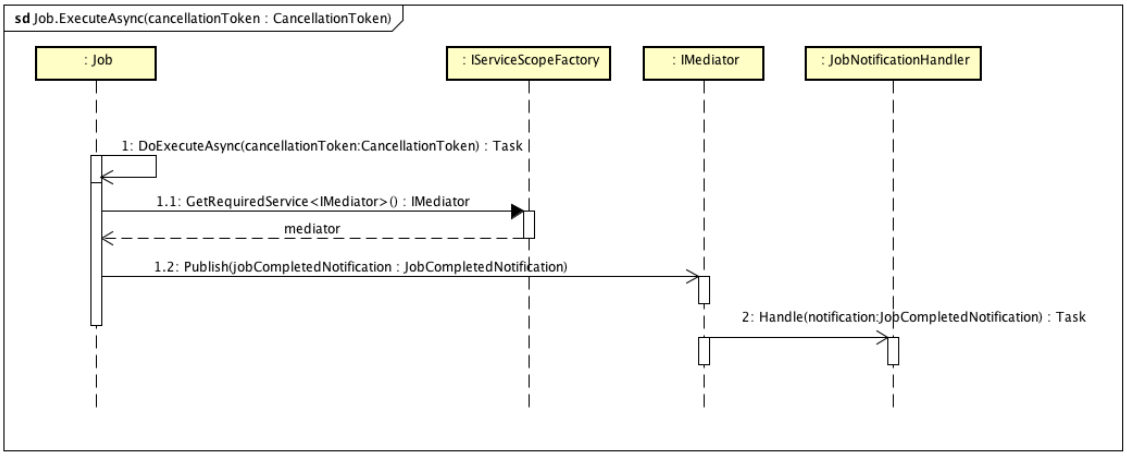


Figure 23 Job.ExecuteAsync sequence diagram

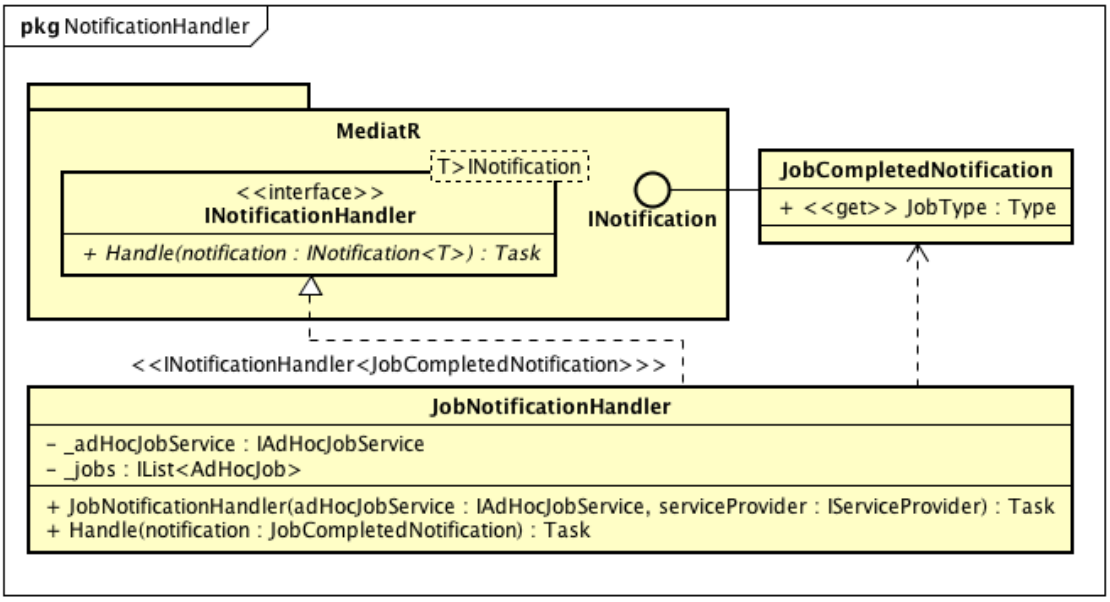
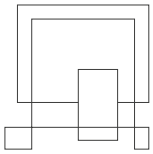


Figure 24 JobNotificationHandler class diagram

4.2.7.2 ScheduledJobService

The primary reason for the service is that the backend does not use any persistent data source and a solution to provide data for UI is to retrieve data from Forecast Service(s) on startup and repeat the execution with certain period (ScheduledJob). Earlier section has already shown the ScheduledJob class. It uses a Cron based schedule, intended



to be read from configuration file. This way the user can decide, how often should CacheUpdateJob run. ScheduledJobService is an implementation of IHostedService which handles execution of scheduled (periodic) jobs, pooling them in one-minute interval. There is only one scheduled job implementation to date – CacheUpdateJob, but the design supports larger number in the case of future extensions. The design of ScheduledJobService is based on implementation guidelines from Microsoft and Maarten Balliauw's article for Cron-style scheduled jobs (Balliauw, no date). The constructor wraps every IScheduledJob into JobWrapper, parses its schedule and sets NextRunTime to the current time, so every registered scheduled job runs on application start. ExecuteAsync runs on startup and executes its private ExecuteOnceAsync with one-minute period. ExecuteOnceAsync looks for a job which is due to run and executes it if such job exists. It calls the Increment method before prior job execution, which will set LastRunTime to the current time and NextRunTime to the time of the next scheduled occurrence.

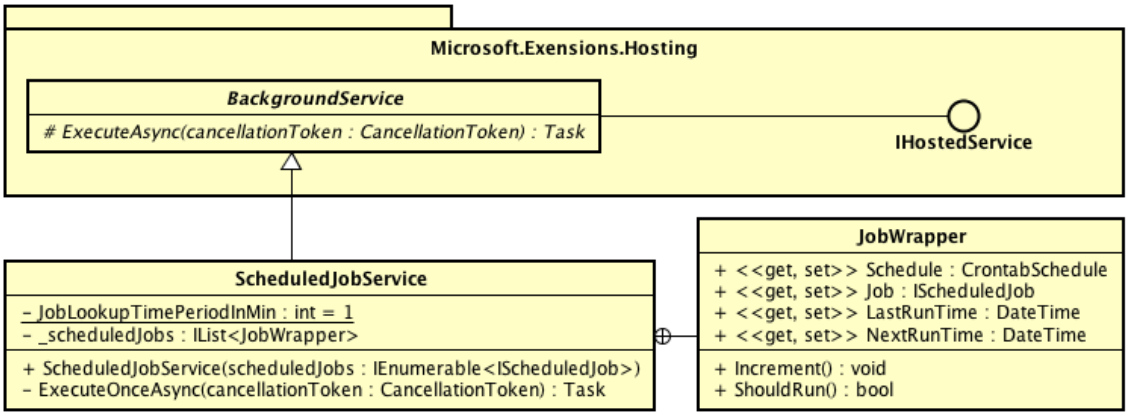
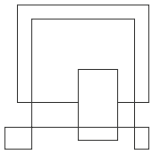


Figure 25 ScheduledJobService class diagram

4.2.7.3 AdHocJobService

AdHocJobService is another way of implementing IHostedService and handles execution of ad-hoc jobs. Design again follows Micosoft's official implementation guidelines. Service makes use of JobQueue and executes AdHocJobs in it in FIFO order, or waits while the queue is empty. Its ExecuteAsync method contains an ifnitine loop which starts with a request for a job from JobQueue.DequeAsync method –



JobQueue uses semaphore and its result is blocked until the queue contains a job. The job is then executed.

