MGP

Medical Gateway Platform

Axel Ország-Krisz Dr.

Richárd Ádám Vécsey Dr.

Table of Contents

Summary.....5

Background.....7

The problem and need.....7

The born of the idea.....8

Type of ventilators.....11

Breathing types.....12

Breath sequences.....12

Ventilatory patterns.....12

MGP – Medical Gateway Platform.....13

Basics of the idea.....13

Architecture.....14

Structure.....16

Data.....17

Types.....18

Operations.....20

UX.....24

Personas.....24

Developer....24
Doctor....25
Patient....26
Beneficiaries....27
Developer....27
Medical worker....28
Patient....28
Examples and use-cases....29
Business plan....34
Market analysis....34
Competitive analysis....36
Sale and marketing strategy....37
Financial plan....37
Capital....38
Cost....39
Income....39
Complete documentation....40
The concept of the code....40
Classes....43
Common Classes....43

ExitObject....43

MGPData....44

MGPDataRowL0....45

MGPDataRowL1....47

MGPDataRowL2....49

MGPDataRowL3....51

MGPError....52

MGPFilter....52

NullPipe....53

Area Classes....54

AreaOfGates....54

AreaOfGates.GateObject....60

DataArea....60

DataArea.__DataAreaEngine....61

DataArea.__EngineCSV....63

DataArea.__EngineDoF....64

DataArea.__EngineSQL....66

InputArea....67

InputArea.InputHandler....69

OutputArea....70

OutputArea.OutputHandler....71

ProcessArea....72

Unique and special solutions in MGP....74

Defensive coding....74

Singleton....74

Protected member variables, functions and classes....74

Cycle time versus instantiation time....75

Demangling....75

Piped communication....76

Junction optimized loops....76

Customizable and regulated loop intervals....76

Advantages of our idea....77

Future plans....78

About us....79

cheXrad....80

DOF – Deep Model Core Output Framework....83

COVID-19 dataset....86

COVID-19 map....87

COVID19CasesWorldwideDatasetFromECDC....88

Summary

Medical Gateway Platform is a data driven solution to connect medical hardware or equipment and to support real-time secure and private data sharing.

Our goal is to solve a real life problem with a simple, but scalable solution. Furthermore, it is important to make a permanent solution that exists after the virus too.

COVID-19 epidemic causes serious shortage of ventilators. Companies from healthcare industry are not capable to satisfy the high demand. People have made open source ventilator projects around a world. Almost every day borns a new idea of how to build a ventilator from scratch. Even if one of them is a very flexible solution with a great usage possibility, there is a great chance to disappear in the dark without any notice. It seems, coordination is necessary and inevitable. However, thinking in the old box like legal regulation, forced cooperation, chambers or panel conversations is a dead end. How can we help developers to build easier, share their knowledge and be a part of a worldwide community? We have to widen their potential by providing a highly scalable coding framework. With MGP the coding process is short, fast and easy. Compared with traditional coding, the working code is some click away in MGP.

We use labeled input and output dataflow to provide the flexibility. Even if it have to rebuild a device due to lack of parts, the code still remain reusable. If the new part is in our database, some click is enough to make the code work. If there is a brand new part, interface is the only one slice of code that have to write.

Doctors are able to supervise patients data real-time. We provide clean and highly customizable user interface where every data of a patient are shown. Devices configuration and attributes can get changed quickly even remotely. Our software has built-in warning and alarm mechanism to be able to work in area where skilled crew are not nearby.

Both the equipment and the patient can receive unique ID so with MGP every ventilator – open source or industrial manufactured – can be part of the same remote maintenance and observation system in the hospital. MGP is able to implement outer legal and encryption filters for providing GDPR compatibility and defending health related sensitive data.

Patients have the chance to check their own medical status. The built-in warning and alarm system works non-stop that helps people to be calm. They know, the medical assistance is always one click away.

MGP implies the possibility of other benefits, if a developer implement the methods. For example, using multiple devices in a hospital can load the built-in oxygen system heavily. The software is able to decrease the overloading by optimizing the breathing sequences.

We decided to create a platform-independent way. That's why we write the framework in multiple programming languages from the beginning such as python, java and C++. There are some unique and notable coding solutions like defensive coding, singletons in python, protected member variables, functions and classes, piped communication, or demangling, that means type checking and using without importing the type itself.

Open source projects are not real competitors for traditional companies of the healthcare industry but the experience with low cost hardware and the know-how of home made design can improve their manufacturing too. The situation is just the same with the software. Differences in the field of performance or quality of hardware parts happen in normal processes too when manufacturers should change their suppliers. MGP has a clearly separated workflow, and a very flexible structure. Quick changes are not issues with it.

We try to invent modern solutions and aspects in a traditional and sensible field of life and industry.

Background

The problem and need

COVID-19 epidemic causes new ways of cooperation between people. Developers try to ease the hardware-need in healthcare. Due to the lack of ventilators, open source projects are born worldwide. We face with the need of coordination. However, we are EU residents, the solution is not based on regulations, chambers or organizations. The key is providing a proper software system to help developers to build easier.

There are two major methods to coordinate groups or people. First of all, there is a regulation-based solution. Maybe it seems reasonable to force the builders of open source ventilators to work together and share their own knowledge with each other, this view is missing a very important part. Hardware and software developers have existing channels to cooperate such as GitHub, Facebook groups, mailing lists. Besides making new rules is not necessary, it can be counterproductive. Regulation maybe keeps people away from development. More than 60 open source ventilator projects exist around the world and we think, there is nothing wrong with it. Building the next one maybe seems wasting of resources such as time, money and parts but this view is a dead end. A prosper-oriented kind of thinking about way of invention can lead us to new and more efficient hardware. Instead of trying to coordinate these groups with regulation, we provide a better solution. Making a common platform helps the groups make the prototype work faster by ease and shorten the coding process.

The lack of supplies is the other part of the problem. This shortage is caused by two factors: first, companies, hospitals, inventors buy up the part of ventilator and second, increasing of shipping-time. Even if we can be build a new machine by different blueprints, it cannot work due to the missing operator software. Using a general software as core driver prevents this kind of errors and allows to build a new machine by using different parts.

The born of the idea

We want to solve a real life problem and help to fighting against COVID-19 epidemic. One of the most important goal was to make a permanent solution that exists after the virus too. Other major target was to achieve the things above with a simple but scalable way. Due to the epidemic has not just social impact but economical also, we decided to create an open source solution that are able to make profit us and our contributive partners at the same time.

We are deep learning developers who like to communicate other developers and programmers via GitHub, Private Groups, forums and e-mail. We know, building a reliable ventilator is much more harder than writing the code. A lot of industrial companies announced, they want to build ventilators. Unfortunately, this is not the way of things works. It cannot be a valid case making cars or other goods and turning on a switch and the next item will be ventilator. The code and coding process is different. If we want to oversimple it, every code is only a "Hello world!" script. However, coding process means connect sensors with actuators that in this special case is for example oxygen pump or the pushing part of bag valve mask.

The "code" contains three parts:

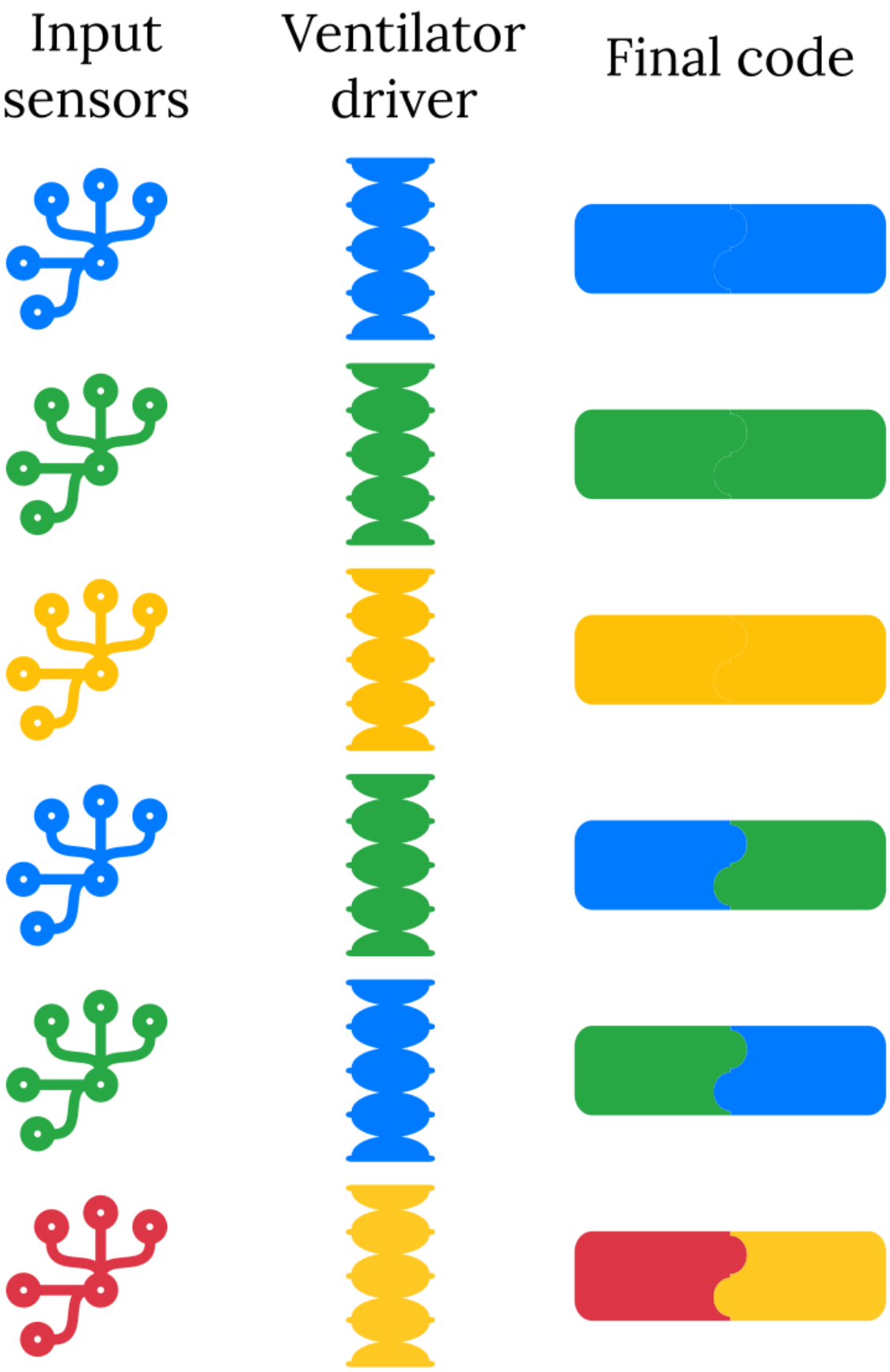
- getting sensor data
- converting sensor data to driven signal
- create driven signal to actuator

The first and third parts regularly come with the hardware driver. That's why the programmer have to write only a connection or inquiry layer. The middle part have to be coded by own. It can be interpret as a bridge to the first and third part. If we want to swap the main part of ventilator to an other, we have to modify the code. Different devices come with different software and driver structure.

A general gateway layer can help developer to make their own invention to be flexible and re-edifyable. We provide an input and an output API that waits for and sends labeled data. In addition, we provide a rule-based process and gateway system that helps to transform this data from raw format to a controller signal. Due to this gateway and process system anybody is able to reedit or to fine tune an existing code. We mean literally anybody, not just the original programmer, but doctor or other developer. Switching a part of ventilator can be easy with our system, since it means only some clicking in the coding interface and the new code can be ready to work.

The source of that simplicity leans on two major principles; fully separated structure, and fully connected data-flow via labeling. Fully separated structure means that our software have 5 different part which are connected via data-pipes only. Each part have limited responsibility, therefore each part have clear role in the whole process. Data flows between parts in a strict way and with chained logic. Strict way mean that the flow of the data have fixed directions and chained way mean that the data have process pointers. Process pointers brings the possibility of developing the running of a system further by analyzing its workflow. Data itself can be messy and requires a lot of attention and deep understanding to work with in in complex processes or at a higher scope. Therefore we decided to use labeling. Data labels are clear. The developer just shoots a glance at a data element in our system and the label tells everything. On the other hand labeling connects all the projects together where the logic of the workflow is the same because they all should use the same labels to identify their data and events. Therefore newcomer developers can more easily build or improve an existing project with the help of Medical Gateway Platform.

Picture on the next page shows a culture-independent visualization about how you can imagine a drag and drop coding that connects the part of devices and provides the changeability of those parts.


Input sensors
Ventilator driver
Final code

Type of ventilators

Mechanical ventilators can be grouped by various viewpoints:

- Positive-pressure ventilation
  - Invasive
    - Endotracheal intubation
    - Tracheostomy
  - Non-invasive
    - CPAP: Continuous Positive Airway Pressure
    - APAP: Autotitrating (adjustable) Positive Airway Pressure
    - BiPAP: Bilevel Positive Airway Pressure
- Negative-pressure ventilation
  - Iron Lung
  - Cuirass

However, our solution is able to handle any type from above, open source ventilator generally are CPAP. Some of more sophisticated devices are other type of non-invasive positive-pressure ventilators.

Endotracheal intubation means that through the nose or mouth, the tube is inserted into the patient's airway. Tracheostomy is when the tube inserted to the patient's airway through a hole made into directly to the airway.

Breathing types

- Volume control
- Pressure control

For breathing the device can control the volume or the pressure. Both of them is a valid choice, it depends on the details of device and its parts and measuring sensors.

Breath sequences

There are only three basic breath sequences:

- CMV
- IMV
- CSV

CMV does not allow spontaneous breath, while IMV allows all spontaneous breaths to occur between mandatory breaths, and CSV means that all spontaneous breaths.

Ventilatory patterns

Ventilatory patterns comes from mixing the two types of breathing type with three types of breath sequences:

- VC-CMV
- VC-IMV
- PC-CMV
- PC-IMV
- PC-CSV

MGP – Medical Gateway Platform

Medical gateway platform is a data driven solution to connect medical hardware or equipment and to support real-time secure and private data sharing. It helps to developers, medical staff, patients and data scientists at the same time.

Basics of the idea

Our idea is a software that helps to make work the open source ventilators. However, this is just one of a possible use-cases. Doctors, nurses or operators are able to monitorize the working process. since it handles and contains data to operate medical equipment. This platform is a general solution that means it can be used not only for ventilators, but for any other type of medical devices.

We provide a unique filter solution to handle the data that comes in the system. Handling can be conversion, encryption or any possible way of modification. These rules are written by developer but there is a chance to edit or tune later, even if during a real life operation.

Due to health-data is a sensitive and protected data, we provide the possibility of placing legal and encryption filters. Encrypting the data is useful for example when the patient is a highly protected person or the usage is related to any medical experiment. Legal filter grants to export the stored data without breaking any kind of legal regulation such as GDPR or act of countries about usage of health or personal-related data.

Leverage of this concept is that Medical Gateway Platform will exist and prosper after the COVID-19 epidemic. The collected data may be base of future development by companies from industry. Due to platform has ability to accept any kind medical device, it is a useful accessories of every hospital or healthcare centre. Facing with the next epidemic after COVID-19 is easier, if people use general platforms like our idea.

Architecture

The platform has 5 big parts:

- input handler area
- process area
- gateway area
- output handler area
- data storage and storage handler area

Input handler area is responsible to translate sensors and states data to the form which the next parts understands. This part consists of code which plays the role of a driver. When a developer finishes setting up the project physically he will know which sensor or switch on which port or device communicates. Those things can be very different even in the case when two persons builds the same project. One uses pin 10 on a Raspberry Pi another uses pin 20 for the same task. Those questions are more technical than important things. However from the view of maintaining the machine having a good overview is essential. Input handler area converts that overview to labels and in fact to real code. The developer can implement the method how to read and get the data from hardware and he should label the data to give the meaning to it. In this are filtering as available too. With filters the developer can control what to send to the main maintenance flow, to the data storage and to the user interface. Filtering is not necessarily since small project doesn't need it at all but industry level, international or global solutions require that kind of possibilities. Therefore we give the ability of adding filters but doesn't require it.

Process area works with already labeled data only. It is a higher level, much more abstract part. The responsibility of this area is to prepare all the date for the gates. This process means two tasks. First one is to group the data by the needs of each gate

respectively. Second task is to process the data, unify, transform, whatever the need of the gate is. In this area filters are available but not required as earlier. Processing area can send data to main flow, data storage flow and to the user interface as well just like the previous area.

Area of Gates is the part of the software that actually decides what and when to do or do nothing. The responsibility of this part is the decision only. Area of gates differs a bit from the earlier parts because built-in automation is available here too. Every gate have two input channel. One is based on the processed data from the previous area and the other is based on automated data like PWM signal or a Sinus signal for example. The gate has its own logic to decide about the importance of signals and of data. Everything in this area is free to customize. Also filters are available here and data can be sent to data storage and to user interface as well.

Output handler area is more like place for drivers just like input handler area. The responsibility of this part is to translate labeled data to signals understandable for actuators. The response about the result of action – if there is any result directly from the actuator can be sent to data storage or user interface. In this area filters are also available.

Data storage and storage handler area is where the data get stored and where external requests can be served. The responsibility of this part is to accept and serve data. Data storage can be DoF, CSV and SQL at the moment but more data storage systems can be implemented as well. Storing all the data is useful to improve projects. Data can be not just saved but sent as well. With the help of Medical Gateway Platform doctors can view the state of patients remotely they can even encroach if its needed.