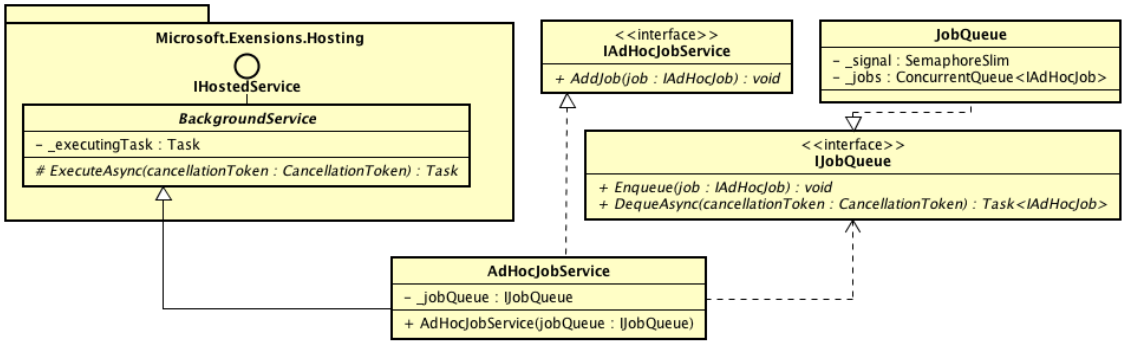


Figure 26 AdHocJobService class diagram

4.3 ForecastMonitor UI

The framework chosen for the frontend is Angular 6. When discussing the project idea with the Product owner it was decided that Angular will be used for the frontend since the current system (Patient Flow) is developed in AngularJS and an upgrade to Angular is planned, so the team will also be able to continue the development of the Forecast Monitor if necessary. For creating and maintaining the frontend application Angular CLI was used because it provides an easy way to generate different parts for the application like components, interfaces, services, etc. And it also keeps the project structured correctly. The system is modular, combining small components into the whole application.

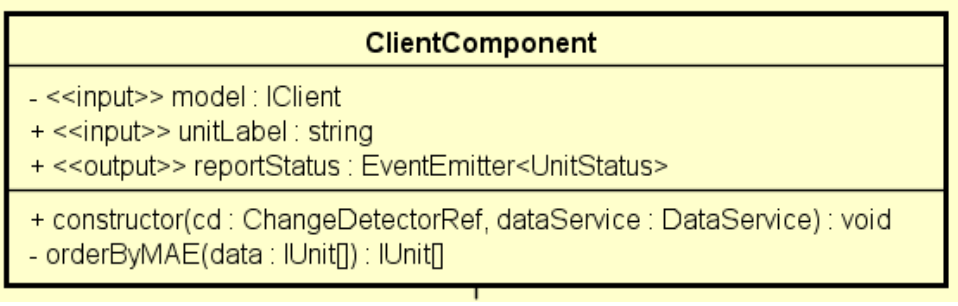
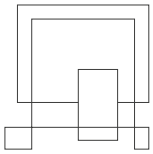


Figure 27 Client Component class diagram



Full UI system architecture diagram in Appendix F – ForecastMonitor UI System Architecture.

4.3.1 Design patterns

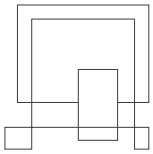
Angular is designed to be used with design patterns, and especially with ones that are not so commonly used like reactive programming and unidirectional data flow.

4.3.1.1 Reactive programming

Reactive programming is a declarative programming paradigm concerned with data streams and the propagation of change (Keval Patel, no date). Angular uses RxJS to implement the *Observable* pattern. An *Observable* is a stream of asynchronous events that can be processed with array-like operators (Lasorsa, 2018).
In the Forecast Monitor UI, Observables are used to update the view when the component receives information from the backend. Because the core function of the system is to be used as a dashboard, pattern like this allows a dynamically changing content without risks of memory leaks. This is elaborated more in the Implementation section.

4.3.1.2 Unidirectional data flow

Angular uses unidirectional data flow, meaning change detection cannot cause cycles. Using unidirectional data flow helps to maintain simpler and more predictable data flows in applications, along with substantial performance improvements (Lasorsa, 2018).
In the Forecast Monitor UI, the data flow is from the viewModel to the view and all interactions with the view are dispatching actions and not triggering cycles. This pattern is used because as a dashboard system most of the actions are automatic (page load, data received from the backend service), so they originate from the business logic and not from the GUI, so in those cases the application doesn't need to run cycles for detecting changes on the view. Implementing a two-way data binding for example will cause unnecessary cycles when rendering the view and respectively affect the performance.



4.3.1.3 Model – View separation

Angular have a component-based architecture but it's following some of the Model View ViewModel principles, however it doesn't implement the pattern completely. In angular, Typescript classes or interfaces are used to define the model. However, models are just plain-old JavaScript objects. The view is defined by the template which consist of HTML, CSS and some Angular specific elements and attributes and the "viewModel" is represented by the Typescript part of the component where all the logic for this component is located and all changes are reflected on the view (Lukas Marx, 2016).

In the Forecast Monitor UI, interfaces are used to define the models' structure and the rest of the application is based on components whit each one having its own view and viewModel. For retrieving data for the models however we use the Reactive Programing paradigm by using Services which fetch data from the backend and provide it to the viewModel through an observable data stream. This pattern is used mainly because that is how the Angular framework is designed but also having a component-based architecture allows us to have small reusable components that are easy to test and control.

4.3.2 User Interface

The user interface is designed by following the drafts from the UX designer and the CSL theme. Additional adjustments (if necessary) are made after a consultation with the UX designer.

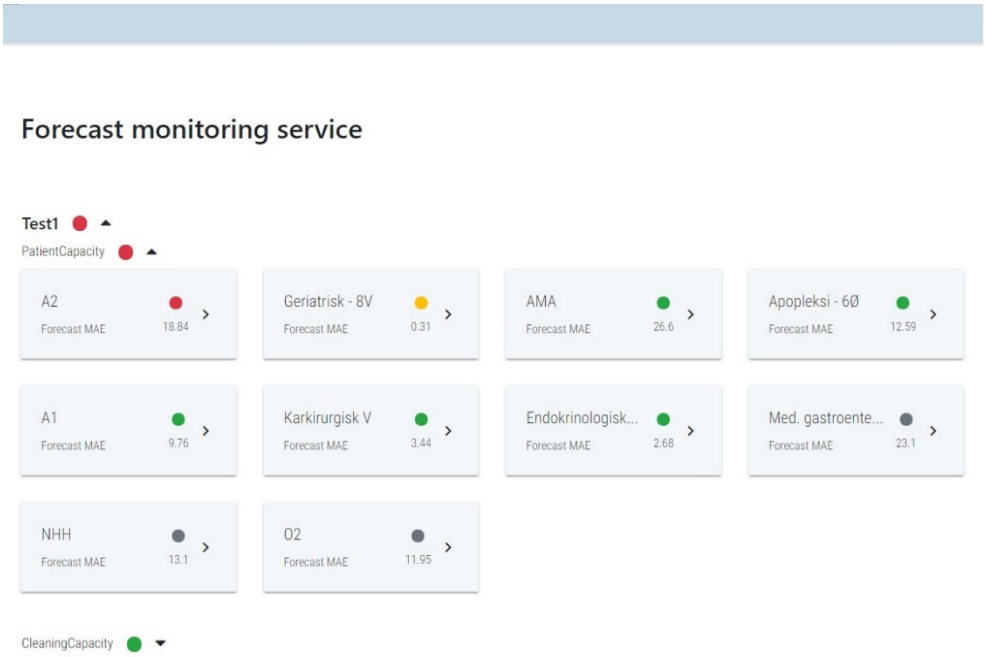


Figure 28 Dashboard page

On the figure above is displayed the dashboard page of the application, where all installations (regions) are listed with their clients and all units for each client. The design is following a nested tree structure in order to provide a better and quicker overview of the system. Each unit has a performance indicator which is propagated to the client and from there to the installation where the worst performance is considered. With this structure whenever there is a unit with a red performance indicator it's client and installation are automatically unfolded in this way the dashboard will always show the underperforming units, so they can be examined and potentially fixed. The unfolding happens whenever a new data is retrieved, on page load and the backend sends data every 10 min through the socket connection.

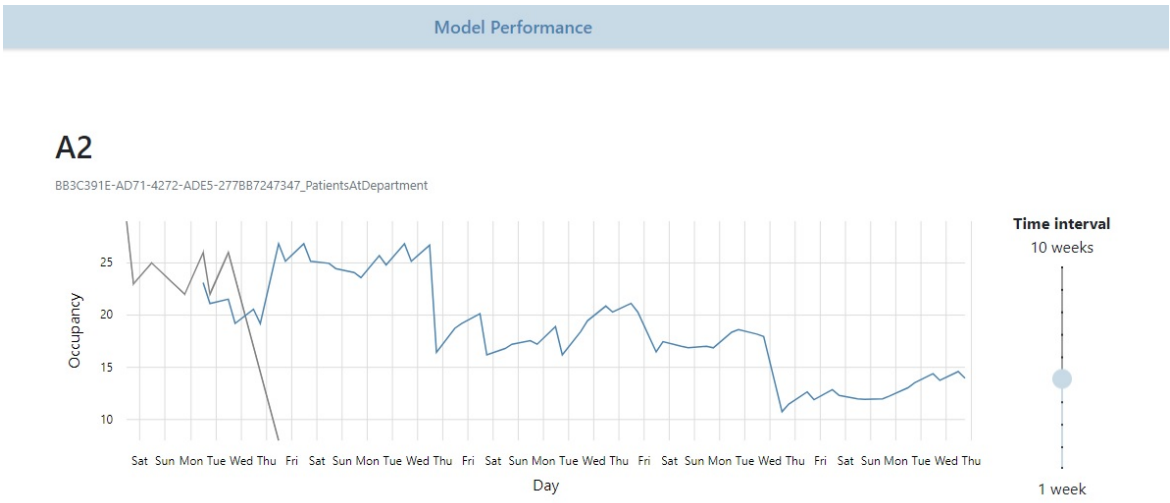


Figure 29 Model Performance page

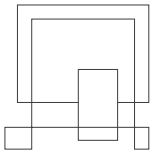
The Model Performance page is designed to show a performance graph for the specific unit with both the historical (real) and predicted data for a 2 weeks interval back by default. The interval can be configured with the slider on the right side. On the graph the “blue” line is showing the prediction data and the “grey” one is the historical. The system is in development mode and that is why the data looks inconsistent and there is not much of historical data.

5 Implementation

Implementation contains information on how the designed solution was implemented, which frameworks and libraries were used and a description of the UI.

5.1 ForecastMonitor Service

The backend was designed for and built by ASP.NET Core 2.1 framework (Daniel Roth, Rick Anderson, 2018). Even though ASP.NET Core is a relatively new addition to the .NET stack and none of Systematics' systems was using it at the time of writing, It was for some of its unique out-of-the-box new features, such as ability to create RESTful services (Web API) in a very easy and agile way, built-in dependency



injection, logging, background services and a thread-safe Http client provider. Moreover, it's so close to the rest of .NET family that there is almost no learning curve and a very short transition phase.

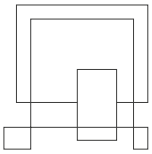
5.1.1 Startup class

.Net Core uses a Startup class, which is a class for registering services using its DI container and to configure app's request pipeline & middleware.

```
public class Startup
{
    3 references | Tomas Izo, 74 days ago | 1 author, 3 changes | 0 exceptions
    private IConfiguration Configuration { get; }

    0 references | Tomas Izo, 71 days ago | 1 author, 3 changes | 0 exceptions
    public Startup(IHostingEnvironment env)...

    // This method gets called by the runtime. Use this method to add services to the container.
    0 references | Tomas Izo, 21 days ago | 1 author, 32 changes | 0 exceptions
    public void ConfigureServices(IServiceCollection services)...

    // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
    0 references | Tomas Izo, 42 days ago | 1 author, 1 change | 0 exceptions
    public void Configure(IApplicationBuilder app, IHostingEnvironment env, IMapper mapper)...
}
```

Figure 30 .Net Core Startup class overview

Typically, third party middleware or pipeline setup is done via extension methods – which helps to isolate configuration logic of a given package and keeps the class readable.

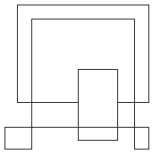


```
        // MediatR
        services.AddMediatR();

        // SignalR
        services.AddSignalR();

    }

    // This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
    0 references | Tomas Izo, 44 days ago | 1 author, 1 change
    public void Configure(IApplicationBuilder app, IHostingEnvironment env, IMapper mapper)
    {
        mapper.ConfigurationProvider.AssertConfigurationIsValid();

        app.UseSwaggerUi(typeof(Startup).GetTypeInfo().Assembly, cfg => cfg.DefaultProfile());

        app.UseRewriter(RewriterConfiguration.RewriteOptions);

        app.UseCors(builder =>
        {
            builder.AllowAnyHeader()
                .AllowAnyMethod()
                .AllowCredentials()
                .AllowAnyOrigin();
        });

        app.UseWebSockets();

        app.UseSignalR(routes => routes.MapHub<UnitHub>("/hub/units"));
```

Figure 31 Service Startup class middleware setup

5.1.2 Background services & consuming scoped services

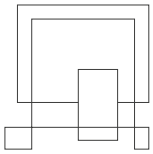
The design mentioned earlier that .Net Core comes with background services and they are being used for executing jobs. Creation of background service can be done via implementing IHostedService or by inheriting from a base class BackgroundService. AdHocJobService uses the latter method. Its ExecuteAsync method starts as soon as the backend launches and tries to execute queued jobs. An empty JobQueue is blocked by a semaphore and makes the service wait instead of pooling at fixed interval, until it has a job.



```
7 references | Tomas Izo, 47 days ago | 1 author, 1 change
public class AdHocJobService : BackgroundService, IAdHocJobService
{
    private readonly ILogger _logger;
    private readonly IJobQueue _jobQueue;

    0 references | Tomas Izo, 47 days ago | 1 author, 1 change
    public AdHocJobService(IJobQueue jobQueue, ILoggerFactory loggerFactory)...

    3 references | Tomas Izo, 47 days ago | 1 author, 1 change
    public Task AddJob(IAdHocJob job)...

    0 references | Tomas Izo, 47 days ago | 1 author, 1 change
    protected override async Task ExecuteAsync(CancellationToken cancellationToken)
    {
        this._logger.LogInformation($"{nameof(AdHocJobService)} is starting.");
        while (!cancellationToken.IsCancellationRequested)
        {
            var job = await this._jobQueue.DequeAsync(cancellationToken);
            try
            {
                this._logger.LogInformation($"{nameof(AdHocJobService)} is executing {job.GetType()}.");
                await job.ExecuteAsync(cancellationToken);
            }
            catch (Exception e)
            {
                this._logger.LogError(e, $"Error occured during execution of job: {job.GetType()}.");
            }
        }
        this._logger.LogInformation($"{nameof(AdHocJobService)} is stopping.");
    }
}
```

Figure 32 AdHocJobService implementation

Jobs are registered by DI as singletons and need to create its own scope to be able to access scoped or transient services. This is done by passing IServiceScopeFactory into its constructor. The figure below shows how PerformanceCalculationJob calculates the unit performance using scoped services.