Structure

The fundamentals of Medical Gateway Platform are solo machines, they can be computers or micro computers which maintain a medical device like a ventilator or they can be computers or cellphones of doctors or of other medic who controls patient's care remotely. Moreover MGP stations can provide data to the government or to research institutes to be well informed for statistic or for research. All this can made with our software easily but in case if solo medical machine maintenance is the goal only, it can done with it too. Therefore we think, our solution is powerful enough to create a platform which is appealing enough for existing and potential developers to cooperate with the help of it. Moreover we think, companies from the medical industry will see the advantages of using our solution too.

In local scope MGP provides a low level input and output interface to communicate with sensors, switches and actuators. All the data in the flow should have label so having a clear overview is always possible. There is a much higher level functionality based on low level grounds too. Since every input data is labeled and have a timestamp, exact processing of the data is very easy. Processor function can be even simple arithmetical or conversion logic as artificial intelligence based module. The next step of the flow is real decision where gate based algorithm decides to act or do nothing. The conception of gates shows very human alike thinking pattern because from labeled data it decides to act or to do nothing. The potential actions are also very straightforward since they have their own labels too. Due to needs of medical equipment the structure of this area also includes automated signals like PWM templates or Sinus curve signals. Those tools are good to implement simple – even printed circuits alike – logic.

At an even higher perspective MGP also shows its strength. It provides simple yet fully configurable user interface and data management at the local level.

However till now we spoke about machine maintenance as local level feature Medical Gateway Platform can act like a server for remote observing and controlling as well. It is the same software but with different setup. The connection can be established on LAN or via Internet too. Since MGP allows filtering at any step of data collection and data service full privacy and encryption can be solved wherever it is needed.

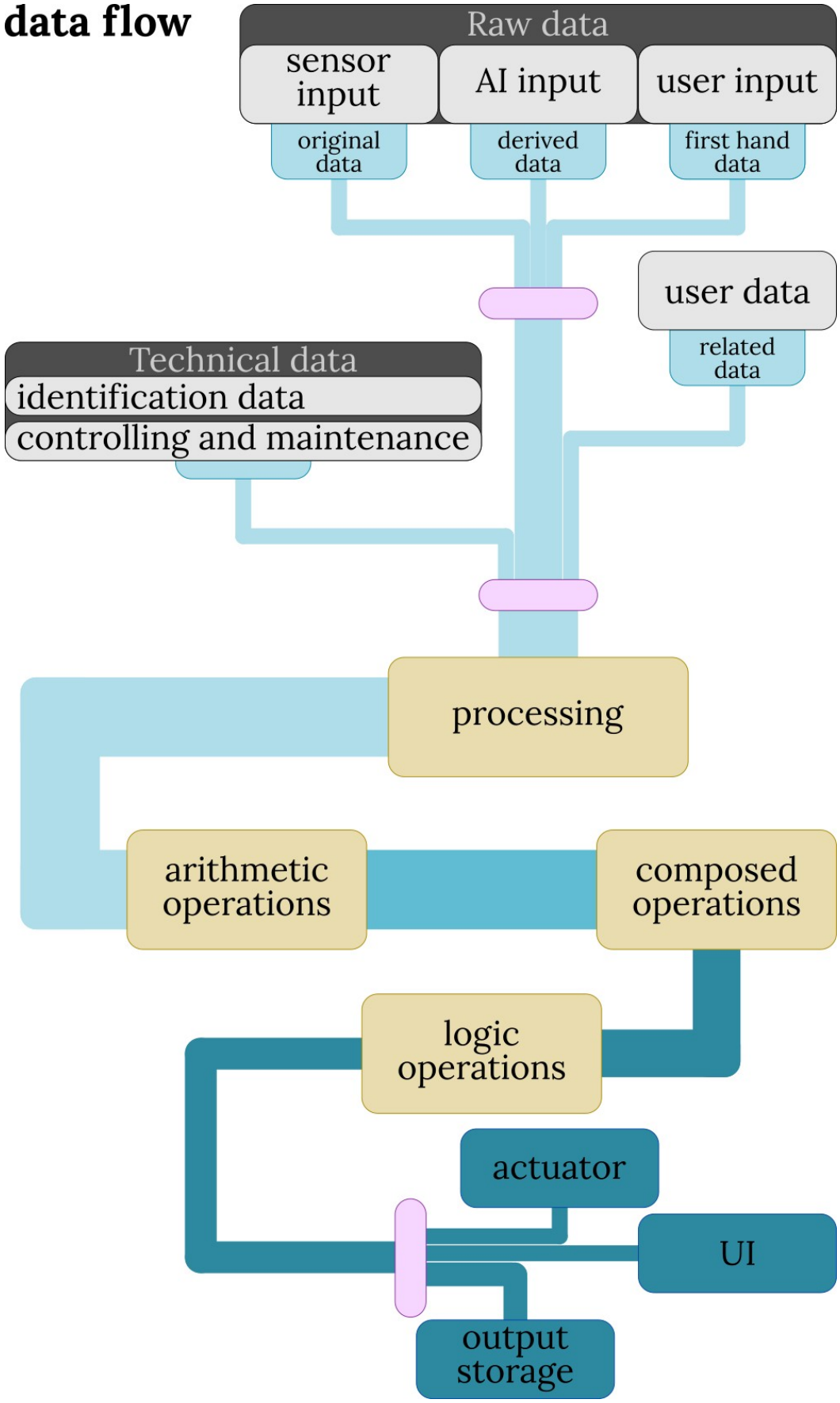
To manage remote connections Medical Gateway Platform doesn't require central service, the whole process can be solved like decentralized network.

Data

There are some way to examine the handled data like types, based on feasible operation. Categorizing data is not for just a scientifically trick for time-killing. However, each input is "data", there are a lot of differences between them. Some of them like timestamp can be interpreted with other telemetry data. A single timestamp value is useless. ID of Patient is just a number, but it is capable to connect the telemetry with a unique and identified real-life person. This can be harmful for the patient. That's why this kind of data have to be used only on the necessary place and time. Generally, we try to grant the highest protection to data. That's why we add options to filter and encrypt the data. The main output format is dof which is a highly scalable dataset format. As we present below, we provide the possibility of using encryption and legal filters. This results not only GDPR compatibility, but it fits in any regulation about data protection. However, encryption is useful when the goal is not the legal handling, but the physical protection of data.

Types

- 1. Input
  - 1.1. Telemetry
    - 1.1.1. Raw data
      - 1.1.1.1. Sensor data
      - 1.1.1.2. AI data
      - 1.1.1.3. Input data from user
    - 1.1.2. Technical data
      - 1.1.2.1. Identification purpose
      - 1.1.2.2. Controlling and maintenance purpose
  - 1.2. User data
- 2. Output
  - 2.1. Actuator signal
    - 2.1.1. Trigger based
    - 2.1.2. Cycle based
    - 2.1.3. Limited
      - 2.1.3.1. Pressure controlled
      - 2.1.3.2. Volume controlled
      - 2.1.3.3. Mixed
  - 2.2. UI data
  - 2.3. Stored or exported data


data flow
Raw data
sensor input
AI input
user input
original data
derived data
first hand data
user data
related data
Technical data
identification data
controlling and maintenance
processing
arithmetic operations
composed operations
logic operations
actuator
UI
output storage

Operations

Operations can be grouped by different patterns. Method means types of using the data. Routing means how the data flows in the system. Value modification is a highly axiomatic name, it embraces all the possible type of modifying from easy calculations till advanced logic operators.

- method
  - processing
  - storing
  - visualizing

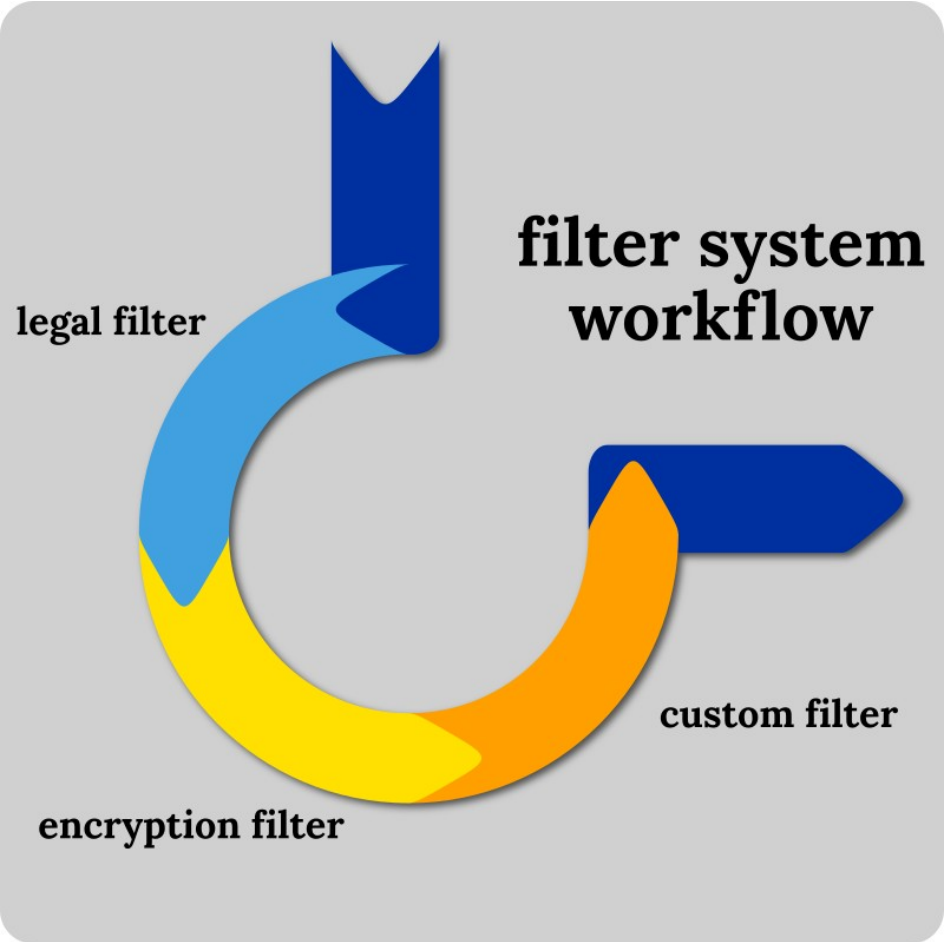
The most important part of a computer software is its relation with the data and the flow of data. Some most important programming languages treat data like streams which aim to mean data always flow. There are various kind of data and it is very common that the original input form of it does not fit the needs of a specific part of the code. Or on the contrary, the code does not need the whole complex data just a part of it. Therefore processing is basic building block of software.

If we transformed data with a reason it seems to be an obvious step to store it. In computing working with data always involves some kind of storing since data appears at various locations of a machine. Those storage are holding data for a short while only. Long term storage must be separately implemented and it must good planned too for example because of many legal aspect of data.

Visualization is also important due to human nature. Colors, curves, sizes say in most cases much more than simple numbers or rates. Due to the fact that every software works first and foremost with data every software need highly elaborated visualization techniques of it.

- routing
  - handling
  - filtering

The biggest difference between two components is that handling is an operation-driven processing, while filtering is a data-driven processing. Handling is necessary and the only necessary part to create a fully functional code. If it is created, the device are ready to work. However, it may be need to add something extra to the system. The cause is not the operation of device, but any human- or legal-related cause. For example, the hospital want to encrypt the data or it is must to cleaned the exported data from sensitive information due to GDPR. Handling is a response to the problems of mechanic, biology and information technology. Filtering is a response to the problems of society and legal systems.



- value modification
  - arithmetic operators
    - summation +
    - subtraction -
    - multiplication *
    - division /
    - exponentiation ^{x}
  - composed operators
    - high-passing
    - low-passing
    - encryption
    - noising
  - logic operators
    - AND
    - OR
    - XOR
    - <
    - >
    - only 1
    - only 2

Each of them is working by filters or handling scripts. These are modifying the workflow, the data transformations and the route of unique data points.

Arithmetic operators are well-known, widely used and very straightforward value modifiers addition, subtraction, multiplication, division and exponentiation doesn't need to be clarified. Composed operators are different. They are built from simple structures. High passing and low passing are simple composed operators since they work like gates. They let the value through if the value is higher (high-pass) or lower (low-pass) a specified limit and otherwise they block the flow.

Encryption is commonly used due to security reason. It codes data with an algorithm. The data is protected this way against unauthorized, incompetent entities. Medical Gateway Platform widely supports encryption on every level of its workflow.

The goal of noising is totally different from encryption. As long as encrypted data can be 100 % recovered, the goal of noising is to "ruin" data forever since a good noise have the same characteristics as the original data. Nosing is commonly used at dataset creation especially for data science and deep learning purposes. Just like encryption noising is also available in MGP at every level since it can added like a filter function.

Logical operators are the most basic building block of programming and electrical circuits. Therefore they are used in MGP everywhere but they have special importance in the gate decisions since channels of processed data and automated signals are connected via logical operators. Even though the operator AND seems to be familiar, in a gate there is 2 possible version of it depending on the question, in an "AND is True" case data of which channel should be sent further. The situation is just the same with OR. If both side is True it behaves just like AND. The operator XOR is not that widely used. It evaluates to True if a side is True but it is False if both is True or none is True at all. The operators ">" and "<" are showing importance. If there is a signal on the side which more important is, it decides. If there is no signal on that side, the sate of the other signal decides. In case of the operator ONLY only the state of the specified signal counts.

UX

Personas

MGP provides solutions for three different types of users.

Developer

Developer is an engineer or programmer who want to build an open source ventilator. They have the necessary knowledge to piece together the parts and make it mechanically work. A little bit coding knowledge is required, but not mandatory to have a degree or a similar education level. That means, understanding basic mathematical calculations and equations is an essential skill. They are enthusiast and dedicated to helping people during COVID-19 epidemic as soon as possible.

They facing with these problems:

- code writing is a difficult
- if one of the main parts have to be switch to a different type, the code must be rewritten
- testing mechanisms have to coded

The user experience that we provide with MGP:

- easy, drag and drop coding
- fitting the existing code to a new, replaced part of device can be achieved with some click
- fast testing with built-in testing mechanisms
- lint-alike error checking flow helps to analyze the errors in code

Doctor

The second type of typical user is the medical worker. However, it is a broad category, the ideal representation of group is a medical doctor. Other workers are able to use the dives, but doctors have a very unique skill. They can intervene to the medical treatment of patient because of knowledge and power. Furthermore they understand the human biological flows, pulmonary ventilation and COVID-19 disease. They help patient, but there is shortage of ventilators in hospitals. They have not enough time to learn the handling of each different type of open source ventilator. However, they are able to handle a similar system like user interfaces of common ventilators. They use smart devices and mobile phones. They have access to computers in the hospital. Besides this access, our typical doctor have basic knowledge about how to use a computer such as clicking with mouse and using menus in a program.

They facing with these problems:

- huge need of ventilator
- not enough time and energy to learn to use a lot of new software
- lack of coding skills
- open source ventilator rarely use automatic alarm and warning system

The user experience that we provide with MGP:

- selected devices and patient with some clicking
- clean user interface where all patients' data are shown
- plug-in and work solutions for device
- build-in warning and alarm mechanism
- code can be modified during real-time operation

Patient

Patient is the third type of major possible user groups. They are infected with COVID-19. They have bad conditions, for the survive and recovery they are on ventilation system in hospital. They are not in unconscious state. The patient is interested in own medical state and records. However, they want to defend own personal related and health data. Only one skill is necessary, such as handling smart device or the familiarity with computer programs. They are able to access computer device like laptop, tablet or at least smart phone.

They facing with these problems:

- lack of information about own medical state
- shortage of ventilators risks chance of survival
- being quarantine in hospital after ventilation till full recovery is boring
- do not understand the language and logic of medical device interfaces
- do not feel safety due to overwhelmed healthcare system and medical workers
- suffer from doctors does not have enough time to talk and support 0-24 control

The user experience that we provide with MGP:

- clean user interface
- only important data are visible
- easily check the stored data
- health data can be shared with other doctors or medical companies
- devices work non-stop and support full-time supervision
- easy to ask medical assistance via user interface

Beneficiaries

Beneficiaries of this project can be sliced 3 major parts. This related to the content of previous chapter, but we use a different scope. As you can see, the viewpoint is widen and focuses not only on the three main personas.

- Direct beneficiaries
  - Developers
  - Medical workers
  - Patients
- Secondary beneficiaries
  - Industrial companies
  - Data scientists
  - Other third party contributors
- Indirect beneficiaries
  - Hospitals
  - Healthcare systems of countries

Developer

MGP helps developer to create a working code faster. Our platform widen the number of possibilities for helping. Programmers without building skills can code template. Developers use these predefined templates to build a medical device. When any part of device have to be replace a new and different part, the new code can be finished with some click.

Medical worker

Doctors and nurses have the chance to observe not just only one device or one patient, but multiple device at the same time. The MGP has two different error function. Warning comes when the error is small and connects to the working flow, for example there is a data without saving. Alert pops up when the error relates to patient health and life. When the device is working mode and suddenly the input dataflow is disconnected, it throws an alert. Same as, when the device does not give back response from output.

Patient

Patients are be able to watch their own datastream. Furthermore, it is possible to share this data real-time. It does not matter where is the patient and the doctor. It must be difficult for somebody being in hospital in a foreign country, but he or she have the chance to share medical data with his or her doctor from home country. This process helps to the doctors to discuss about patient. New way of cooperation minimizes the risk factors during medical treatment.

Examples and use-cases

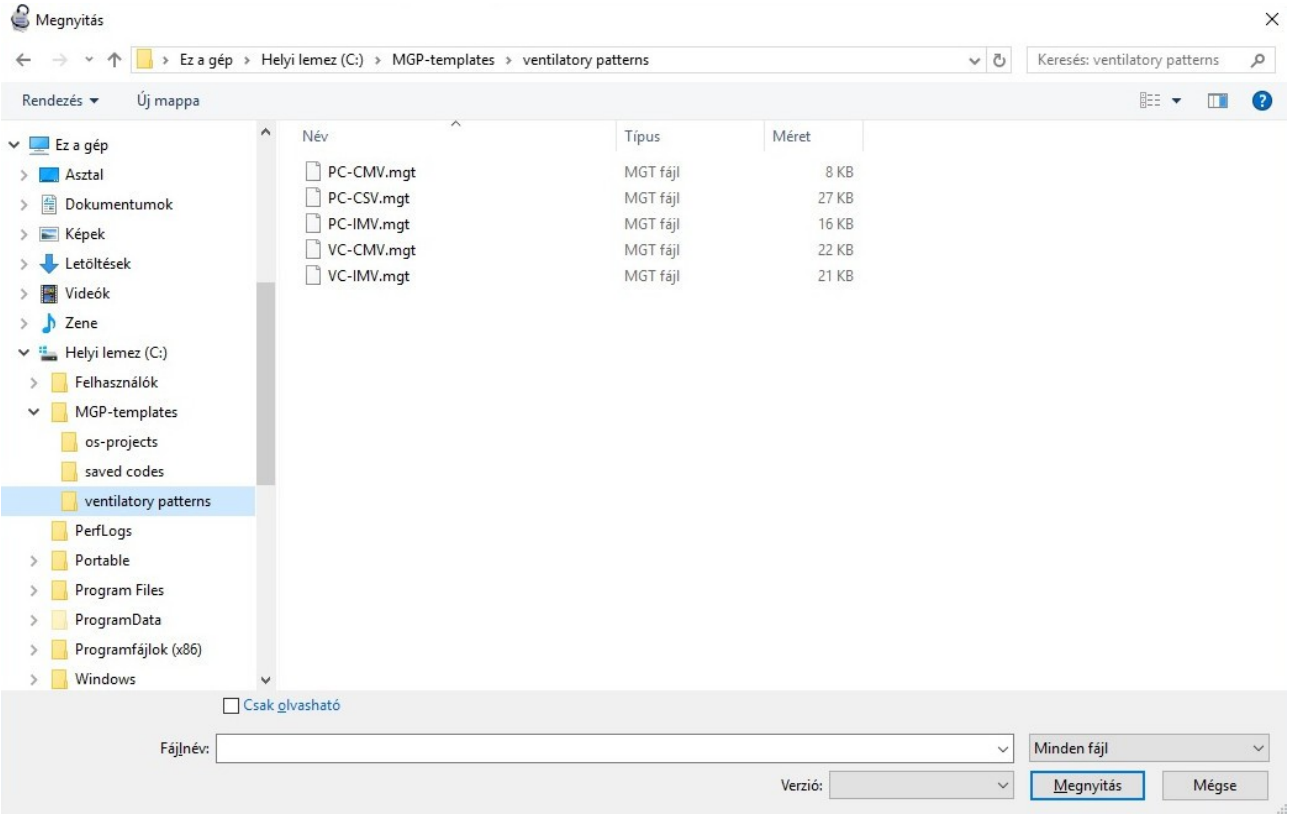
In this chapter we present screenshots about MGP. It implies the main screen, opening process, and example views of doctor, developer and a system administrator. We decided to show the most interesting or exciting parts of MGP.

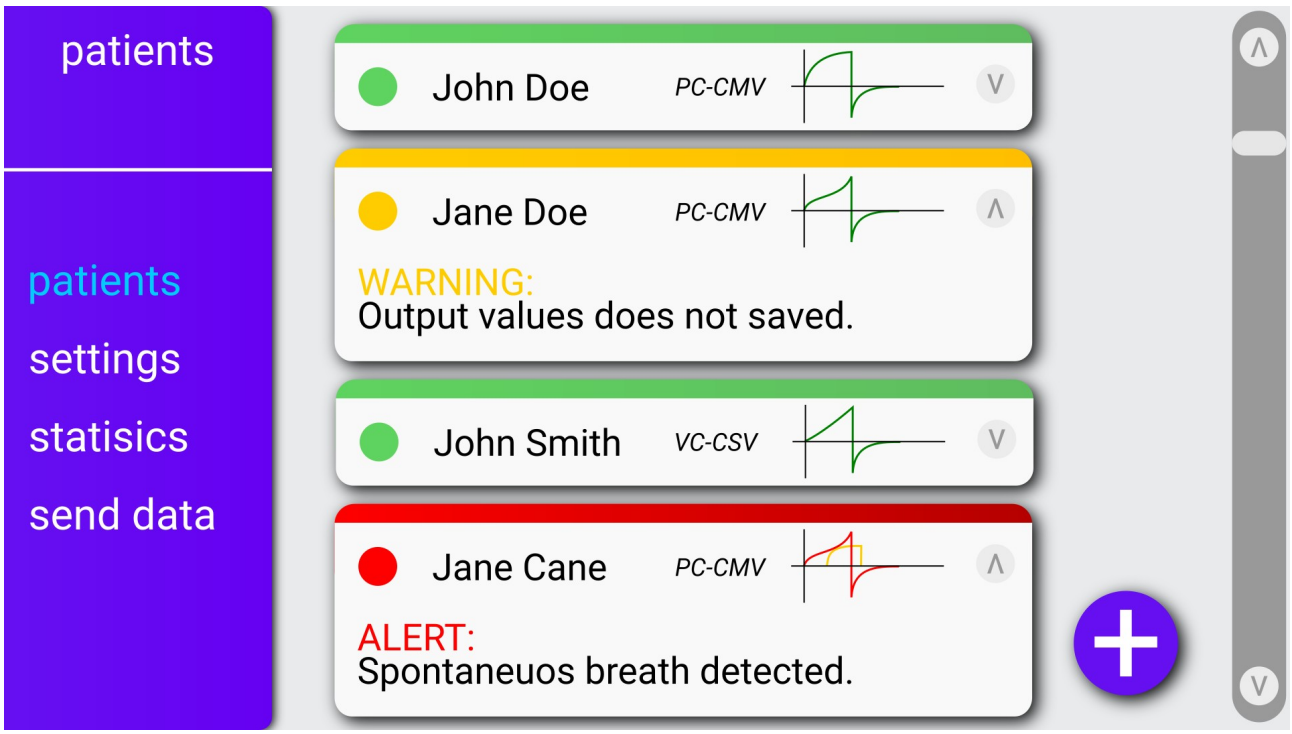
Doctors see patient-related data on one dedicated page with the most important features such as name, ventilatory patterns, breathing cycle graph and status bar for signaling okay, warning and alarm statuses. Data are shown on patient-related unique cards. Adding new patient is easy with the giant bubble button. These screens work not only desktop computers, but mobile devices as well. Huge buttons support the functionality during easy clickability to medical staff with gloves.

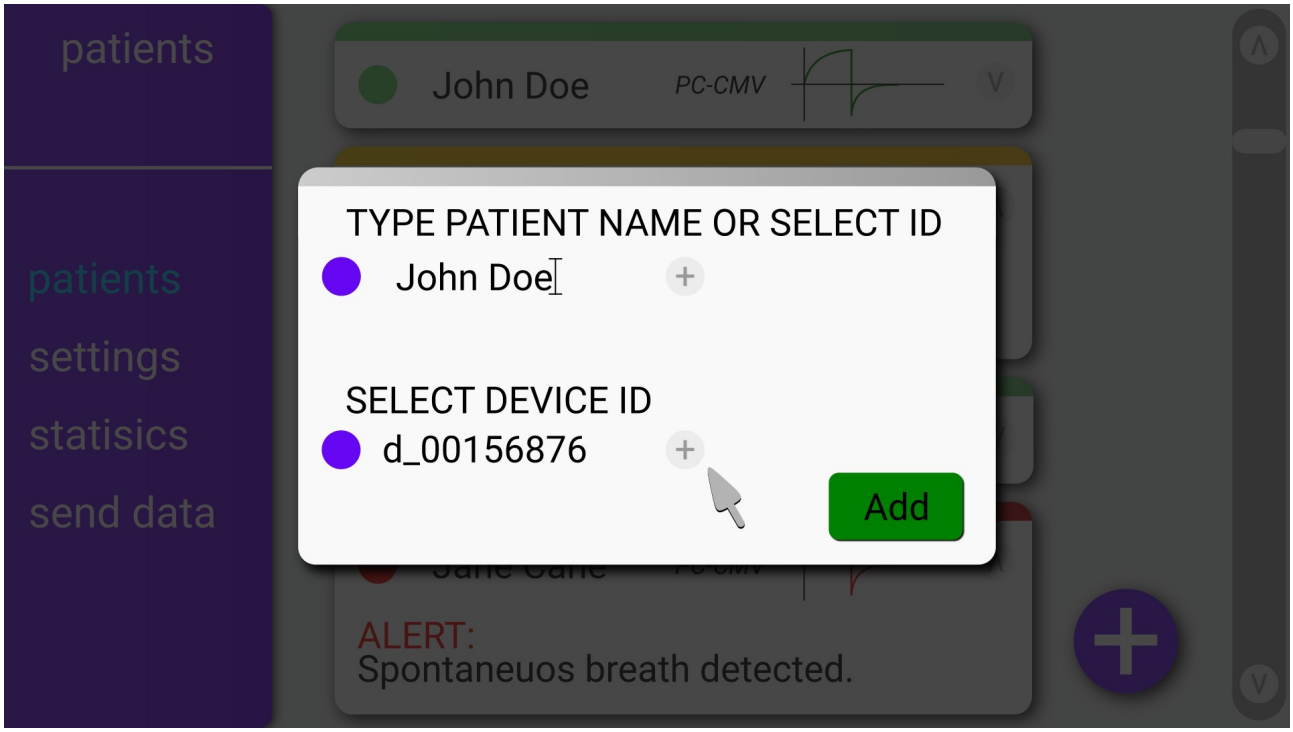
Skeletons of developers' screens are similar like interface for doctors but filled with different contents. There are multiple way to create new code. First of all there is a chance to load in an already existed open source project. Secondly, developers are able to choose the ventilation workflow only option. It is useful for designing a product or build code based on an empty guideline. Finally, hardcore programmers are able to write a brand new code from sratch. However, these are the main categories, we provide a wizard helping system even if they used a brand new code option. We help to anybody at anytime during the coding process. That's why we have a lint-alike feedback system. This helps to fix error faster and be patient safe.

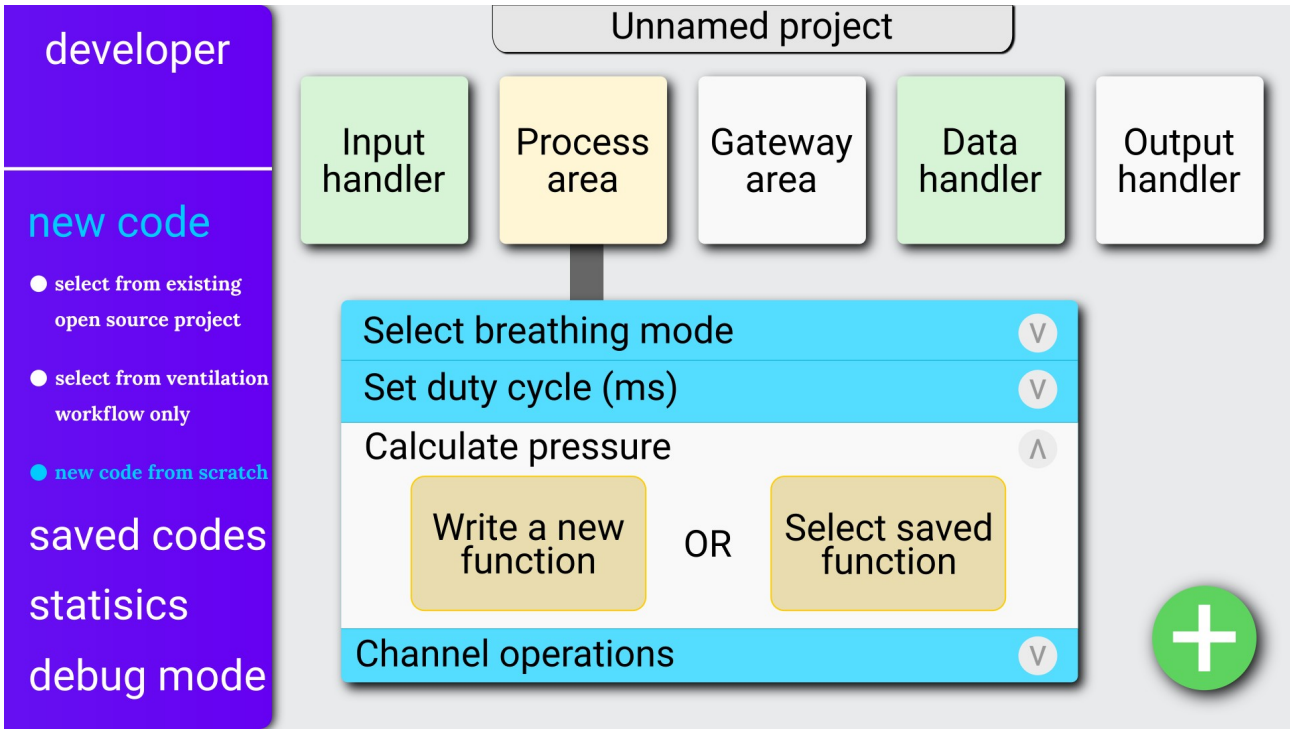
We decided to show a screen about one of our greatest side-invention: oxygen level optimalization. In a hospital where multiple device is used from a built-in oxygen source, devices can load the oxygen network unbalanced. The devices are able to optimize the breathing sequences for balancing the consumption. The process time depends on the number of devices and the type of used breathing sequences. If a developer adopt this functionality, different devices can be work together to release the most capacity as possible.