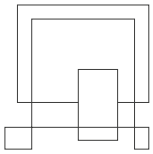


```
1 reference | Tomas Izo, 29 days ago | 1 author, 4 changes
private void DoPerformanceCalculation()
{
    using (var scope = Factory.CreateScope())
    {
        var performanceLogic = scope.ServiceProvider.GetRequiredService<IPerformanceCalculationLogic>();
        var unitDataService = scope.ServiceProvider.GetRequiredService<IUnitDataService>();
        var modelDataService = scope.ServiceProvider.GetRequiredService<IModelDataService>();

        var units = unitDataService.GetAllUnits();
        foreach (var unit in units)
        {
            var model = modelDataService.GetLatestEvaluableModel(unit);
            if (model != null)
            {
                var mae = performanceLogic.MAE(model);
                model.Mae = mae;
                unit.Mae = mae;

                var zScore = performanceLogic.ZScore(unit);
                unit.ZScore = zScore;

                this._logger.LogDebug($"{unit.Key} | MAE: {unit.Mae} | Z-score: {unit.ZScore}");
            }
            else
            {
                this._logger.LogInformation($"{unit.Key}: not evaluable");
            }
        }
    }
}
```

Figure 33 Consuming scoped services

5.1.3 Job chaining using MediatR

AdHocJobService runs any job that is in its queue and MediatR makes use of it to define a job sequence (Bogard, 2018). Every job inherits from the Job class and implements its logic in DoExecuteAsync method. To execute the logic, job services call ExecuteAsync – the wrapper method, which, at the end, notifies NotificationHandler (another MediatR component) about the job completion. Once the notification is received, the handler adds another job in the queue, given that a job successor is pre-defined.



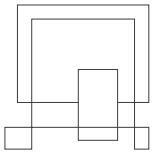
```
3 references | Tomas Izo, 47 days ago | 1 author, 1 change
public abstract class Job : IJob
{
    protected readonly IServiceScopeFactory Factory;

    2 references | Tomas Izo, 47 days ago | 1 author, 1 change
    protected Job(IServiceScopeFactory scopeFactory)
    {
        Factory = scopeFactory;
    }

    4 references | Tomas Izo, 47 days ago | 1 author, 1 change
    protected abstract Task DoExecuteAsync(CancellationToken cancellationToken);

    7 references | Tomas Izo, 47 days ago | 1 author, 1 change
    public async Task ExecuteAsync(CancellationToken cancellationToken)
    {
        await DoExecuteAsync(cancellationToken);
        using (var scope = this.Factory.CreateScope())
        {
            var mediator = scope.ServiceProvider.GetRequiredService<IMediator>();
            await mediator.Publish(new JobCompletedNotification(GetType()), cancellationToken);
        }
    }
}
```

Figure 34 Job class implementation

5.1.4 Units publishing with SignalR over Web Sockets

The design has already mentioned advantages of SignalR – a library for developing real-time web applications, typically using Web Sockets (*Introduction to ASP.NET Core SignalR | Microsoft Docs*, no date). It has has a simple implementation and plays a role in units publishing. There are just three steps:

- Create a hub – (UnitHub) can be anywhere in the assembly and does not implement any logic, only inherits from Hub class
- Register SignalR in startup class – a figure with example of Startup class (shown earlier) has already shown Startup.Configure() method with this step by calling app.UseSignalR() extension method
- Inject HubLifetimeManager to send messages – UnitPublishingService sends new units by calling hubManager.SendAllAsync() on a topic “UnitsUpdate” (any client who listens to the topic will receive the update)



```
3 references | Tomas Izo, 30 days ago | 1 author, 4 changes
public class UnitPublishingService : IUnitPublishingService
{
    private readonly IUnitDataService _dataService;
    private readonly IMapper _mapper;
    private readonly ILogger _logger;
    private readonly HubLifetimeManager<UnitHub> _hubManager;

    0 references | Tomas Izo, 30 days ago | 1 author, 1 change
    public UnitPublishingService(IUnitDataService dataService, IMapper mapper,
        HubLifetimeManager<UnitHub> hubManager, ILoggerFactory loggerFactory)
    {
        this._dataService = dataService;
        this._mapper = mapper;
        this._logger = loggerFactory.CreateLogger<UnitPublishingService>();
        this._hubManager = hubManager;
    }
    3 references | Tomas Izo, 30 days ago | 1 author, 3 changes
    public async Task PublishUnits(CancellationToken cancellationToken)
    {
        var daoUnits = this._dataService.GetAllUnits();
        var dtoUnits = daoUnits.Select(this._mapper.Map<DtoUnit>).ToList();
        this._logger.LogDebug($"Publishing {dtoUnits.Count} units");
        await _hubManager.SendAllAsync("UnitsUpdate", new object[] { dtoUnits }, cancellationToken);
    }
}
```

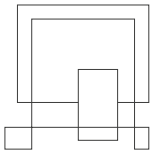
Figure 35 Publishing units with SignalR

5.1.5 Mapping

Mapping dao objects to dto objects is done with AutoMapper (*AutoMapper*, 2018). It allows to define mapping profiles with custom logic and has ability to resolve semantically similar properties automatically. IMapper can be registered by DI container via extension method and used in any service that needs to perform mapping by calling generic Map method. Figure below show the mapping profile class. UnitPerfomanceValueResolver contains a logic of mapping z-score to a color label.

```
3 references | Tomas Izo, 29 days ago | 1 author, 3 changes
public class DaoToDtoMappingProfile : Profile
{
    1 reference | Tomas Izo, 29 days ago | 1 author, 3 changes
    public DaoToDtoMappingProfile()
    {
        CreateMap<DaoInstallation, DtoInstallation>()
            .ForMember(dto => dto.LastUpdate, cfg => cfg.MapFrom(dao => dao.TimeStamp));
        CreateMap<DaoClient, DtoClient>()
            .ForMember(dto => dto.LastUpdate, cfg => cfg.MapFrom(dao => dao.TimeStamp));
        CreateMap<DaoUnit, DtoUnit>()
            .ForMember(dto => dto.Performance, cfg => cfg.ResolveUsing<UnitPerformanceValueResolver>())
            .ForMember(dto => dto.LastUpdate, cfg => cfg.MapFrom(dao => dao.TimeStamp));
    }
}
```

Figure 36 AutoMapper mapping profile



5.2 ForecastMonitor UI

The framework chosen for the front end is Angular 6, a stable version of the Angular framework with some good improvements from its previous version.
For example, in Angular 6 the http library was replaced with the new HttpClient API which is faster, more secure and allows better implement the reactive programming pattern.