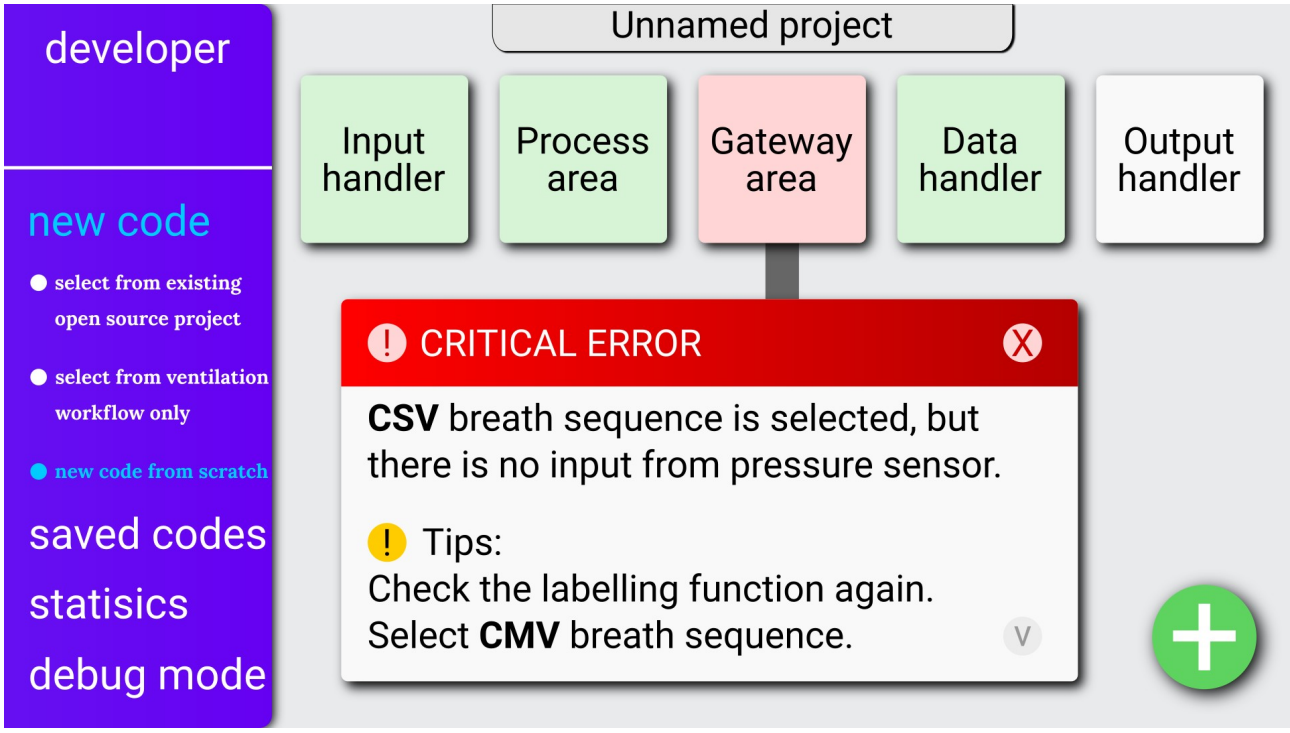
Medical Gateway Platform

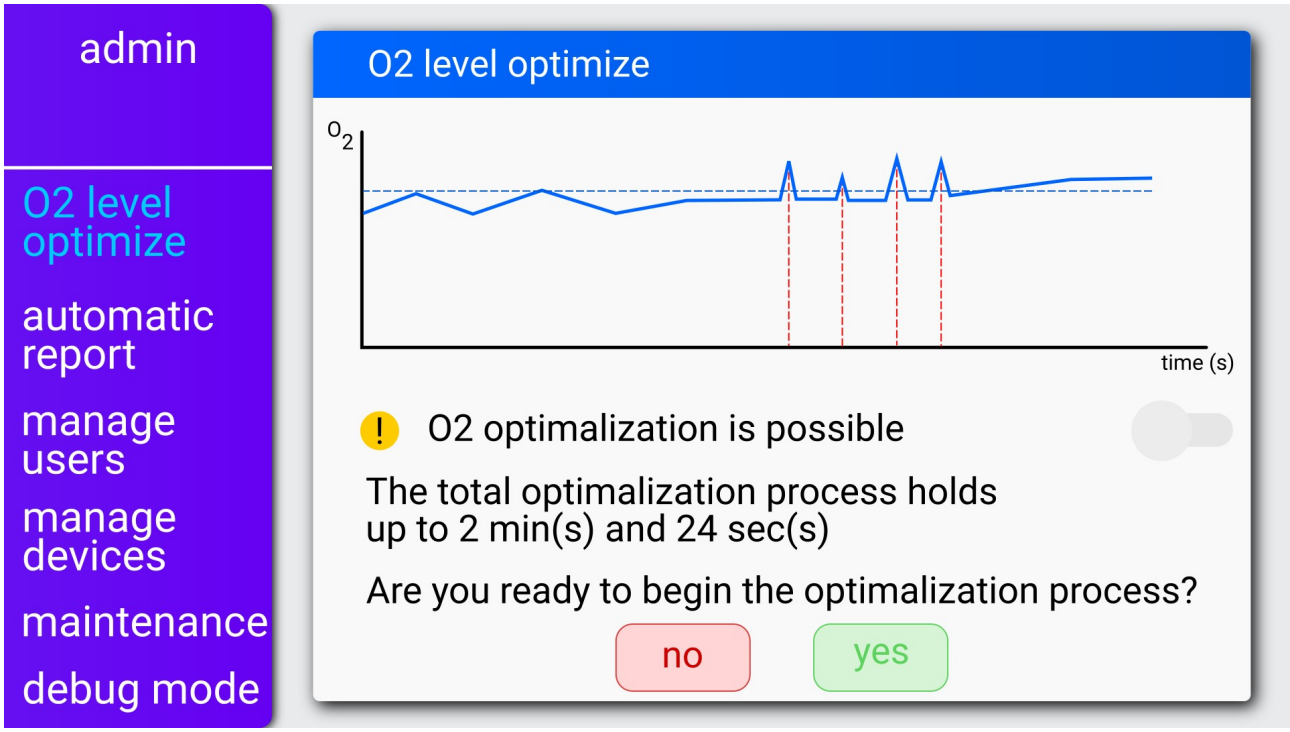

Megnyitás
Ez a gép › Helyi lemez (C:) › MGP-templates › ventilatory patterns
Keresés: ventilatory patterns
Rendezés
Új mappa
Ez a gép
Asztal
Dokumentumok
Képek
Letöltések
Videók
Zene
Helyi lemez (C:)
Felhasználók
MGP-templates
os-projects
saved codes
ventilatory patterns
PerfLogs
Portable
Program Files
ProgramData
Programfájlok (x86)
Windows
Név
Típus
Méret
PC-CMV.mgt MGT fájl 8 KB
PC-CSV.mgt MGT fájl 27 KB
PC-IMV.mgt MGT fájl 16 KB
VC-CMV.mgt MGT fájl 22 KB
VC-IMV.mgt MGT fájl 21 KB
Csak olvasható
Fájlnév:
Minden fájl
Verzió:
Megnyitás
Mégse

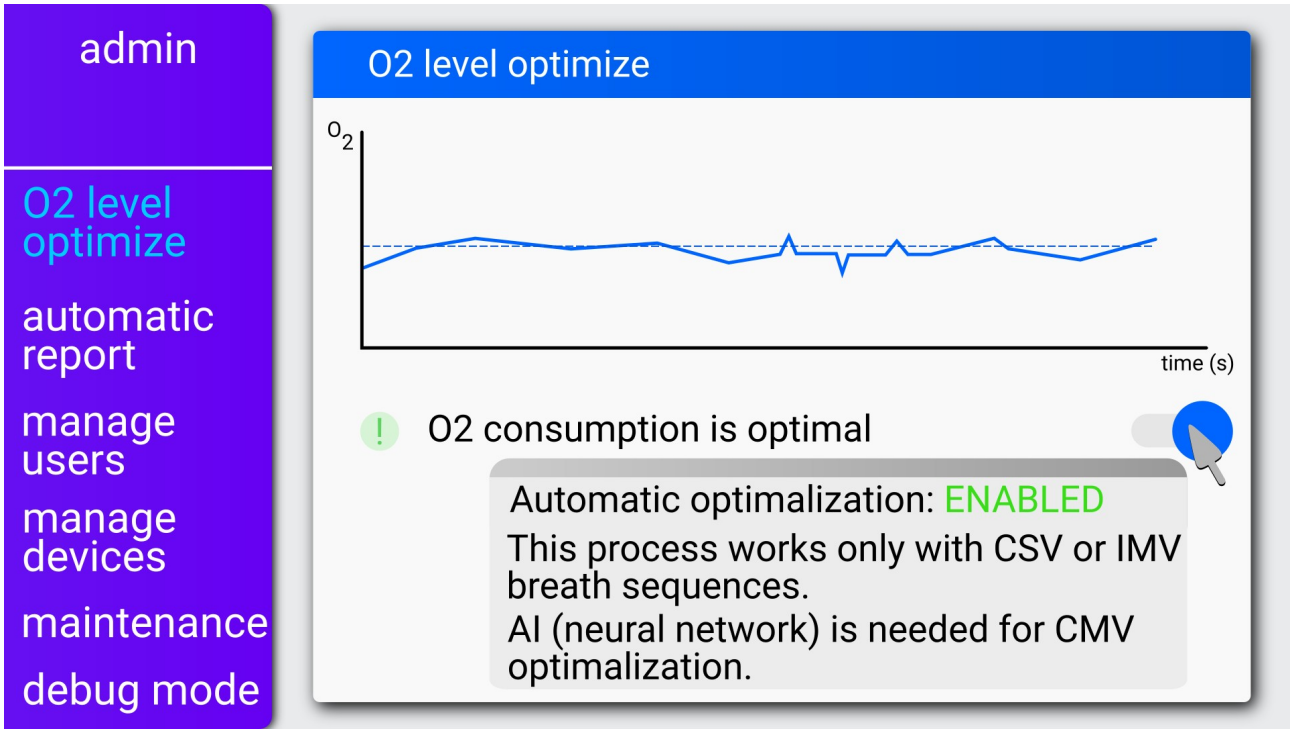

patients
patients
settings
statisics
send data
John Doe
PC-CMV
Jane Doe
PC-CMV
WARNING:
Output values does not saved.
John Smith
VC-CSV
Jane Cane
PC-CMV
ALERT:
Spontaneuos breath detected.


patients
patients
settings
statisics
send data
John Doe
PC-CMV
TYPE PATIENT NAME OR SELECT ID
John Doe
SELECT DEVICE ID
d_00156876
Add
ALERT:
Spontaneuos breath detected.


developer
new code
select from existing open source project
select from ventilation workflow only
new code from scratch
saved codes
statisics
debug mode
Unnamed project
Input handler
Process area
Gateway area
Data handler
Output handler
Select breathing mode
Set duty cycle (ms)
Calculate pressure
Write a new function
OR
Select saved function
Channel operations


developer
new code
select from existing open source project
select from ventilation workflow only
new code from scratch
saved codes
statisics
debug mode
Unnamed project
Input handler
Process area
Gateway area
Data handler
Output handler
CRITICAL ERROR
CSV breath sequence is selected, but there is no input from pressure sensor.
Tips:
Check the labelling function again.
Select CMV breath sequence.


admin
O2 level optimize
automatic report
manage users
manage devices
maintenance
debug mode
O2 level optimize
O2
time (s)
O2 optimalization is possible
The total optimalization process holds up to 2 min(s) and 24 sec(s)
Are you ready to begin the optimalization process?
no
yes


admin
O2 level optimize
automatic report
manage users
manage devices
maintenance
debug mode
O2 level optimize
O2
time (s)
O2 consumption is optimal
Automatic optimalization: ENABLED
This process works only with CSV or IMV breath sequences.
AI (neural network) is needed for CMV optimalization.

Business plan

The chapters of this project book contain the part of a common business plan. However, we decided to create a section about the financial relations of MGP and other market related analyses.

Market analysis

Typing “open source ventilator” searching words into GitHub's search field gets more than 90 results [1]. The challenges details on official site of EuvsVirus hackathon speaks only 60 teams globally [2].

According to nsmedicaldevices.com, about 880.000 additional ventilators will be needed globally. Share of USA from this demand is nearly 75.000 pieces, while together France, Germany, Italy, Spain and UK require additional 74.000 devices [3].

Three of the seven biggest medical ventilator manufacturers do not related to Europe [4].

- Becton Dickinson – USA
- Philips – The Netherlands
- Hamilton Medical – Swiss and USA
- Fisher & Paykel Healthcare – New Zealand
- Draeger – Germany
- Medtronic – Ireland and USA
- GE Healthcare - USA

Population of Europe are getting older and older. The number of people aged 80 or over will reach 66.1 million by 2080 according to European Hospital and Healthcare Federation [5]. However, it is in a far future, the trend seems inevitable. The basis of tomorrow healthcare system roots in today. Countries' healthcare related spending are very different. The private share of health expenditure in EU15 is significantly higher than in EU13.

In 2014 there were 521 hospital beds for 100.000 inhabitants in Europe. The earlier trend was a downgrading, the number of hospital beds decreased by about 7% between 2004 and 2014. [5]

There is an other problem. The last overall statistic report was dated in 2014. It is good for long-time analysis, but it is not serves the fast activity and planning. We provide a module to report real-time about patient to European institutes such as ECDC or inland ministries as well.

source links:

[1] https://github.com/search?q=open+source+ventilator (visit time: 24. 04. 2020)

[2] https://euvsvirus.org/challenges/ (visit time: 24. 04. 2020)

[3] https://www.nsmedicaldevices.com/analysis/coronavirus-ventilators-global-demand/ (visit time: 24. 04. 2020)

[4] https://www.nsmedicaldevices.com/analysis/seven-ventilator-manufacturers/ (visit time: 24. 04. 2020)

[5] http://www.hope.be/wp-content/uploads/2018/07/2018_Hospitals-in-EU-28-Synthesis-final-for-publication-002.pdf (visit time: 26. 04. 2020)

Competitive analysis

Every new service or product says about itself, it is a very unique solution without any competitor. It may be true, but in small pieces, there are always competitors. Even if they play in a bigger league. This is similar in MGP's case.

Our biggest competitors the ventilator maker companies. It does not narrowed to the healthcare sector, since during an epidemic crises, several company try to escape from a downward economic spiral with building medical related devices. It is have to be noticed, that the competitive opposition is only a surface. There is no competition at all. The healthcare industry has its own drivers and software. It has been worked very well. This COVID-19 epidemic points out that this attitude unsustainable. The capacities of healthcare industry are calibrated to the common demand, not for the peaking by caused an epidemic. Without a general driver solution, companies do not have the ability to satisfy this temporary demand. In a normal situation it means only to miss some profit, but this time it can be measured by number of patients' life. We see this relation as a possible way of contribution.

Open source ventilator builders are not competitors since we provide solutions for their problems. When a developer has ability and will to build a devices on their own skills, that is very welcomed. If they are really committed to the open source, they find our solution one way or another, and share their experiences and codes on our platform.

Driver providers can be competitors, but each of them focuses only its own product. By the way, these product does not equals ventilators. These are only the parts of it. It implies the core of cooperation as well. Some company builds a huge community to its product, such as Adafruit or Nvidia. This kind of thinking can be adapted to the market of healthcare related parts. When a driver provider supports our labeling method natively in its product, developers buy it more like due to easiness of development process.

Sale and marketing strategy

Although MGP is created for developers, programmers, doctors and companies from healthcare industry, we can take advantage of advertising and marketing, especially social marketing. That's why it would be good to advertise the product. This goal can be achieved for free, if big tech companies take in the project in their advertisements, or at least mention it on own Facebook channel or newsletter.

The more people hear about our platform, the more chance is to get us helped with talented programmers and developers. It is great not just us, but healthcare companies, since they can a solid base for recruiting.

Financial plan

When speaking about financial plan, there are two fields come out: first is the necessary capital and costs, and the second is the incomes.

We create our Medical Gateway Platform as an open source project, but it does not mean it could not make profit. We can sell preprocessed data for data scientist or we can help recruiting developers for tech companies. We try to provide our solution for free during the longest time as possible, but we would like to present, we are able to think in complex economical scenarios as well.

There are a lot of black area about what will be the world economy after the COVID-19 epidemic. We can be sure in two statement:

- The faster a company gets up from floor, the bigger chance to reach leading position.
- The industry will based on data, information and machines much more. Data give leverage, machines give freedom and resiliency to the company.

Capital

Due to our goal was to make a production ready solution, the project is ready to use. There is no need money investment at the start. However, if somebody or any company supports the project, it can grow faster. This support does not mean money or capital injection. Using our product and building in a production flow is more than enough. We know, companies are profit oriented organizations even if they are maintenance CSR protocols or they are committed to the humankind. This support consumes money, time and human resources, but it has 3 advantages:

- data
- free or cheap innovation
- possibility of join a bigger flow of goods

A huge amount of data arrive or be generated in our system. Building on these data can be the base of future inventions. Using a narrow view, companies are be able to optimize the production process and their products to reach a more profitable stand.

An open source platform always collects enthusiastic developers. MGP is able to attract not just hardware developers, but software programmers since the product is open to coding only solutions. This kind of thinking maximizes the collectable knowledge. Every type of developer can be a part of it. This knowledge is useful for industry. A lot of big company use the coding solutions or just slices from GitHub. Our platform can collect knowledge about medical related open source innovations.

Companies can be change an entire production chain from any type of goods to medical devices if the blueprints and drivers are provided. Our platform helps developers to make open source projects like ventilators or other medical related device due to providing simple solution for coding the driver software. Companies have the chance to produce these devices and machines or any part of them. Resilience comes profitability.

Cost

Costs are similar like capital. Due to the project is open source, ready to use and operates without central control, speaking about costs is meaningless. The platform are be able to run various devices and environments. The hospitals does not have to buy new computers or other expensive drivers. Hospitals with common computers can install and use the platform software immediately.

When a company tries to adopt the platform in its production chain, it has cost, that depends on multiple factors such as the modification size of production flow or the efficiency of flow management. It is important, this kind of costs connected to the decisions of the company, not us.

Income

In spite of open source project, it has the ability to make income and profit. The most evident source is data-selling. There are a lot of data scientist and researcher who would like to pay for properly complied datasets. The export mechanism can grant this. However, the beneficiary of this income can be the hospital if its employee make datasets. This platform is used by patient, so it has the possibility to connect patient to medical research company. However, there are established way to do that, the platform provides extra health data connected to the patient.

Open source projects regularly request donation and money support. However, it is possible to collect an enormous amount of money by donation, we think, asking donation during the epidemic crisis is inappropriate. There is a lot of area where money literally saves lives. For example, it is better for the community, if the money goes for building new ventilators. However, this source can be useful after the COVID-19 epidemic. If every person, who was treated by open source ventilator via MGP, gives only a little donation, we will be able to grow and expand the platform to new devices and illnesses.

Complete documentation

The concept of the code

This code is kind of a production ready prototype. From one view it is just a prototype since first of all it demonstrates the concepts of how medical equipment can maintained as code-less as possible. On the contrary from other view it is a production ready software since it can maintain open source ventilators from any platform of full-size computers, micro computers or mobile devices.

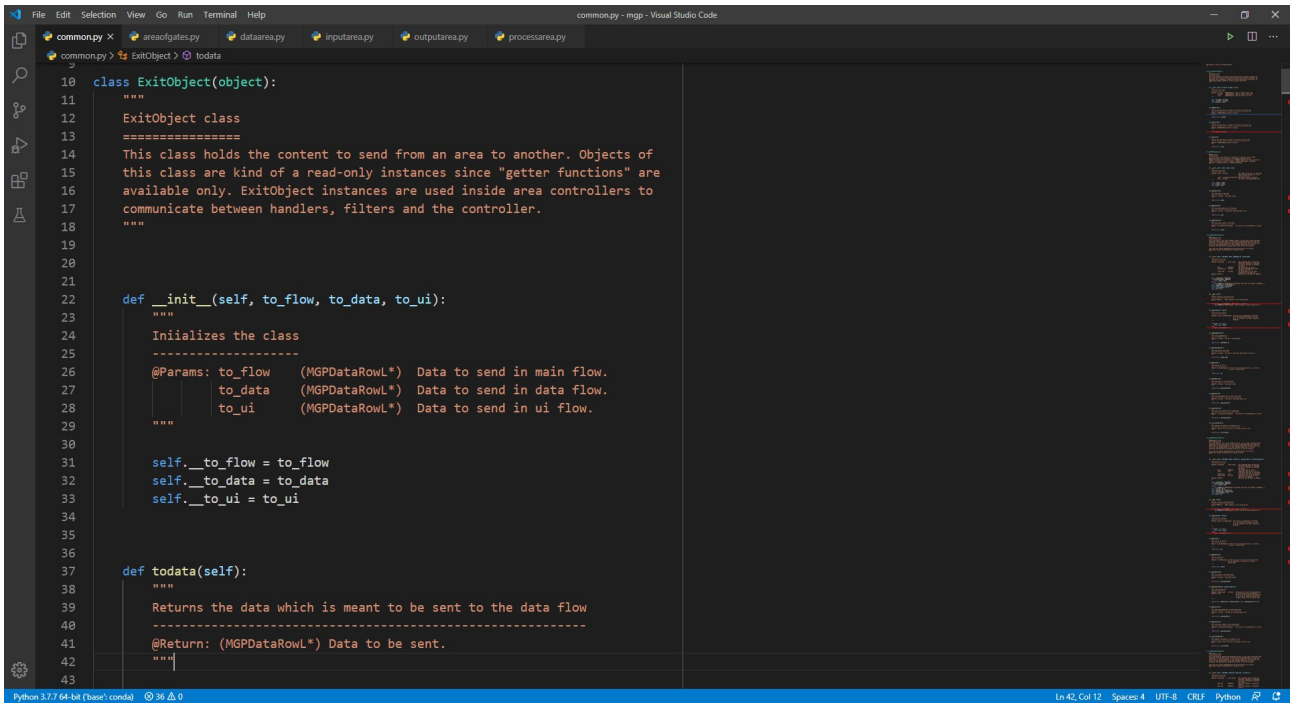
The production ready code is written in Python and the User Interface is written in combination of PHP, HTML5/CSS3 and JavaScript – more precisely mostly in Jquery. The reason of this is to provide as platform independent solution as possible. Python has another advantage too. To modify a Python code you don't have to have strong architecture skills and you don't have to care variable types, scopes, etc. Python can be installed easily and the – maybe newcomer – developer don't have to care about compiler directives or other mysterious things like that. The use PHP is not mandatory for end-users and developers can manage basic projects without it however using PHP on Linux based platform ain't a big issue at all. Simplicity of HTML5/CSS3 and JavaScript/JQuery codes is their strength too. Those codes can be used on any platform where browser is available. Therefore the production ready part of our project almost doesn't have hardware limits.

There are C++ header files and some Java code in our repository as well. The reason of publishing code in more languages is to demonstrate our devotion of developing this project much further. The scalability of the concept demands on worldwide use in the medical equipment industry because our solution has simple yet powerful surface and deep, fully controllable core at the same time.

Python is our leading language at the moment. The builders of open source ventilators are using custom circuits, Arduino or Raspberry Pi mostly. Although Arduino uses C we consider the solo basic looping architecture a dead end in further development. Circuits can provide basic and static logic only. At the level of a Raspberry Pi high variety of programming languages is available. Python is simple yet commonly usable programming language and the writing of a code in it doesn't require strong architecture skill.

At a deeper level of our code it doesn't seem to be typical python-ish one. The reason is because defensive coding is implemented as much as possible in this language. We think the concept behind the code is much more important in the future medical equipment industry than the this code itself. But we think too this simple code provides a fast easy-to-walk bridge between existing or even new open source ventilators right now. This is very important these days.

Python



Java

```java
public class ExitObject {

    private Object flow;
    private Object data;
    private Object ui;

    /**
     * Constructor L0
     *
     * @param to_flow   Data to send in main flow.
     * @param to_data   Data to send in data flow.
     * @param to_ui     Data to send in ui flow.
     */
    public ExitObject(MGPDataRowL0 to_flow, MGPDataRowL0 to_data, MGPDataRowL0 to_ui) {

        this.flow = to_flow;
        this.data = to_data;
        this.ui = to_ui;

    }

    /**
     * Constructor L1
     *
     * @param to_flow   Data to send in main flow.
     * @param to_data   Data to send in data flow.
     * @param to_ui     Data to send in ui flow.
     */
    public ExitObject(MGPDataRowL1 to_flow, MGPDataRowL1 to_data, MGPDataRowL1 to_ui) {

        this.flow = to_flow;
        this.data = to_data;
        this.ui = to_ui;
```

C++

```cpp
#ifndef EXIT_OBJECT_HPP
#define EXIT_OBJECT_HPP

template <typename MGPDataRow>
/**
 * class ExitObject
 * ================
 *
 * This class holds the content to send from an area to another. Objects of this
 * class are kind of a read-only instances since "getter functions" are
 * available only. ExitObject instances are used inside area controllers to
 * communicate between handlers, filters and the controller.
 */
class ExitObject {

    private:
    MGPDataRow flow, data, ui;

    public:

    /**
     * Constructor
     *
     * @param to_flow   Data to send in main flow.
     * @param to_data   Data to send in data flow.
     * @param to_ui     Data to send in ui flow.
     */
    ExitObject(MGPDataRow to_flow, MGPDataRow to_data, MGPDataRow to_ui);

    /**
     * Returns the data which is meant to be sent to the data flow
     *
     * @return  Data to be sent.
     */
    MGPDataRow getData();
```

Classes

Common Classes

ExitObject

class ExitObject(to_flow, to_data, to_ui)

This class holds the content to send from an area to another. Objects of this class are kind of a read-only instances since “getter functions” are available only. ExitObject instances are used inside area controllers to communicate between handlers, filters and the controller.

to_flow (MGPDataRowL0 | MGPDataRowL1 | MGPDataRowL2 | MGPDataRowL3) Data to send in main flow. This flow means the maintenance flow in Machine Level and background data analysis in Medical level.

to_data (MGPDataRowL0 | MGPDataRowL1 | MGPDataRowL2 | MGPDataRowL3) Data to send in data flow. This flow means the connection to DataArea usually.

to_ui (MGPDataRowL0 | MGPDataRowL1 | MGPDataRowL2 | MGPDataRowL3) Data to send in ui flow. This flow means the connection to the User Interface usually.

ExitObject.todata()
Returns the data which is meant to be sent to the data flow.
Returns MGPDataRowL0 or MGPDataRowL1 or MGPDataRowL2 or MGPDataRowL3
In case if data of this flow got filtered it returns None.

ExitObject.toflow()
Returns the data which is meant to be sent to the main flow.
Returns MGPDataRowL0 or MGPDataRowL1 or MGPDataRowL2 or MGPDataRowL3
In case if data of this flow got filtered it returns None.

ExitObject.toui()

Returns the data which is meant to be sent to the ui flow.

Returns MGPDataRowL0 or MGPDataRowL1 or MGPDataRowL2 or MGPDataRowL3

In case if data of this flow got filtered it returns None.

MGPData

class MGPData(label, value, unit)

This class holds the content of a data point in the most strict sense. Objects of this class are kind of a read-only instances since "getter functions" are available only. MGPData object are used to hold the content of an event in fact it means a labeled measurement, a labeled decision, a labeled action or a labeled something.

label (string)

The label of the data. It describes what the data actually is, like source or action.

value (string | int | float | bool)

The actual value of the data.

unit (string)

The name of the measurement unit.

MGPData.getlabel()

Gets the label of the data.

Returns string, the actual label.

MGPData.getunit()

Gets the measurement unit of the data.

Returns string, The name of the measurement unit.

MGPData.getvalue()

Gets the value, amount of the data.

Returns string or int or float or bool, the value of the measurement or action.

MGPDataRowL0

class MGPDataRowL0(timestamp, data, equipment_id, event_type)

This class holds a Zero Level database record. It acts like a read only data container from one side and it is an actual storage handling tool from the other side. As a data container it has "getter functions" only and data must be set at the instantiation. As a data storage helper tool it can be finalized with getting the storage record id or it can be filtered.

timestamp (time alike)
The timestamp when the data was generated. Generate a timestamp as soon as possible to minimize the delay.
data (MGPData)
The actual data to store.
equipment_id (string)
The ID of the equipment where the data are from.
event_type (string)
The description of the event connected to the data row.

This class has custom implementation of deconstruction. If tried to deconstruct without finalization it throws an error.

MGPDataRowL0.finalize(result)
Finalizes the instance.
result (int or NoneType)
The result of consumption of the data. If it is stored, the result is an ID. If it is filtered, the result should be NoneType.

MGPDataRowL0.getequipment()
Gets the equipment's ID.
Returns string, the ID of the equipment.

MGPDataRowL0.geteventtype()

Gets the type of the event.

Returns string, the name of the event type stored in the row.

MGPDataRowL0.getid()

Gets the ID of the row.

Returns int or NoneType, if the ID of the row already exists, it returns it, else it returns None.

MGPDataRowL0.getlabel()

Gets the label of the stored data.

Returns string, the actual label.

MGPDataRowL0.getunit()

Gets the measurement unit of the stored data.

Returns string, the name of the measurement unit.

MGPDataRowL0.getvalue()

Gets the value, amount of the stored data.

Returns string or int or float or bool, the value of the measurement or action.

MGPDataRowL0.is_filtered()

Gets whether the object is filtered or not.

Returns bool, True if the row is filtered, False if not.

MGPDataRowL1

class MGPDataRowL1(timestamp, data, l0_id=None, patient_id=None, patient_pwd=None)

This class holds a First Level database record. It acts like a read only data container from one side and it is an actual storage handling tool from the other side. As a data container it has "getter functions" only and data must be set at the instantiation. As a data storage helper tool it can be finalized with getting the storage record id or it can be filtered.

timestamp (time alike)
The timestamp when the data was generated. Generate a timestamp as soon as possible to minimize the delay.
data (MGPData)
The actual data to store.
l0_id (int) [optional]
The ID of an L0 row from which the flow is inherited.
patient_id (int) [optional]
The ID of the patient.
patient_pwd (string) [optional]
The password to get the ID of the patient.
This class has custom implementation of deconstruction. If tried to deconstruct without finalization it throws an error.

MGPDataRowL1.finalize(result)
Finalizes the instance.
result (int or NoneType)
The result of consumption of the data. If it is stored, the result is an ID. If it is filtered, the result should be NoneType.

MGPDataRowL1.getid()

Gets the ID of the row.

Returns int or NoneType, if the ID of the row already exists, it returns it, else it returns None.

MGPDataRowL1.getl0id()

Gets L0 record ID

Returns int or NoneType, if there is an ID of an L0 row from which the flow is inherited, it returns it, else it returns None.

MGPDataRowL1.getlabel()

Gets the label of the stored data.

Returns string, the actual label.

MGPDataRowL1.getpatientid(patient_pwd=None)

Gets the patient's ID.

patient_pwd (string) [optional]

The password the get the patient's ID.

Returns int or NoneType, it returns the ID of the patient if there is an ID stored and the password or the lack of password matches the original setup, else it returns None.

MGPDataRowL1.getunit()

Gets the measurement unit of the stored data.

Returns string, the name of the measurement unit.

MGPDataRowL1.getvalue()

Gets the value, amount of the stored data.

Returns string or int or float or bool, the value of the measurement or action.

MGPDataRowL1.is_filtered()

Gets whether the object is filtered or not.

Returns bool, True if the row is filtered, False if not.

MGPDataRowL2

class MGPDataRowL2(timestamp, data_ch1, data_ch2, l1_id=None)

This class holds a Second Level database record. It acts like a read only data container from one side and it is an actual storage handling tool from the other side. As a data container it has "getter functions" only and data must be set at the instantiation. As a data storage helper tool it can be finalized with getting the storage record id or it can be filtered.

timestamp (time alike)

The timestamp when the data was generated. Generate a timestamp as soon as possible to minimize the delay.

data_ch1 (MGPData)

Data from channel 1 (processed source).

data_ch2 (MGPData)

Data from channel 2 (automated source).

l1_id (int) [optional]

The ID of an L1 row from which the flow is inherited.

This class has custom implementation of deconstruction. If tried to deconstruct without finalization it throws an error.

MGPDataRowL2.finalize(result)

Finalizes the instance.

result (int or NoneType)

The result of consumption of the data. If it is stored, the result is an ID. If it is filtered, the result should be NoneType.

MGPDataRowL2.getid()

Gets the ID of the row.

Returns int or NoneType, if the ID of the row already exists, it returns it, else it returns None.

MGPDataRowL2.getl1id()

Gets L1 record ID

Returns int or NoneType, if there is an ID of an L1 row from which the flow is inherited, it returns it, else it returns None.

MGPDataRowL2.getlabel(channel)

Gets the label of the stored data.

channel (int)

The ID of the channel to get label from.

Returns string, the actual label.

MGPDataRowL2.getunit(channel)

Gets the measurement unit of the stored data.

channel (int)

The ID of the channel to get unit from.

Returns string, the name of the measurement unit.

MGPDataRowL2.getvalue(channel)

Gets the value, amount of the stored data.

channel (int)

The ID of the channel to get value from.

Returns string or int or float or bool, the value of the measurement or action.

MGPDataRowL2.is_filtered()

Gets whether the object is filtered or not.

Returns bool, True if the row is filtered, False if not.

MGPDataRowL3

class MGPDataRowL3(timestamp, data, l0_id=None, patient_id=None, patient_pwd=None)

This class holds a Third Level database record. It acts like a read only data container from one side and it is an actual storage handling tool from the other side. As a data container it has "getter functions" only and data must be set at the instantiation. As a data storage helper tool it can be finalized with getting the storage record id or it can be filtered.

timestamp (time alike)
The timestamp when the data was generated. Generate a timestamp as soon as possible to minimize the delay.
data (MGPData)
The actual data to store.
l2_id (int)
The ID of the L2 row from which the flow is inherited.
response (string)
The response resulted from L2 record.

This class has custom implementation of deconstruction. If tried to deconstruct without finalization it throws an error.

MGPDataRowL3.finalize(result)
Finalizes the instance.
result (int or NoneType)
The result of consumption of the data. If it is stored, the result is an ID. If it is filtered, the result should be NoneType.

MGPDataRowL3.getid()
Gets the ID of the row.

Returns int or NoneType, if the ID of the row already exists, it returns it, else it returns None.

MGPDataRowL3.getl2id()

Gets L2 record ID

Returns int, the ID of the L2 row from which the flow is inherited.

MGPDataRowL3.getresponse()

Gets the stored response.

Returns string, the content of the stored response.

MGPDataRowL3.is_filtered()

Gets whether the object is filtered or not.

Returns bool, True if the row is filtered, False if not.

MGPError

class MGPError()

This class provides unique error interface for Medical Gateway Platform. It is a subclass of Exception. It doesn't have any additional behavior than its parent.

MGPFilter

class MGPFilter(rule_function, filter_type)

This class is a wrapper class for custom filter functions. It is kind of a read-only wrapper since attributes cannot be set run-time.

rule_function (callable)

A function that is called to be used as a filter. However this class supports the freedom to accept positional and keyword arguments, it's highly recommended to accept ExitObject as first argument in the function.

filter_type (string)

The identifier of the filter.

MGPFilter.apply(*args, **kwargs)

Calls rule function and returns its result.

*args, **kwargs (object[s])

Parameters to be transferred to rule function.

Returns ExitObject, the result of filtering process.

MGPFilter.gettype()

Gets the given type identifier of the filter.

Returns string, the type identifier of the filter.

NullPipe

class NullPipe()

This class provides endpoint-less alternative to multiprocessing.PipeConnection class sending functionality. The goal of the class is to speed up the workflow of the areas since no if junctions are needed to filter non-running data flows.

NullPipe.send(arg)

Does nothing.

arg (object)

Data to ignore.

Area Classes

AreaOfGates

class AreaOfGates(flow_source, outputarea, ch1, ch2, gatelogics, loop_interval=100, data_pipe=None, ui_pipe=None)

This class provides the functionality of the Area Of Gates. The task of the class is to maintain a loop where processed and automation data is handled together to decide whether to output something to an actuator or not.

flow_source (multiprocessing.PipeConnection)
The pipe where source data come from.
outputarea (OutputArea)
The instance of Output Area where actuator functionality is implemented.
ch1 (object)
Handler object for processed data gating.
ch2 (object)
Handler object for automated gating.
gatelogics (dict)
Dictionary of gate logics. Key must be the gates to register and values must be the logic identifier of the gate.
loop_interval (int) [optional]
Time in milliseconds between cycles.
data_pipe (multiprocessing.PipeConnection) [optional]
Pipe to send data to the data flow.
ui_pipe (multiprocessing.PipeConnection) [optional]
Pipe to send data to the UI flow.

Handler functions of gating must always send output to each label that is handled by the Area of Gate. The class, as a parent of those handler objects copies all the

registered labels to .gates member of the handlers. The channel 1 handler must have an eval() function and the channel 2 handler must have a .tick() function. The output of both functions should be a tuple. The first element of the tuple should be a dictionary of AreaOfGates.GateObject alike objects with all the keys from the gates list.

Supported gating logic are followings:

P OR A (P)

Opens the gate if channel 1 (A) or channel 2 (P) or both open. If both would open the gate it sends the data content of channel 1 (P).

P OR A (A)

Opens the gate if channel 1 (A) or channel 2 (P) or both open. If both would open the gate it sends the data content of channel 2 (A).

P AND A (P)

Opens the gate if channel 1 (A) and channel 2 (P) both open. It sends the data content of channel 1 (P).