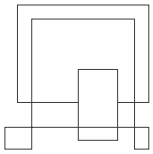
```
57      const initialSource = this.dataService.getUnits(
58        this.installation_id,
59        this.id
60      );
61
62      const updateSource = this.dataService.onUnitsUpdate.pipe(
63        map(units => units.filter(u => u.client_id === this.id)),
64        tap(units =>
65          console.log('Units Update: ' + units.length + ' units received')
66        )
67      );
68
69      this.activeModels$ = concat(initialSource, updateSource).pipe(
70        map(this.orderUnits),
71        tap(this.updateClientStatus)
72      );
```

Figure 37 Retrieve unit's data – viewModel

In the above example is shown data is being pulled from the backend service. Combining two streams of data, initialSource is used to get the data when the page loads and updateSource for updating the data afterwards for example when new data comes from the socket connection and piping the output through common functions like the mapping function in the above example.

```
83    getUnits = (installation_id: number, client_id: number) => {
84      // Create a subinstance of the handler with the operation that failed and the return object
85      const handler = this.handler('Get Units', []);
86      return this.http
87        .get<IUnit[]>(
88          this.API +
89            `/units?installationId=${installation_id}&clientId=${client_id}`
90        )
91        .pipe(
92          tap(() => this.dialogRef.closeAll()),
93          retryWhen(handler.retryStrategy),
94          catchError(handler.handleError)
95        );
96    }
```

Figure 38 Retrieve unit's data – service



The above figure is a good example of the benefits from using reactive programming and Observables. Error handling and retry strategies can be implemented on the service level and ensure the consumer of the service that this method will always return a valid data. On line 85 an instance of the handler is created with the name of the operation so if an error occurs it can be display/logged correctly and also the object to be returned in case of an error in order to ensure that the application can continue working. With this approach the service can be consumed on many places without worrying about implementing error handling and retry strategy on every call.

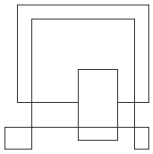
```
<section class="client">
    <header class="primary-border-hover" [ngClass]="{'collapsed': status !== UnitStatus.Red}" fxLayoutGap="1rem" data
        <p class="mb-0 client-name">{{name}}</p>
        <app-performance-indicator [status]="status"></app-performance-indicator>
    </header>
    <div class="collapse" [ngClass]="{'show': status == UnitStatus.Red}" id="client-content-{{id}}">
        <div class="content">
            <app-model *ngFor="let model of activeModels$ | async" [model]="model" [label]="unitLabel"></app-model>
        </div>
    </div>
</section>
```

Figure 39 Part of the client component

```
<mat-card class="accent-background" routerLink="model/{{installation_id}}/{{id}}">
    <mat-card-content *ngIf="!loading; else progress">
        <div class="card-column">
            <div class="mb-0 model-name">
                <span [matTooltip]="name || 'Unit not found: ' + id">{{name || 'Unit not found: ' + id | truncate:15:'...'}}</span>
            </div>
            <span class="mae-lable" *ngIf="label; else progress">{{label}}</span>
        </div>
        <div class="card-column nested">
            <div class="card-column center">
                <div class="mae-indicator">
                    <app-performance-indicator [status]="performance"></app-performance-indicator>
                </div>
                <span class="mae-value">{{mae}}</span>
            </div>
            <div class="card-column">
                <mat-icon>chevron_right</mat-icon>
            </div>
        </div>
    </mat-card-content>
</mat-card>
```

Figure 40 Part of the unit component

Using the Angular framework, allows breaking the application into small components and combining them together, as show in the two figures above the client component is fairly small and contains multiple unit components which are created with a loop, and it is the same with the higher-level components like installations and the dashboard which is the main component. This way a template can be defined once and then repeated whenever it's needed but it also introduces a problem in the communication between the components. Because parent and child component doesn't share their

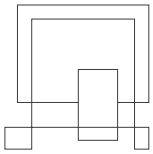


scope the only way of communication is through events that the child component can emit, and the parent can listen for.

```
updateClientStatus = (units: IUnit[]): void => {
  // If the client found have no status set yet consider it as Not enough data
  if (!this.status) {
    this.status = UnitStatus.Grey;
  }
  // Reduce function for getting the worst performing unit
  const getWorstPerformance = (
    _worst: UnitStatus,
    current: UnitStatus
  ): UnitStatus => (current < _worst ? current : _worst);

  // Update the client status
  this.status = units
    .map(u => UnitStatus[u.performance])
    .reduce(getWorstPerformance, UnitStatus.Grey);

  // Report the status to the installation
  this.reportStatus.next({
    id: this.id,
    status: this.status
  });

  // Force change detection to update the view
  this.cd.detectChanges();
}
```

Figure 41 Update clients' status function

One such situation is when the application receives the units for a specific client. The client status needs to be updated to match the worst performing unit and then inform the installation about the client's status. Since the installation doesn't have access to the scope of the client after it was instantiated the only way of informing the installation about the change in the client's status is by emitting an event. In the figure above this event is called "reportStatus" and it emits an object containing the client id and its status.



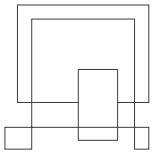
```
updateInstallationStatus = (report: IStatusReport): void => {
  // Reduce function for getting the worst performing unit
  const getWorstPerformance = (
    _worst: UnitStatus,
    current: UnitStatus
  ): UnitStatus => (current < _worst ? current : _worst);

  // Find and update the local client
  const target = this.clients.find(c => c.id === report.id);
  target.status = report.status;

  const isLast = this.clients.indexOf(target) === this.clients.length - 1;

  if (isLast) {
    // Update the installation status
    this.status = this.clients.map(c => c.status).reduce(getWorstPerformance);
    // Order the clients by status
    const ordered = _.orderBy(this.clients, 'status', 'asc');
    this.updater.next(ordered);

    // Report the installation status
    this.reportStatus.next({
      id: this.id,
      status: this.status
    });
  }
}
```

Figure 42 Update installations status

Because an installation can have multiple clients, each time a client reports a status it needs to be stored and the updated must happen only when all clients have reported their status so then the worst performing one can be found and its status assigned as the status of the installation. This is one of the downsides for using reactive programming and component-based architecture, but it's worth compared to the performance and code readability we gain.

5.3 Test Projects

The section after this one covers test strategy and test varieties, revealing that four kinds of automated tests were used for the solution. This section will briefly touch on some of the libraries and frameworks used for the testing.



5.4 Testing framework - NUnit

There were another two testing frameworks in consideration (MSTest & XUnit), but NUnit (*NUnit.org*, 2018) won because it has the richest feature set from all, it's used in another Systematics projects and their developers have a positive experience with it.