P AND A (A)

Opens the gate if channel 1 (A) and channel 2 (P) both open. It sends the data content of channel 2 (A).

P XOR A

Opens the gate if channel 1 (A) or channel 2 (P) would open but doesn't open if both would open.

P > A

Channel 1 (P) controls the gate, channel 2 (A) matters only if there is no data on channel 1 (P).

P < A

Channel 2 (A) controls the gate, channel 1 (P) matters only if there is no data on channel 2 (A).

P ONLY

Gate follows the behavior of channel 1 (P) only.

ONLY A

Gate follows the behavior of channel 2 (A) only.

AreaOfGates.loopinterval(new_value=None)

Gets or sets the value of loop interval.

new_value (int) [optional]

The new interval in milliseconds.

AreaOfGates.showimplemented()

Prints implemented get logic identifiers.

AreaOfGates.start()

Handles main loop.

AreaOfGates.__logic_and_a(ch1, ch2, ch1data, ch2data) [protected (hidden) function]

Performs logic P AND A (A).

ch1 (bool)

Whether channel 1 asks for opening or not.

ch2 (bool)

Whether channel 2 asks for opening or not.

ch1data (MGPData)

Data from channel 1.

ch2data (MGPData)

Data from channel 2.

AreaOfGates.__logic_and_p(ch1, ch2, ch1data, ch2data) [protected (hidden) function]

Performs logic P AND A (P).

ch1 (bool)

Whether channel 1 asks for opening or not.

ch2 (bool)

Whether channel 2 asks for opening or not.

ch1data (MGPData)

Data from channel 1.

ch2data (MGPData)

Data from channel 2.

AreaOfGates.__logic_only_a(ch1, ch2, ch1data, ch2data) [protected (hidden) function]

Performs logic ONLY A.

ch1 (bool)

Whether channel 1 asks for opening or not.

ch2 (bool)

Whether channel 2 asks for opening or not.

ch1data (MGPData)

Data from channel 1.

ch2data (MGPData)

Data from channel 2.

AreaOfGates.__logic_only_p(ch1, ch2, ch1data, ch2data) [protected (hidden) function]

Performs logic P ONLY.

ch1 (bool)

Whether channel 1 asks for opening or not.

ch2 (bool)

Whether channel 2 asks for opening or not.

ch1data (MGPData)

Data from channel 1.

ch2data (MGPData)

Data from channel 2.

AreaOfGates.__logic_or_a(ch1, ch2, ch1data, ch2data) [protected (hidden) function]

Performs logic P OR A (A).

ch1 (bool)

Whether channel 1 asks for opening or not.

ch2 (bool)

Whether channel 2 asks for opening or not.

ch1data (MGPData)

Data from channel 1.

ch2data (MGPData)

Data from channel 2.

AreaOfGates.__logic_or_p(ch1, ch2, ch1data, ch2data) [protected (hidden) function]

Performs logic P OR A (P).

ch1 (bool)

Whether channel 1 asks for opening or not.

ch2 (bool)

Whether channel 2 asks for opening or not.

ch1data (MGPData)

Data from channel 1.

ch2data (MGPData)

Data from channel 2.

AreaOfGates.__logic_p_g_a(ch1, ch2, ch1data, ch2data) [protected (hidden) function]

Performs logic P > A.

ch1 (bool)

Whether channel 1 asks for opening or not.

ch2 (bool)

Whether channel 2 asks for opening or not.

ch1data (MGPData)

Data from channel 1.

ch2data (MGPData)

Data from channel 2.

AreaOfGates.__logic_p_l_a(ch1, ch2, ch1data, ch2data) [protected (hidden) function]

Performs logic P < A.

ch1 (bool)

Whether channel 1 asks for opening or not.

ch2 (bool)

Whether channel 2 asks for opening or not.

ch1data (MGPData)

Data from channel 1.

ch2data (MGPData)

Data from channel 2.

AreaOfGates.__logic_xor(ch1, ch2, ch1data, ch2data) [protected (hidden) function]

Performs logic P XOR A.

ch1 (bool)

Whether channel 1 asks for opening or not.

ch2 (bool)

Whether channel 2 asks for opening or not.

ch1data (MGPData)

Data from channel 1.

ch2data (MGPData)

Data from channel 2.

AreaOfGates.GateObject

class AreaOfGates.GateObject(open, data)

This class stores the result of a single gate label process from channel 1 or channel 2. The use of this simple class is not necessary but it shows the minimal required setup of a class to be used for this purpose. No additional check is included since object of this class has minimum life-cycle, they get instantiated and deleted almost countless.

open (bool)
Whether to open the gate or not.
data (MGPData)
Data to be transferred through the gate.

DataArea

class DataArea(storage_type, configdict, data_source, loop_interval=5000, ui_pipe=None)

This class provides the whole functionality of the Data Area. The task of this class is to store and retrieve data with the help of the database manager engine selected by the user.

Supported storage types are:
DoF
CSV
SQL

storage_type (string)
Type of the selected storage method. Must be one of the implemented types.
configdict (dict)
Key value pair based settings according to the needs of the selected storing engine.
data_source (multiprocessing.PipeConnection)

The pipe where source data come from.

loop_interval (int) [optional]

Time in milliseconds between cycles.

ui_pipe (multiprocessing.PipeConnection) [optional]

Pipe to send data to the UI flow.

DataArea.engine (DataArea.__DataAreaEngine)

Member "engine" is class level variable. It provides the singleton-like functionality. However in Python DataArea.engine member could be called straight. To maintain a clean coding style it's strongly recommended to use DataArea's loop for data management.

DataArea.hasengine() (classmethod)

Gets whether DataArea has an engine or not.

Returns bool, True if engine is instantiated, False else.

DataArea.showimplemented()

Prints implemented storage identifiers.

DataArea.start()

Handles main loop.

DataArea.__DataAreaEngine

class DataArea.__DataAreaEngine(engine)

This class provides a wrapper for a DataArea.__EngineCSV, DataArea.__EngineDoF or DataArea.__EngineSQL engine, which serves the actual data storage handling. The use of a wrapper has advantages in simplification of type check and involves a future possibility to improve the data management process workflow. The workflow itself consists of basic data storage functionality based on CRUD model.

engine (__EngineCSV or __EngineDoF or __EngineSQL)

The instance of engine to work with.

DataArea.__DataAreaEngine.create(level_id, record)

Wraps the engine's create() function.

level_id (int)

The identifier of the storage level.

record (MGPDataRowL0 or MGPDataRowL1 or MGPDataRowL2 or MGPDataRowL3)

The data to store.

Returns int, the ID of the record or -1 in case of failure.

DataArea.__DataAreaEngine.delete(level_id, record_id)

Wraps the engine's delete() function.

level_id (int)

The identifier of the storage level.

record_id (int)

The ID of the record.

Returns bool, True if succeed, False if failed.

DataArea.__DataAreaEngine.read(level_id, record_id)

Wraps the engine's read() function.

level_id (int)

The identifier of the storage level.

record_id (int)

The ID of the record.

Returns MGPDataRowL0 or MGPDataRowL1 or MGPDataRowL2 or MGPDataRowL3 or bool, data if succeed, False if failed.

DataArea.__DataAreaEngine.update(level_id, record_id, record)

Wraps the engine's update() function.

level_id (int)

The identifier of the storage level.

record_id (int)

The ID of the record.

record (MGPDataRowL0 or MGPDataRowL1 or MGPDataRowL2 or MGPDataRowL3)

The data to update.

Returns bool, True if succeed, False if failed.

DataArea.__EngineCSV

class DataArea.__EngineCSV(configdict)

This class provides CSV endpoint to the data management workflow. The engine realizes basic data storage functionality based on CRUD model.

configdict (dict)

Settings to instantiate engine.

DataArea.__EngineCSV.create(level_id, record)

Creates a new record in the database.

level_id (int)

The identifier of the storage level.

record (MGPDataRowL0 or MGPDataRowL1 or MGPDataRowL2 or MGPDataRowL3)

The data to store.

Returns int, the ID of the record or -1 in case of failure.

DataArea.__EngineCSV.delete(level_id, record_id)

Deletes a record from the database.

level_id (int)

The identifier of the storage level.

record_id (int)

The ID of the record.

Returns bool, True if succeed, False if failed.

DataArea.__EngineCSV.read(level_id, record_id)

Reads a record from the database.

level_id (int)

The identifier of the storage level.

record_id (int)

The ID of the record.

Returns MGPDataRowL0 or MGPDataRowL1 or MGPDataRowL2 or MGPDataRowL3 or bool, data if succeed, False if failed.

DataArea.__EngineCSV.update(level_id, record_id, record)

Updates a record in the database.

level_id (int)

The identifier of the storage level.

record_id (int)

The ID of the record.

record (MGPDataRowL0 or MGPDataRowL1 or MGPDataRowL2 or MGPDataRowL3)

The data to update.

Returns bool, True if succeed, False if failed.

DataArea.__EngineDoF

class DataArea.__EngineDoF(configdict)

This class provides DoF endpoint to the data management workflow. The engine realizes basic data storage functionality based on CRUD model.

configdict (dict)

Settings to instantiate engine.

DataArea.__ EngineDoF.create(level_id, record)

Creates a new record in the database.

level_id (int)

The identifier of the storage level.

record (MGPDataRowL0 or MGPDataRowL1 or MGPDataRowL2 or MGPDataRowL3)

The data to store.

Returns int, the ID of the record or -1 in case of failure.

DataArea.__ EngineDoF.delete(level_id, record_id)

Deletes a record from the database.

level_id (int)

The identifier of the storage level.

record_id (int)

The ID of the record.

Returns bool, True if succeed, False if failed.

DataArea.__ EngineDoF.read(level_id, record_id)

Reads a record from the database.

level_id (int)

The identifier of the storage level.

record_id (int)

The ID of the record.

Returns MGPDataRowL0 or MGPDataRowL1 or MGPDataRowL2 or MGPDataRowL3 or bool, data if succeed, False if failed.

DataArea.__EngineDoF.update(level_id, record_id, record)

Updates a record in the database.

level_id (int)

The identifier of the storage level.

record_id (int)

The ID of the record.

record (MGPDataRowL0 or MGPDataRowL1 or MGPDataRowL2 or MGPDataRowL3)

The data to update.

Returns bool, True if succeed, False if failed.

DataArea.__EngineSQL

class DataArea.__EngineSQL(configdict)

This class provides SQL endpoint to the data management workflow. The engine realizes basic data storage functionality based on CRUD model.

configdict (dict)
Settings to instantiate engine.

DataArea.__ EngineSQL.create(level_id, record)
Creates a new record in the database.
level_id (int)
The identifier of the storage level.
record (MGPDataRowL0 or MGPDataRowL1 or MGPDataRowL2 or MGPDataRowL3)
The data to store.
Returns int, the ID of the record or -1 in case of failure.

DataArea.__ EngineSQL.delete(level_id, record_id)
Deletes a record from the database.
level_id (int)
The identifier of the storage level.
record_id (int)
The ID of the record.
Returns bool, True if succeed, False if failed.

DataArea.__ EngineSQL.read(level_id, record_id)
Reads a record from the database.
level_id (int)
The identifier of the storage level.
record_id (int)
The ID of the record.

Returns MGPDataRowL0 or MGPDataRowL1 or MGPDataRowL2 or MGPDataRowL3 or bool, data if succeed, False if failed.

DataArea.__EngineSQL.update(level_id, record_id, record)

Updates a record in the database.

level_id (int)

The identifier of the storage level.

record_id (int)

The ID of the record.

record (MGPDataRowL0 or MGPDataRowL1 or MGPDataRowL2 or MGPDataRowL3)

The data to update.

Returns bool, True if succeed, False if failed.

InputArea

class InputArea(flow_pipe=None, data_pipe=None, ui_pipe=None)

This class provides the functionality of the Input Area. The task of the class is to translate information received from sensors and state variables to data for the Processing Area.

flow_pipe (multiprocessing.PipeConnection) [optional]

Pipe to send data to the main flow.

data_pipe (multiprocessing.PipeConnection) [optional]

Pipe to send data to the data flow.

ui_pipe (multiprocessing.PipeConnection) [optional]

Pipe to send data to the UI flow.

InputArea.is_registered(name)

Checks whether a handler is registered or not.

name (string)

The name of the handler to check.

Returns bool, True if name is registered, False if not.

InputArea.read(handler_name, rawdata)

Translates raw data to processable data.

handler_name (string)

The identifier of the handler.

rawdata (object)

The data to be sent to the handler.

InputArea.register(handler_name, handler_type, handler_function, filter_function=None)

Registers a handler to an identifier.

handler_name (string)

Identifier to register.

handler_type (string)

Type of the handler.

handler_function (callable)

The handler function.

filter_function (callable) [optional]

The filter function.

InputArea.unregister(handler_name)

Unregisters a handler.

handler_name (string)

The identifier to delete.

InputArea.InputHandler

class InputArea.InputHandler(handler_type, handler_function, filter_function)

This class wraps handler chain for Input Area processes. The chain consists of a handler and a filter function.

handler_type (string)
Type of the handler.
handler_function (callable)
The handler function.
filter_function (callable or NoneType)
The filter function or None if no filter should be applied.

InputArea.InputHandler.filter(exitdata)
Applies filter on the data.
exitdata (ExitObject)
Data to filter.
Returns ExitObject, the filtered data.

InputArea.InputHandler.gettype()
Gets the type of the handler.
Returns string, the type of the handler.

InputArea.InputHandler.handle(rawdata)
Translates the raw data and applies filter.
rawdata (object)
Data to translate and filter.
Returns ExitObject, the translated and filtered data.

InputArea.InputHandler.justhandle(rawdata)
Translates the raw data without filter.
rawdata (object)

Data to translate.

Returns ExitObject, the translated data.

OutputArea

class OutputArea(data_pipe=None, ui_pipe=None)

This class provides the functionality of the Output Area. The task of the class is to translate information received from the Area of Gate to commands that the actuators understand.

data_pipe (multiprocessing.PipeConnection) [optional]

Pipe to send data to the data flow.

ui_pipe (multiprocessing.PipeConnection) [optional]

Pipe to send data to the UI flow.

OutputArea.act(handler_name, data)

Translates gate action to data for actuator.

handler_name (string)

The identifier of the handler.

data (MGPData)

The data to be sent to the handler.

OutputArea.is_registered(name)

Checks whether a handler is registered or not.

name (string)

The name of the handler to check.

Returns bool, True if name is registered, False if not.

OutputArea.register(handler_name, handler_type, handler_function, filter_function=None)

Registers a handler to an identifier.

handler_name (string)

Identifier to register.

handler_type (string)

Type of the handler.

handler_function (callable)

The handler function.

filter_function (callable) [optional]

The filter function.

OutputArea.unregister(handler_name)

Unregisters a handler.

handler_name (string)

The identifier to delete.

OutputArea.OutputHandler

class OutputArea.OutputHandler(handler_type, handler_function, filter_function)

This class wraps handler chain for Output Area processes. The chain consists of a handler and a filter function.

handler_type (string)

Type of the handler.

handler_function (callable)

The handler function.

filter_function (callable or NoneType)

The filter function or None if no filter should be applied.

OutputArea.OutputHandler.filter(exitdata)

Applies filter on the data.

exitdata (ExitObject)

Data to filter.

Returns ExitObject, the filtered data.

OutputArea.OutputHandler.gettype()

Gets the type of the handler.

Returns string, the type of the handler.

OutputArea.OutputHandler.handle(data)

Translates the data and applies filter.

data (ExitObject)

Data to translate and filter.

Returns ExitObject, the translated and filtered data.

OutputArea.OutputHandler.justhandle(data)

Translates the data without filter.

data (ExitObject)

Data to translate.

Returns ExitObject, the translated data.

ProcessArea

class ProcessArea(flow_source, processor_function, loop_interval=100, flow_pipe=None, data_pipe=None, ui_pipe=None)

This class provides the functionality of the Processing Area. The task of the class is to maintain a loop where input data gets processed and sent to Area of Gates and other to other flows as well.

flow_source (multiprocessing.PipeConnection)

The pipe where source data come from.

processor_function (callable)

Callable function to process input data. Every data gets processed with the same function.

loop_interval (int) [optional]

Time in milliseconds between cycles.

flow_pipe (multiprocessing.PipeConnection) [optional]

Pipe to send data to the main flow.

data_pipe (multiprocessing.PipeConnection) [optional]

Pipe to send data to the data flow.

ui_pipe (multiprocessing.PipeConnection) [optional]

Pipe to send data to the UI flow.

ProcessArea.loopinterval(new_value=None)

Gets or sets the value of loop interval.

new_value (int) [optional]

The new interval in millisendos.

ProcessArea.start()

Handles main loop.

Unique and special solutions in MGP

Defensive coding

Python is not the best programming language to implement defensive coding style. However in most use cases the need to do this is limited, when we develop a solution for people with even light coding skills, the use of defensive style can be a good prevention against mistakes. Moreover since there are medical equipment.

Singleton

The DataArea class of the Medical Gateway Platform gets instantiated as a singleton. This behavior is not at all common in Python since the ability of creating static object is not at all supported in this programming language. The most common way of making a singleton is to use a protected class from inside its parent. This is the situation in this case too.

Protected member variables, functions and classes

Classes in MGP are widely using protected member variables, functions and classes to operate with. Protected state means the content and the ability of overriding or calling is hidden from outside even from children or subclasses. In programming languages this kind of an architecture is quite common but in Python this ain't at all usual. The wide use of protected behavior aims to demonstrate the robust architecture behind the scene and targets to prevent developers with lighter coding skills to do something that endangers the clear and safe workflow of an equipment.

Cycle time versus instantiation time

Saving time means saving resources in coding. In a software like MGP there are two main source of saving time. One concept is letting things just going at instantiation and another solution is trusting the type and functionality of variables in loops. Fast instance creation is comfortable at start because things gets started faster while trusting instances behavior results less response time and lower resource consumption during the whole lifetime. In real-world use-case both concepts has their own reason of existence. In MGP both ways are used. Loops trust their variables since they are mostly protected and type-checked at the instantiation. On the other hand instances with short lifetime gets instantiated without further checks. They born and die in automated inner cycles so the lack of additional control doesn't risk the stability of the workflow.

Demangling

Demangling means type checking and using without importing the type itself. It is not that common in coding since single-process applications doesn't need them even in case when they are using multiple threads. Software with multiple processes are different because the separated processes are independent representations of code. Therefore it matters a lot whether or not to import or include something into a process. The implementation of demangling helps reducing the resource usage of the software – especially in runtime memory. The code asks for the type identifier of an instance instead of directly checking it from in included prototype. Moreover instances used to bring their functionality along so calling members can achieved without inclusion as well.

Piped communication

The communication between separated processes seem to be ceremonious from a distant view. In practice there are several ways to solve it. In the development of MGP we decided to implement communication between processes with the help of datapipes. Those pipes are volatile variables which have a sender and a receiver side. Pipes are straight connected between workflow areas the leak of data is impossible. In the near future dispatcher alike or API based communication can be also implemented.

Junction optimized loops

Yet nowadays runtime junctions are considered time consuming solutions. This opinion is true even if we are talking about very small amount of time. Since we aim to maintain medical equipment each nanosecond counts. Instead of junctions we are using data streams that lead to nowhere and some other tricks to not to loose time needlessly. The implementation of this approach differs in programming languages.

Customizable and regulated loop intervals

Let a thread sleeping is one of the most efficient way of saving resources but on the other hand it means lack of control or increase of response time. Some processes afford having long response time while some other processes are not allowed to do this. It depends on use-case and hardware too. MGP has a design which let developer deciding about loop frequency in every area. Moreover the frequency of a loop is basically corrected in nanosecond level. In real life the accuracy of this depends on hardware and programming language too.

Advantages of our idea

First and foremost our main advantage is that we provide a basically open solution which highly accords to the way of human thinking. For a developer it is easy to use just like drag and drop coding. With our connectors and class concepts existing code fast can fit to a new. This is important if parts should be replaced of a device or in device conception. The shortage of parts is not that big issue anymore. The development time decreases because we provide fast testing with built-in testing mechanisms and lint-alike error checking flow.

When a machine gets manufactured and Medical Gateway Platform begins to work in daily practice its new advantages get more important role. A new device can be started with with a couple of clicks. Both the equipment and the patient can receive unique ID so with MGP every ventilator – open source or industrial manufactured – can be part of the same remote maintenance and observation system in the hospital. We provide clean and highly customizable user interface where every data of a patient are shown. Devices configuration and attributes can get changed quickly even remotely. Our software has built-in warning and alarm mechanism to be able to work in area where skilled crew are not nearby.

On the direct UI of the device only important data are visible just like at industrial manufactured machines. This behavior can be changed to reach more complex data easily like stored data to check longer period of ventilation history. Patients and doctors have full control on sensitive health data. Our software doesn't need central servers to operate, the data leak can be fully excluded.

Our solution is for medical equipment like Google Auto and Apple CarPlay for vehicles and drivers. We try to invent modern solutions and aspects in a traditional and sensible field of life and industry.

Future plans

There are 3 most important steps in front of MGP. First one is get used by as much open source ventilator projects as possible. This way the effort of devoted people can be used with more efficiency in the near future. After MGP proved its strength it's time to make the next step forward.

Second step is to reach the use of MGP based solutions in the medical industry too. Open source projects are not real competitors for traditional companies of the healthcare industry but the experience with low cost hardware and the know-how of home made design can improve their manufacturing too. The situation is just the same with the software. Differences in the field of performance or quality of hardware parts happen in normal processes too when manufacturers should change their suppliers. MGP has a clearly separated workflow, and a very flexible structure. Behavior of the software can be changed even runtime. Therefore we are thinking in a changing world which we might face in the near future our solution is a good platform for healthcare industry to build ventilators.

If we proved the strength of Medical Gateway Platform in maintaining and remote controlling ventilators we are eager to step even forward and use MGP in case of other medical devices as well to build a strongly unified medical platform for a highly data driven use.

We are thinking with open source code companies can build cheap solutions quickly. This ability can help the companies in Europe to have greater share in the world scale medical industry. We aim to utilize those facts and circumstances to promote our solution for everyone.

About us

Axel Ország-Krisz Dr.

https://hyperrixel.com/en/axel

Richárd Ádám Vécsey Dr.

https://hyperrixel.com/en/richard

We are two deep learning developers and data scientists. We work a lot of different field of AI developing, but our main objective is a computer vision in healthcare. Nowadays, most companies like to hire developers who are outsiders, who have other skills and education. We are lawyers who have jumped from law to programming and deep learning. We had a lo of opportunity from Facebook and Udacity. These companies maintain scholarship programs to get knowledge people around the world. We were two of them. Computer Vision Nanodegree or Deep Learning Nanodegree were supported by Facebook. Besides the knowledge, we got a lot of new viewpoint and the possibility to help others. We are thankful and during the COVID-19 epidemic, we are trying to get back something to the community and to the people around the Globe. Under these sign, we created the following projects.

👉 cheXrad

CheXrad is pulmonary application with 4 different artificial intelligence models. It can be trained very fast with CoCoLo technology. The software's goal is to support doctors' decisions with analyzing x-ray images.

Each artificial intelligence network is trained separately so their knowledge can be specialized or generalized as well. The neural network analyze the content of an image independently from each other too. The result can be a simple decision about the X-ray picture: healthy – not healthy. In case of illness the concrete disease is also communicated.

The four separated neural networks work like healthcare counselor group of several specialists which only purpose is to support the doctor. It gives not only a second opinion but also third and fourth opinion of the actual case. The result of each network will be evaluated and visualized after the decision process. Multi-level hierarchy is the key so the computer could decide even if it did not see the actual problem during the training process.

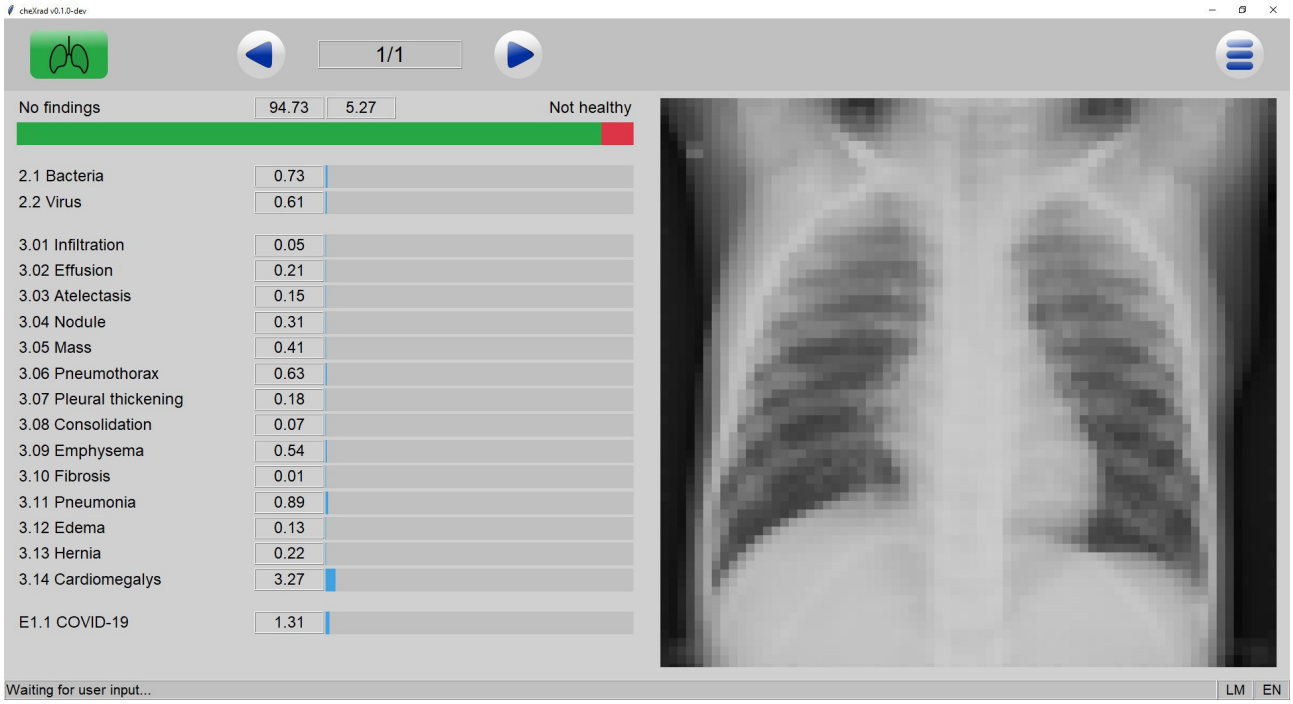
Repository is available here:

https://github.com/hyperrixel/cheXrad

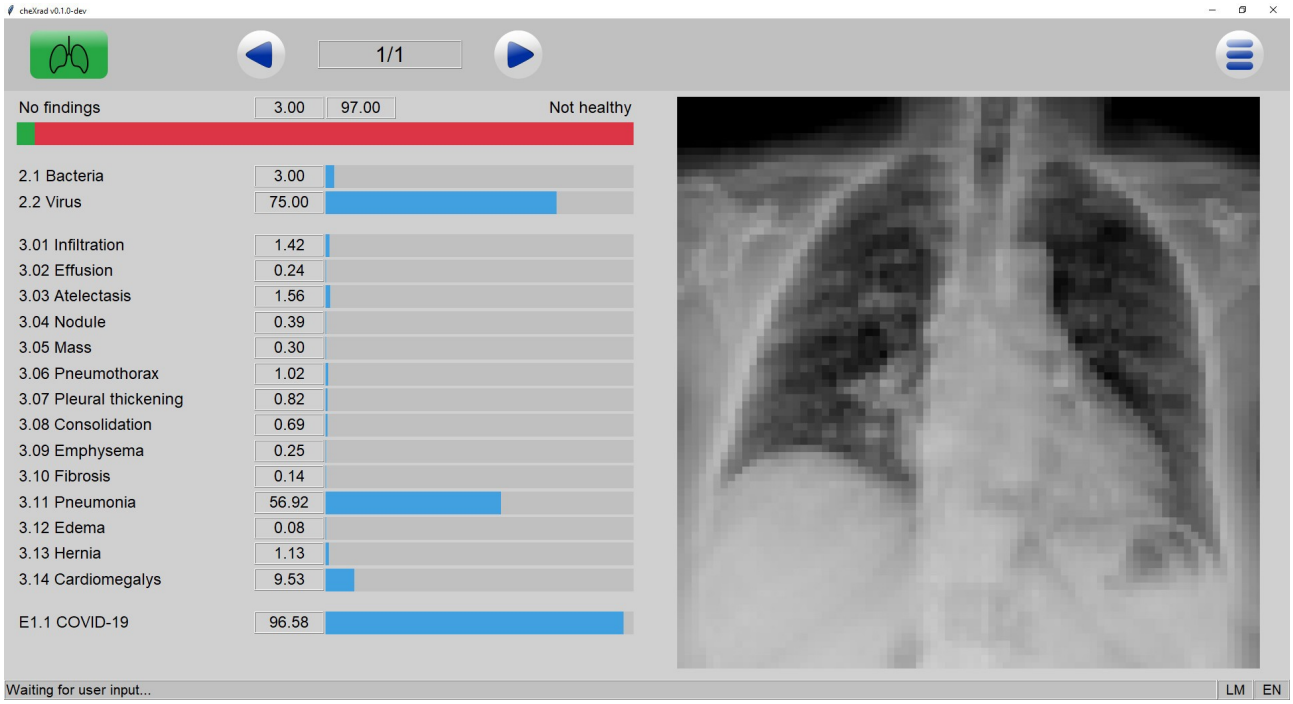
Screenshots:

X-ray images on screenshots are pixelized due to regulation about usage of health data. Images on "not healthy" pictures is made from a patient in COVID-19 disease according to https://github.com/ieee8023/covid-chestxray-dataset GitHub repository. There is two available visual format, light and dark theme.

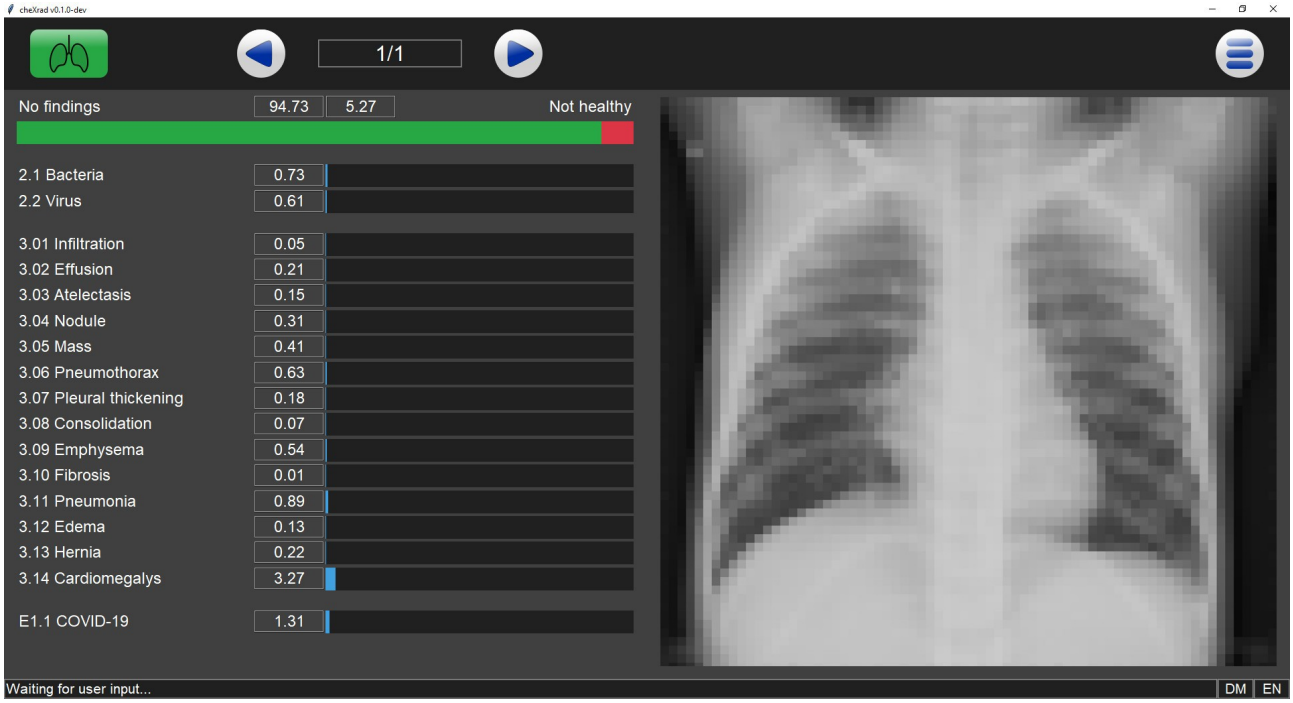
No findings in light theme



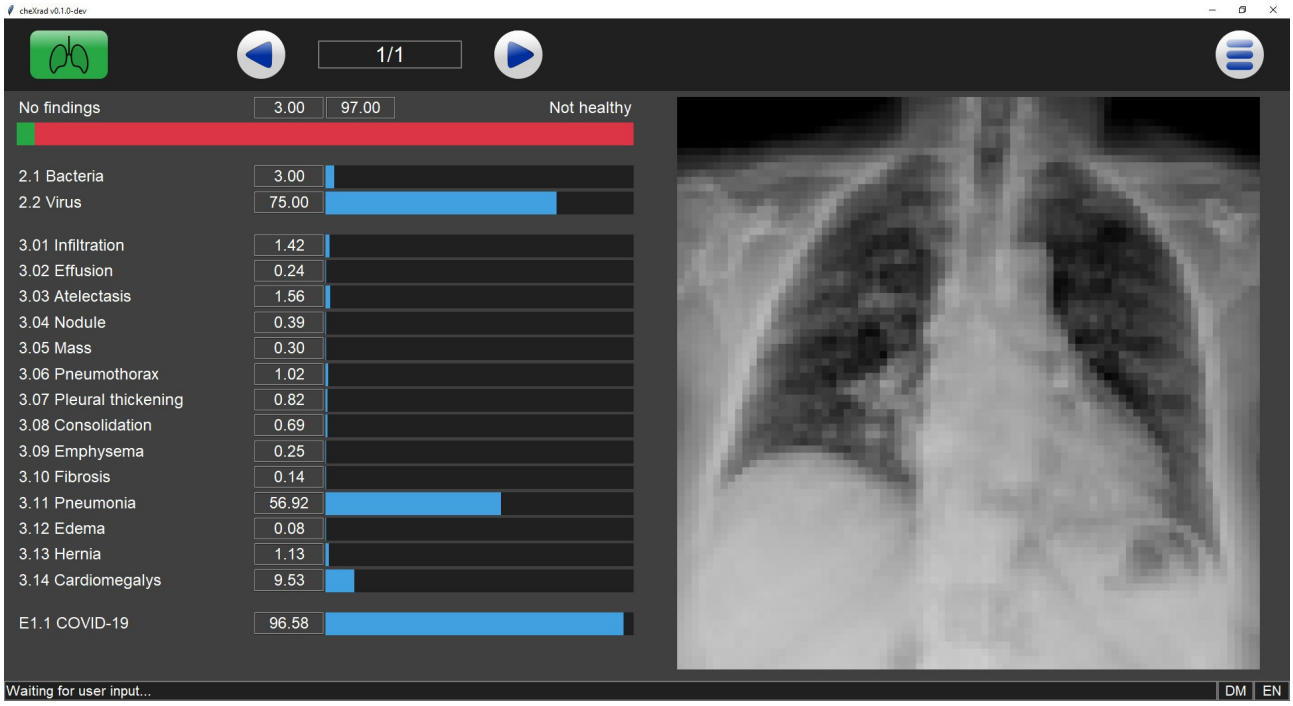
Not healthy in light theme



No findings in dark theme



Not healthy in dark theme



👉 DOF – Deep Model Core Output Framework

DoF is a highly scalable dataset format which helps deep learning scientists to work with foreign or sensitive data. DoF provides fast dataset sharing and data-secure at the same time.