5.4.1 FluentAssertions

The assertion library used in all test cases (*Fluent Assertions - Fluent Assertions*, no date). As opposed to NUnit assertions, it makes them more natural, and most importantly, more readable. The library achieves it thanks to its fluent API, built by extension methods.

```
// Assert
units.Should().NotBeNullOrEmpty();
installationIds.Should().AllBeEquivalentTo(installation.Id);
```

Figure 43 FluentAssertions example

5.4.2 Moq & AutoFixture

Moq ('Moq4', 2018) is the most popular mocking framework, while AutoFixture (Mark Seemann, no date) is a library designed to minimize the "Arrange" part of the test case. They are mentioned together, because if used together, AutoFixture can be used as auto-mocking container. Figure with example of UnitTest (section 6.2.1) shows and example, where Fixture.Create() is called to generate and object with random properties. Later, Fixture.Inject() is called to inject a specific mock of an interface. Since then, every service depending on that interface, created by AutoFixture, will be created using that very mock.

```
[SetUp]
0 references | Tomas Izo, 49 days ago | 1 author, 1 change
public void Setup()
{
    Fixture = new Fixture().Customize(new AutoMoqCustomization());
}
```

Figure 44 AutoFixture as auto-mocking container

5.4.3 WireMock.Net & Handlebars.Net

WireMock.Net ('WireMock.Net', 2018) is a library which allows stubbing and mocking Http responses and is highly customizable. Data in ForecastMonitor depends on external service and WireMock allows to mock it – it even allows to create unresponsive, or stateful endpoint, which becomes responsive after a certain amount of calls. It's used also for the UI tests, as a stub of the backed with fixed data.

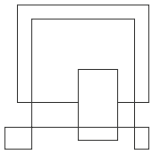
WireMock can do request filtering and reply with a json object read from a file, but it can't filter a response object based on request parameters - That's where Hanlebars.Net comes in ('Handlebars.Net', 2018). The figure below shows an example of implementation of the backends client endpoint stubbing.

```
1 reference | Tomas Izo, 23 days ago | 1 author, 1 change
public static FluentMockServer ConfigureClients(this FluentMockServer server,
    string clientJsonPath = DefaultClientsJson)
{
    var json = TestHelper.ReadJson($"{PathForForecastMonitorTestData}{clientJsonPath}", ProjectFolder);

    Handlebars.RegisterHelper("filter-clients", (output, context, arguments) =>
    {
        var obj = FilterJson(json, arguments);
        output.Write(obj);
    });

    server.Given(Request.Create()
            .WithPath("/clients")
            .WithParam("installationId")
            .UsingGet())
        .RespondWith(Response.Create()
            .WithStatusCode(HttpStatusCode.OK)
            .WithHeader("Content-type", "application/json")
            .WithHeader("Access-Control-Allow-Origin", "*")
            .WithHeader("Access-Control-Allow-Methods", "*")
            .WithBody("{{{filter-clients request.query}}}")
            .WithTransformer());

    return server;
}
```

Figure 45 Http responses stubbing

5.4.4 Selenium WebDriver

Selenium WebDriver is a library with Selenium API – a browser automation tool, used in UI tests (*Selenium - Web Browser Automation*, 2018). Using Selenium rich API together with WireMock.Net allows to minimalize manual tests.

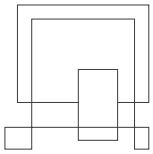


5.4.5 Microsoft.AspNetCore.MVC.Testing

Last, but not least are Microsoft's own .Net Core testing package. It allows to run Integration tests, by running the backend with custom configuration, such as disabling, adding, removing, or mocking certain services via TestServer class. BaseIntergationTest class takes IServiceCollection action in a constructor an applies it to the TestServer before TestFixture runs. This way can any given integration test run under its own configuration. Below is an example of CacheUpdateJobTest (integration test) setup.

```
0 references | Tomas Izo, 50 days ago | 1 author, 1 change
public CacheUpdateJobTests() : base(services => {
    services.RemoveJobScheduler()
        .RemoveForecastSystemServices()
        .AddForecastSystemServiceWorkingMock()
        .AddCacheResetService()
        .AddSingleton(PublishingMock.Object);
})
{
}
```

Figure 46 TestServer services setup example

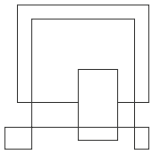


6 Test

This section documents the way testing are conducted in this project. There are many different definitions of terms in the discipline of software testing, in this project, the terms and definitions from TMap are used (*Testing with the TMap Suite*, 2018).

6.1 Test strategy

The main goal for testing in this project is to validate the fulfilment of the functional requirements defined in section 2.1, as well as to ensure the business logic defined along the way of feature development by the stakeholders (specified in the feature FDDs).
Furthermore, the strategy of early testing, is followed in the project in such way, that tests are conducted in as early stage of development and as low implementation level as possible, e.g. if it is possible to achieve the same coverage, then unit tests is preferred over integration test. For further elaboration about how the strategy of early testing has been carried out, please see the Project Execution section of the Process Report.
Figure 47 shows the Test Design Table for feature FDD0001 Display Unit Information. In the table, each test corresponds to one or part of one of the requirements to be fulfilled by the feature implementation. There will always be minimum one test (for a sunshine scenario) per feature requirement in the test design table. Each test specified in the test design table has a globally unique identifier, consisting of the requirement id and test id, so the implementation and result of it can be tracked both in the source code and the manual test specification document. The test type of each test is defined following the strategy of early testing stated above.
The test design table also specifies, at which part of the development the test should be implemented and conducted, it could either be with the implementation of a story, or at the final feature testing stage.



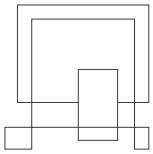
No.	Test condition	Test type (UI, service, integration, manual)	Test conducts with Story no. / Feature	Comments	Result
OS5-1	For each monitored system, there should be an overview of all the monitored forecast pipelines	Manual	Feature	TSP	✅
PP1-1	The solution system should be able to display the Prediction Performance as MAE score.	Manual	Feature	TSP	✅
PP1-2	The MAE score is calculated from comparing the historical forecast to the historical result in intervals that is configurable, but two weeks as the default backwards.	Unit	2	Backend, set to 28 days due to lack of data on test server	✅
PP4-1	The solution system should be able to display performance information in a sorted manner, so the presentation of Region, Client and Unit are ordered by first the severity of the performance indication, then MAE, both in descending order.	GUI	4	Front end, the list of units is ordered by MAE using a third party library	✅
PP5-1	The system indicates current prediction performance in color code, according to: • i. Red - indicates MAE≥3std ii. Yellow - indicates 1.5std≤MAE<3std iii. Green - indicates MAE<1.5std	Unit	2	Test for colorcode "calculation" logic at backend.	✅
PP5-2	The system displays client performance indication as its worst performing unit.	GUI	4	Front end	✅
PP5-3	The system displays region performance indication as its worst performing client.	GUI	4	Front end	✅
PP6-1	The solution system will automatically unfold any client with red performance indication.	GUI	4	Front end	✅

Figure 47 Test Design Table for feature FDD0001

Besides the special case for manual tests, for which the test specifications reside in the Test Specification Document (Appendix E), the rest of the test varieties has their test specification in the Feature Description Documents (FDDs).

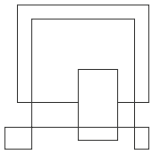
6.2 Test varieties

This section seeks to explain the test varieties that are distinguished in this project, and in which way the test activities are distributed among these. The test varieties are presented in the order they are preferred in this project.



6.2.1 Unit test

Unit tests for backend implementation is in a separate project than the system source code, while for the frontend they are in the same place as the system source code. They have responsibility to test the smallest unit of the system, for the backend it is a class, while for frontend it is an angular component. This is the most common test variety in the project and can be used for business logic assurance, as a conduct of early testing strategy, otherwise they are most used for assisting development.
Below is an example of a unit test that is testing the fulfillment of the requirement PP1, part two - The MAE score is calculated from comparing the historical forecast to the historical result in intervals that is configurable, but two weeks as the default backwards.

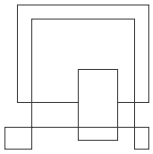


```
[Test, Description("PP1-2")]
0 references | Tomas Izo, 19 days ago | 1 author, 2 changes
public void GetMappablePredictionsAndTimeSeries_Uses_Configurable_HistoricalDataLookupInDays()
{
    // Arrange
    var actualLookupInDays = 5;
    var rand = new Random();
    var model = Fixture.Create<DtoModelInfo>();
    var unit = Fixture.Create<DtoUnitKey>();
    var dates = Fixture.CreateMany<DateTime>(50)
        .OrderBy(_ => _.Date)
        .ToList();

    var predictions = CreateRandomPredictionsFromDates(dates, rand);
    var timeSeries = CreateRandomTimeSeriesFromDates(dates, rand);


    var firstAllowedPredictionDate = predictions.First().PredictionForDate.AddDays(-actualLookupInDays);

    Fixture.Inject(Mock.Of<IForecastSystemClient>(client =>
        client.GetPredictions(It.IsAny<Installation>(), It.IsAny<DateTime?>(),
            It.IsAny<DateTime?>(), It.IsAny<string>(), It.IsAny<int?>()) == Task.FromResult(predictions) &&
        client.GetTimeSeries(It.IsAny<Installation>(), It.IsAny<string>(), It.IsAny<string>(),
            It.IsAny<DateTime?>(), It.IsAny<DateTime?>(), It.IsAny<int?>()) == Task.FromResult(timeSeries))
    );
    Fixture.Inject(Mock.Of<IAppSettingsManager>(manager => manager.HistoricalDataLookupInDays == actualLookupInDays));

    _sut = Fixture.Create<ForecastSystemService>();

    // Act
    var result = _sut.GetMappablePredictionsAndTimeSeries(unit, model).Result;

    var actualPredictionDates = result.Item1.Select(x => x.PredictionForDate).ToList();
    var actualTimeSeriesDates = result.Item2.Select(x => x.Date).ToList();

    // Assert
    actualPredictionDates.Should().OnlyContain(_ => _ >= firstAllowedPredictionDate);
    actualTimeSeriesDates.Should().OnlyContain(_ => _ >= firstAllowedPredictionDate);
}
```

Figure 48 Unit Test

6.2.2 Integration test

Integration tests have the responsibility to test the integration of several components in the system. They are located in a separate project than the system source code. Integration tests are mainly designed and implemented as business logic assurance, as most business logics requires cooperation of many components in order to be carried out but can also be used for development assisting purpose. The figure below shows a test of Http client retry policy. ForecastMonitor Service Http client policy dictates the client to retry calls on fail 6 and the test below validates its functionality.



```
[Test]
0 references | Tomas Izo, 41 days ago | 1 author, 1 change
public async Task Retry_Policy_Ensures_Successful_Request()
{
    // Arrange
    var rand = new Random();
    var triesAfterServerRespondsWithinPolicyLimit = rand.Next(1, 5);
    _forecastSystemStub.ConfigureClientUnresponsiveWithFixedData(triesAfterServerRespondsWithinPolicyLimit);
    var manager = TestServer.Host.Services.GetRequiredService<IAppSettingsManager>();
    var installation = manager.Installations.ToList().First();

    // Act
    var clients = await _sut.GetClients(installation);

    // Assert
    clients.Should().NotBeNullOrEmpty();
}
```

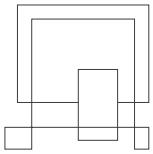
Figure 49 Integration Test

6.2.3 End-to-end test

End-to-end tests have the responsibility to test the backend functionality as a service provider through its API. They are located in a separate project than the system code. They are testing the availability of the service endpoint and roughly the functionality of it by passing the expected parameters and roughly check the structure of the returned value.
Below is an example of End-to-end test, which assures the installations service GET endpoint works as desired.

```
[Test]
0 references | Tomas Izo, 52 days ago | 1 author, 1 change
public async Task Get_Installations()
{
    // Arrange
    var url = "/installations";

    // Act
    var httpResponseMessage = await _client.GetAsync(url);
    var actualResponse = await httpResponseMessage.Content.ReadAsStringAsync();
    Action successStatusCodeVerification = () => { httpResponseMessage.EnsureSuccessStatusCode(); };

    // Assert
    successStatusCodeVerification.Should().NotThrow();
    actualResponse.Should().NotBeNullOrEmpty();
}
```

Figure 50 E2E Test



6.2.4 UI test

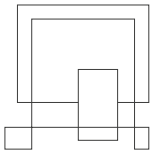
UI tests have the responsibility to test the business logic for user interaction and content presentation. They reside in their own project, and they are implemented with Selenium WebDriver – browser automation library. They are designed for business logic assurance.
Below is an example of a UI test that is testing the fulfillment of the requirement PP6 - The solution system will automatically unfold any client with red performance indication.

```
[Test, Description("PP6-1")]
[TestCase("green", false)]
[TestCase("grey", false)]
[TestCase("yellow", false)]
[TestCase("red", true)]
0 references | Tomas Izo, 16 days ago | 1 author, 1 change
public void Poor_Performing_Clients_Unfold_Automatically(string unitJson, bool shouldBeUnfolded)
{
    // Arrange
    ForecastMonitorServiceStub.AsDashboard(pathToUnits: $"units/{unitJson}.json");

    // Act
    Driver.Navigate().GoToUrl(UIBaseUrl);

    var element = Driver.FindElementByClassName("installation")
        .FindElement(By.Id("installation-content-1"))
        .FindElement(By.Id("client-content-1"));

    element = element.FindElement(By.ClassName("card-column"))
            .FindElement(By.ClassName("model-name"));

    var isUnfolded = element.Displayed;

    // Assert
    isUnfolded.Should().Be(shouldBeUnfolded);
}
```

Figure 51 UI Test

6.2.5 Manual test

Due to the cost of regression testing, the use of manual tests is limited to as little as possible. Hence this test variety is only used, when a business logic cannot be tested with any of the above test varieties. Specification of manual test are documented in the Test Specification Document in Appendix E.

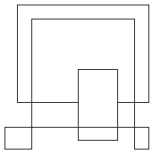


7 Results and Discussion

The result of the project is a working and stable application that is ready to be used even in production. Even tough not fully finished, because of using feature driven development, the features were prioritized and the most important if them were implemented so that at the delivered product is working, contains the most important functionality and is tested enough to show the fulfillment of its requirements. The product is also developed in a way to be scalable, maintainable and extendable which also was one of the requirements.

The main question that was asked in the beginning of the project “How can the performance of the forecast system be visualized?” is answered and the short version is by using a tree structured hierarchy to visualize the hierarchy of the forecast system, color code representation of the overall performance of each unit of the system and in-depth line chart visualizing the historical and prediction data for that model for a specific interval back in time.

The product fulfills its main use case and that is to be used as a dashboard for monitoring the current performance of all algorithms. Using techniques like socket connection for live updating the data and dynamically unfolding installations and clients whenever a unit starts failing allows the product to be used with close to no interaction but observation. Acting is only required when a failing unit needs to be investigated or a new model needs to be trained. Although the most important use cases are implemented there are some that was let behind as they were with lower priority and some were blocked by the forecast system itself.