It is fast because during training process it's enough to run forward the dataset through the pretrained core models only once. It's not necessary to calculate these values again at the next epoch. It saves computational time. These values will be always the same because the core part of the model is no more trained.

It is secure since headless deep learning networks anonymize the unique element of the original dataset. The original data cannot be restored from the saved result, layers make reversibility proof destructions.

The DoF framework is not just a container file. It is away of thinking and sharing datasets to help each others work.

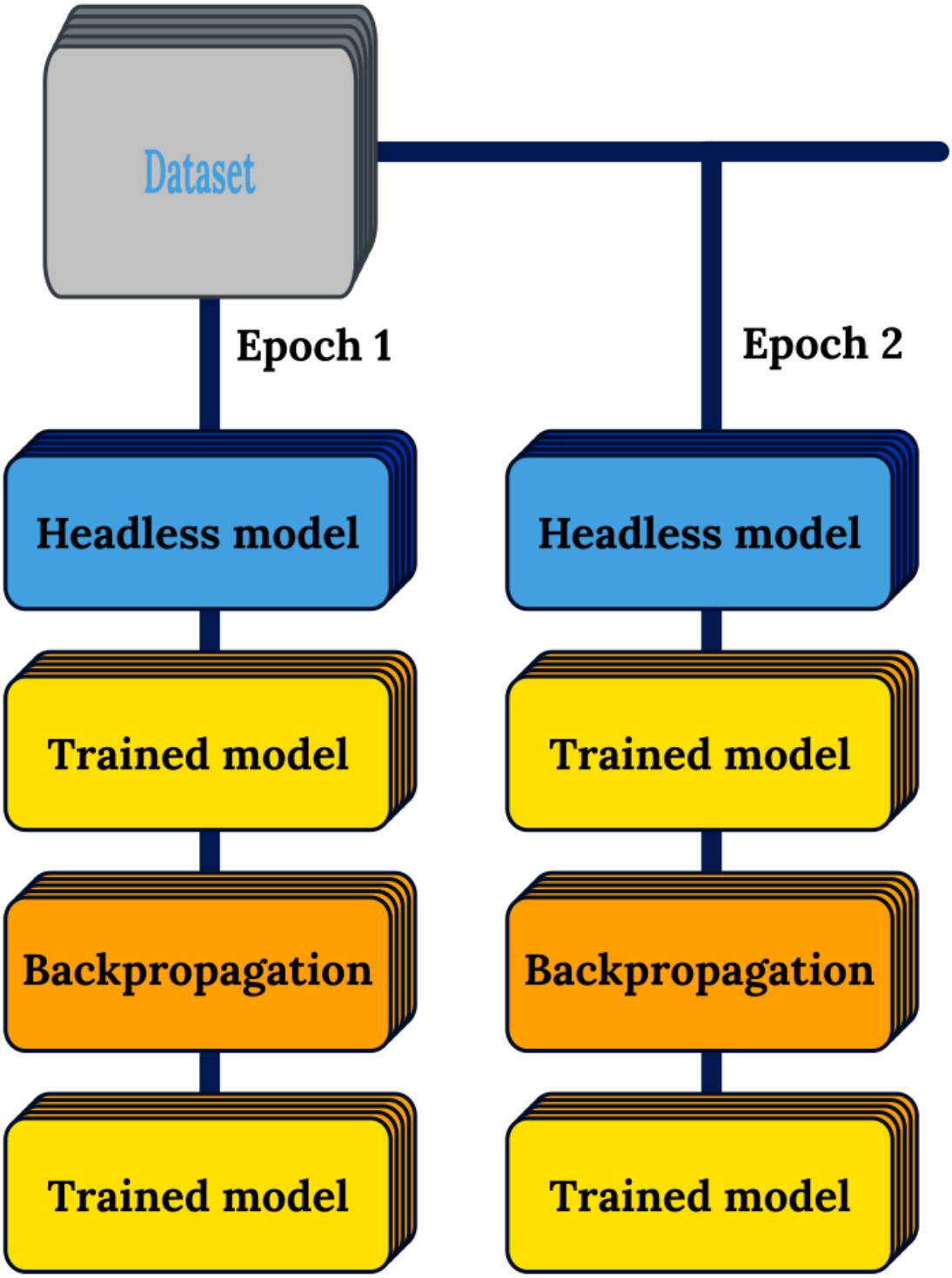
The 4 leverage of DoF:

1. faster deep learning training process
2. consume less storage and bandwidth capacity
3. possibility of addition notes
4. GDPR-, regulation- and legal friendly data-handling

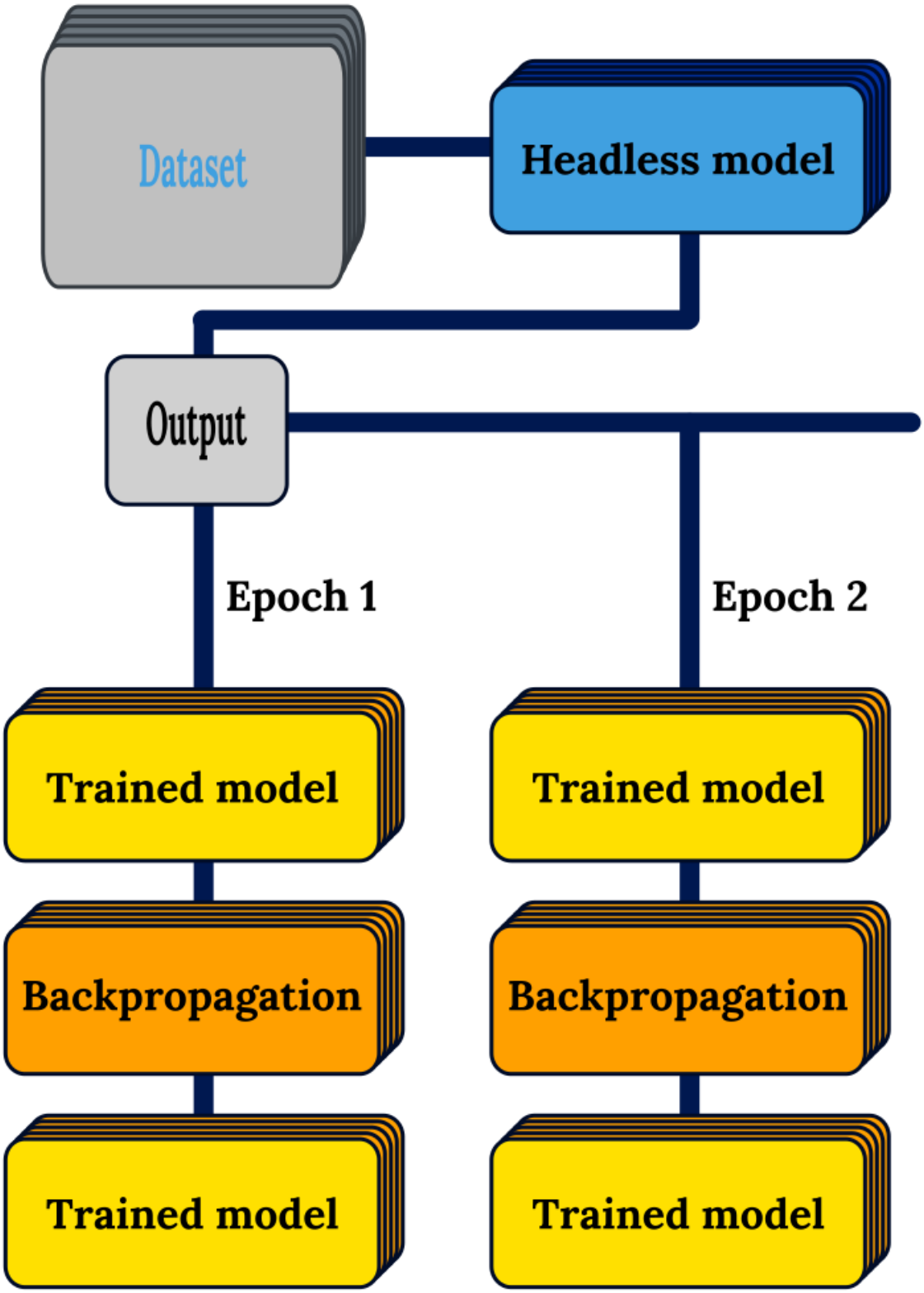
Repository is available here:

https://github.com/hyperrixel/dof

normal train process



DoF train process



👉 COVID-19 dataset

We download the daily dataset from ECDC (European Centre for Disease Prevention and Control) and make a standardized csv from this dataset. It contains the additional and cumulated confirmed cases and death number per days and countries. The standardized structure supports the fast processing. It can be filtered fast and easily. Every common programs and modules for datascientists or journalists be able to handle csv format from pandas in python through R till Excel.

Dataset contains 6 columns in the following order:
date | areacode | cumulated cases | cumulated death | additional cases | additional death

Example:

date	areacode	cumulated cases	cumulated death	additional cases	additional death
2020-03-29	20	536	30	41	6
2020-03-29	27	1187	2	17	0
2020-03-29	30	1061	32	95	4
2020-03-29	31	9762	639	1159	93
2020-03-29	32	9134	353	1850	64
2020-03-29	33	37575	2314	4611	319
2020-03-29	34	72248	5690	8189	832

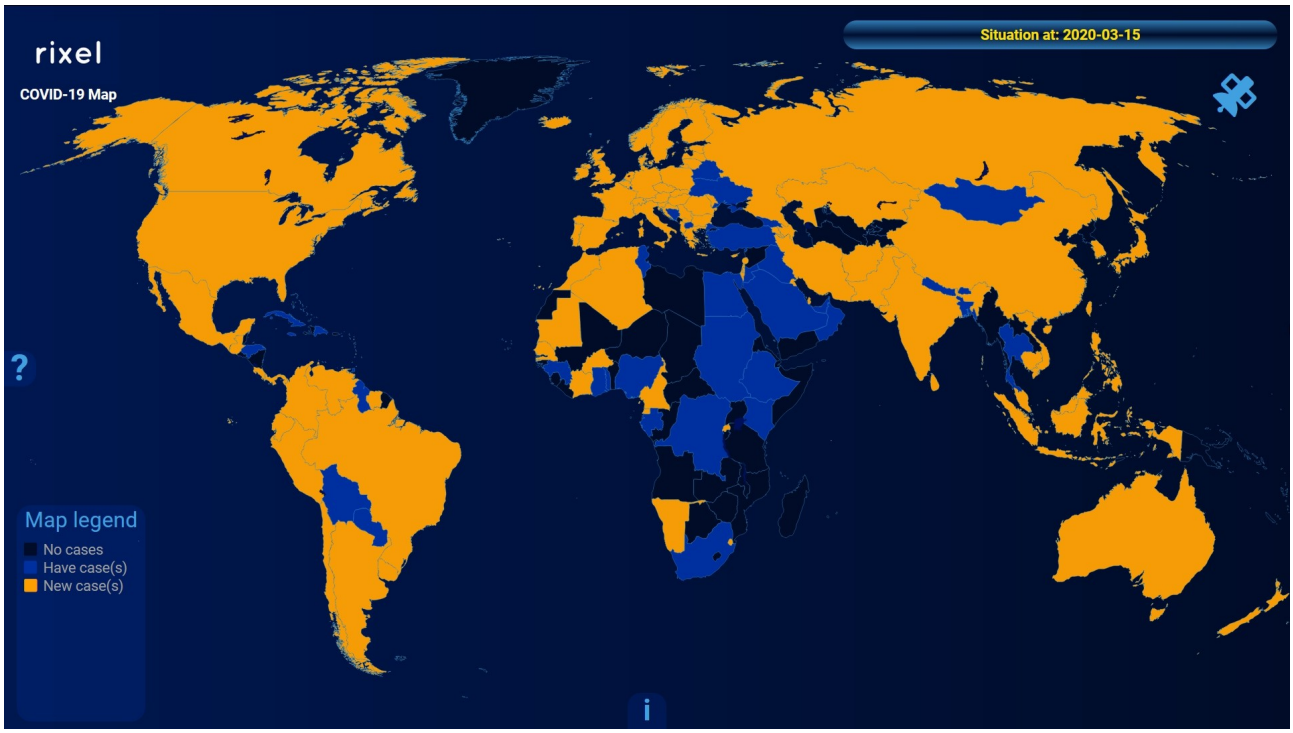
Repository is available here:

https://github.com/hyperrixel/covid19map-dataset

COVID-19 map

Besides the daily dataset-making routine, we wanted to create a simple visualization tool that helps people to understands the raw numbers and add visualization ideas to other datascientists. There are a lot of techniques to process an epidemic dataset, we show an example one of them.

Screenshot:



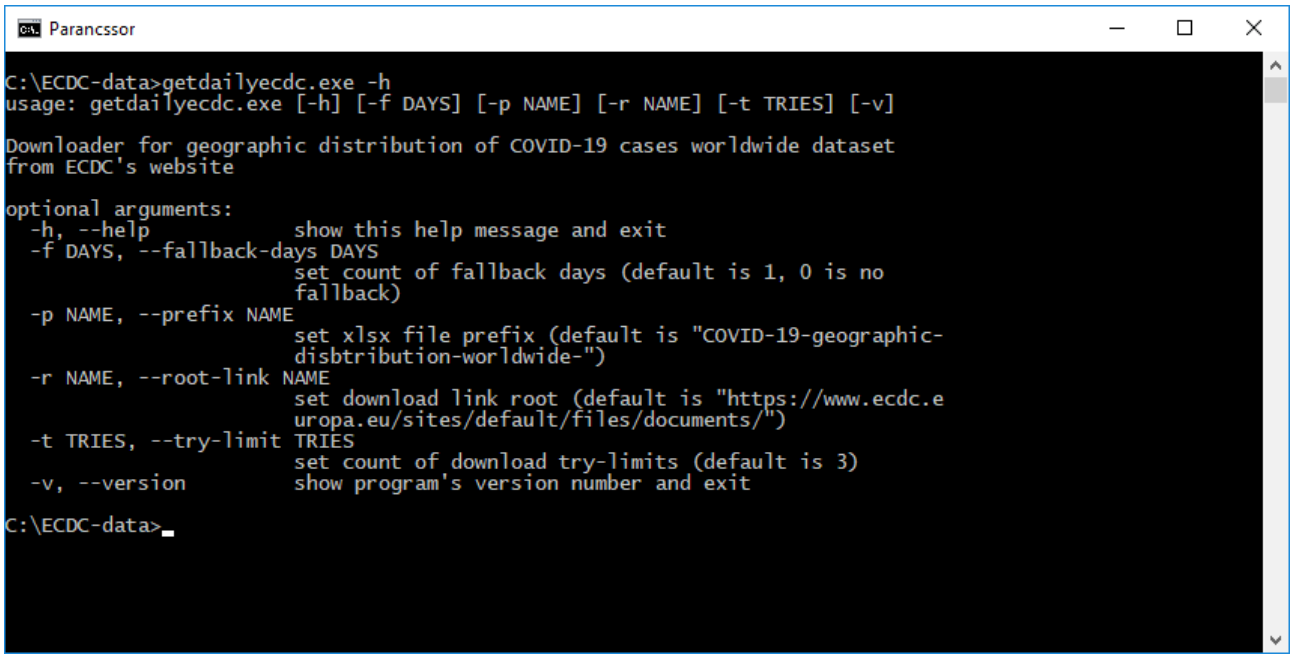
Visualization is available here:

https://www.hyperrixel.com/covid19map/

👉 COVID19CasesWorldwideDatasetFromECDC

The goal of this project is to provide downloader and loader for geographic distribution of COVID-19 cases worldwide dataset from ECDC's website. This project includes multiple files. These can be used as module or as script based on the user's need. As a module ecdcloader.py downloads the latest dataset and loads it into pandas dataframe. As a script it just downloads the latest dataset. It is possible to finetune the parameters with argparser. This method works with the executable file too. For researchers who are not familiar with python, we provide an executable version of script.

Screenshot:



Repository is available here:

https://github.com/hyperrixel/COVID19CasesWorldwideDatasetFromECDC