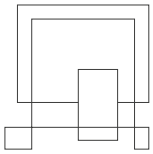
Feature #	Feature Name	Requirements				Status
		OS	PP	FS	MI	
FDD0000	Display Forecast Systems	OS1, OS2, OS3, OS4				Done
FDD0001	Display Unit Information	OS5	PP1, PP4, PP5, PP6			Done
FDD0002	Display Unit Prediction Performance Plot		PP2			Done
FDD0004	Trigger model retrain for certain unit	OS7				Done
FDD0003	Display Model Information				MI1	Blocked By FS
FDD0005	Display Feature Importances			FS1		Blocked By FS
FDD0006	Display Client Prediction Performance		PP7			Not Started
FDD0007	Display Training Data Metrics		PP3			Blocked By FS
FDD0008	Indicate current model underperformance	OS6				Not Started
FDD0009	Display feature value against freuency in Histogram			FS2		Blocked By FS

Figure 52 Feature statuses

As shown in the figure above the solution is unable to display some of the information about a specific model or the feature importance's because the forecast system itself was not able to provide that kind of information. This was discussed in the planning phase of the project and because the forecast system is under development as well those features got a lower priority.

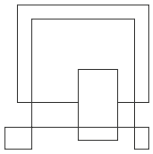
The only pitfall the solution have is that it is strongly dependent of having a fairly big data set to work with and with the current forecast system working in a test environment sometimes there is not enough data to perform some of the calculations and determine the performance.

In conclusion, the goal of the project was reached. The solution is stable, tested and ready for use it provides the most important information needed for monitoring the forecast system. It is also maintainable and extendable so any of the features left behind can be implemented at a later stage from the DABAI team.



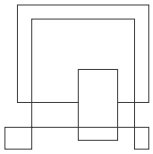
8 Conclusions

Thanks to a solid introduction, the project defined the problem and needs by a set of well-structured requirements. Thorough analysis of the problem proposed a solution that will fit well into Systematic's ecosystem – an Angular web application with a .Net RESTful service. It also makes use of FDD process to link all requirements with features, which helped to design a solution that fulfills all high priority requirements and remains open for the rest. Implementation provides a deeper technical insight on the solution's structure, including list of used packages and frameworks. Test section shown the test strategy and multiple test varieties, which validated implemented solution's proper functionality and fulfilment of its requirements.



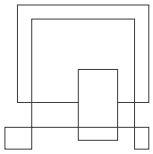
9 Project future

At the current stage, the solution is already used internally, and nothing prevents it from use in the production - with the most important features implemented, fully working and tested. From a technical standpoint of view, no changes are required, but to fully satisfy all the requirements, the solution needs to implement the remainder of the planned features. Both the backend (ForecastMonitor Service) and the frontend (ForecastMonitor UI) are developed with scalability and maintainability in mind, meaning the backend is implemented as a stateless service following the SOLID principles and the frontend is implemented as a component based single page application using reactive programming for handling data.

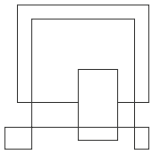


10 Sources of information

Angular (no date). Available at: https://angular.io/ (Accessed: 7 December 2018).
AutoMapper (2018). Available at: https://automapper.org/ (Accessed: 7 December 2018).
Background tasks with hosted services in ASP.NET Core | Microsoft Docs (no date). Available at: https://docs.microsoft.com/en-us/aspnet/core/fundamentals/host/hosted-services?view=aspnetcore-2.2 (Accessed: 7 December 2018).
Balliauw, M. (no date) *Building a scheduled task in ASP.NET Core/Standard 2.0*. Available at: https://blog.maartenballiauw.be/post/2017/08/01/building-a-scheduled-cache-updater-in-aspnet-core-2.html (Accessed: 7 December 2018).
Bogard, J. (2018) 'MediatR - Simple, unambitious mediator implementation in .NET'.
Cache in-memory in ASP.NET Core | Microsoft Docs (no date). Available at: https://docs.microsoft.com/en-us/aspnet/core/performance/caching/memory?view=aspnetcore-2.2 (Accessed: 7 December 2018).
Dabai | Dabai (no date). Available at: https://dabai.dk/ (Accessed: 7 December 2018).
Daniel Roth, Rick Anderson, S. L. (2018) *Introduction to ASP.NET Core | Microsoft Docs*. Available at: https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-2.2 (Accessed: 7 December 2018).
Development, F. (2003) 'Feature-Driven Development', *Development*. doi: 10.1007/1-84628-262-4_9.
Fluent Assertions - Fluent Assertions (no date). Available at: https://fluentassertions.com/ (Accessed: 7 December 2018).
Fowler, M. *et al.* (2002) *Patterns of Enterprise Application*, *I Can*.
'Handlebars.Net' (2018).
IConfiguration Interface (Microsoft.Extensions.Configuration) | Microsoft Docs (no date). Available at: https://docs.microsoft.com/en-us/dotnet/api/microsoft.extensions.configuration.iconfiguration?view=aspnetcore-2.1 (Accessed: 7 December 2018).
Introduction to ASP.NET Core SignalR | Microsoft Docs (no date). Available at: https://docs.microsoft.com/en-us/aspnet/core/signalr/introduction?view=aspnetcore-2.2



(Accessed: 7 December 2018).
Keval Patel (no date) *What is Reactive Programming?* Available at: https://medium.com/@kevalpatel2106/what-is-reactive-programming-da37c1611382 (Accessed: 7 December 2018).
Lasorsa, Y. (2018) 'Introduction to Angular'. Available at: https://gist.github.com/sinedied/214d95c496634a01ab6e0f648e24a22f.
Lukas Marx (2016) *Is Angular 2+ MVVM?* | *malcoded*. Available at: https://malcoded.com/posts/angular-2-components-and-mvvm (Accessed: 7 December 2018).
Mark Seemann (no date) *AutoFixture as an auto-mocking container*. Available at: http://blog.ploeh.dk/2010/08/19/AutoFixtureasanauto-mockingcontainer/ (Accessed: 7 December 2018).
'Moq4' (2018). Available at: https://github.com/Moq/moq4/wiki/Quickstart.
NUnit.org (2018). Available at: http://nunit.org/ (Accessed: 7 December 2018).
Robert C. Martin (no date) *ArticleS.UncleBob.PrinciplesOfOod*. Available at: http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod (Accessed: 7 December 2018).
Schwaber, K. and Sutherland, J. (2013) 'The Scrum Guide', *Scrum.Org and ScrumInc*, (July), p. 17. doi: 10.1053/j.jrn.2009.08.012.
Selenium - Web Browser Automation (2018). Available at: https://www.seleniumhq.org/ (Accessed: 7 December 2018).
Service-oriented architecture (SOA) (no date). Available at: https://www.ibm.com/support/knowledgecenter/en/SSMQ79_9.5.1/com.ibm.egl.pg.doc/topics/pegl_serv_overview.html (Accessed: 7 December 2018).
Testing with the TMap Suite (2018). Available at: http://www.tmap.net/ (Accessed: 7 December 2018).
'WireMock.Net' (2018).



11 Appendices

11.1 A – Feature List
11.2 B – Choice of Models and Methods
11.3 C – Feature Roadmap
11.4 D – Forecast System API Description
11.5 E – Test Specification Document
11.6 F – ForecastMonitor UI Architecture
11.7 G – Use Case Descriptions
11.8 H – API Design
11.9 I – ForecastMonitor Service Architecture
11.10 J – Feature Description Documents
11.11 K – Sprint Reports
11.12 L – Sprint Planning and Burndown Charts
11.13 M – Implementation
11.14 N – Process Report