Blockchain Hackathon 2021

Medi System Decentralised Application

Project Documentation

Anna Qiu, Maximilian Glimm, Felix Wang

Supervisor: Niclas Kannengießer, Mikael Beyene
Karlsruher Institute of Technology, AIFB

15. October 2021

Table of Contents

1. About Medi Systems

Medi Systems is a decentralised application, which uses the Ethereum blockchain. It was developed as part of the university course “Praktikum Blockchain Hackathon 2021”. The course was carried out in cooperation of AIFB institute of KIT and medicalvalues.

2. Concept

In this section is about our thoughts and concepts on/for Medi Systems. We will write about problems we encountered and our solutions to them. This section will not detail the in depth implementation, but give an overview of our design thoughts. For detailed explanations of the code and implementation, see sections 3 and 4.

2.1 Problems and Objectives

The medicalvalues knowledge graph is being trained by machine learning algorithms. In order to improve the product, big amounts of data are needed. These data are being contributed by several different parties (users), such as doctors, research facilities or hospitals. In the case of doctors and hospitals, strict regulations must be adhered to. One challenge is, that patients data cannot leave the hospital or the doctors office. The knowledge graph is trained locally on servers inside the medical facility, so that the data never leave.

As contributions of data play such a big role in the system, it is critical that users not only use the medicalvalues workbench, but also contribute datasets to improve and develop the knowledge graph, that powers it.

Medi Systems is designed to encourage users to contribute datasets. This is achieved by two main objectives. The first being a **reward system** and the second being **fairness and transparency**.

MediSystems rewards contributions with MediCoins. In the future, these MediCoins can be traded for discounts on the medicalvalues workbench. This means, that users, who contribute, pay less for the product. To ensure fairness and transparency, the user needs to be assured, that every user is treated the same way, in the sense, that no one is being favoured by getting more MediCoins for datasets of similar quality. This is a great use-case for the blockchain as the ledger cannot be manipulated easily. If a transaction happens, all participating nodes will be informed. Everyone can research the transaction-history and comprehend what has happened. A critical function is the one, that calculates the value of a dataset. Having the function in the blockchain makes it accessible to the user. This is an advantage compared to a traditional server, where a change in the source code would be hidden from the user.

2.2 Dataset Evaluation Algorithm

2.2.1 Ensuring Fairness and Transparency

The task of the dataset evaluation algorithm is to calculate the value of a given dataset. It gets called after the user has selected the dataset in the frontend application, that he wants to share. The ideal solution to ensure fairness and transparency is to let the user send his dataset directly to the blockchain as a csv file. This way, the whole algorithm that parses and extracts the information from the file would be visible and accessible to all users of the system. A local doctor can be assured that users, like renowned research facilities or big hospitals are not favoured by getting more MediCoins for a dataset of same quality. This helps building trust in the system and help encourage users to contribute data.

The problem is, that the solidity programming language does not seem to support csv parsing yet. Working on a big file would also lead to extremely high execution costs, making the product not viable. To solve this problem, the data must be pre-processed, before they are passed to the function in the smart contract, while also keeping things transparent and comprehensible for the users.

The solution we decided on works as follows: The main evaluation algorithm stays in the smart contract, as anything different would make using the blockchain obsolete and an an alternative approach to reach transparency and fairness would be needed. But instead of taking a whole csv file as a parameter, the dataset evaluation function expects the following.

Parameters	Type
age	string[]
gender	string[]
snomed	string[]
loinc	bool
radlex	bool
numberOfAttributes	uint256
numberOfPatients	uint256
_fileHash	bytes32
disease	string

Explanations for each of them can be found in the description of the `calculateDatasetValue` function in the MediSystem contract. Here, we will take the parameter age as an example to demonstrate how the algorithm overcomes the challenge of not being able to receive the whole csv as an argument.

age is a string array. The function expects the first index, `age[0]`, to contain a string representation of the lowest occurring age in the dataset, the second index `age[1]` to contain the highest occurring age. The third index, `age[2]` is expected to contain a string representation of the number of entries in the dataset, that are not numbers. Using these information, the function can check for implausible data in a given dataset and evaluate the dataset accordingly.
The main concern, that emerged from this solution was, that this diminishes the purpose of the function being in the blockchain, as the client gets to pre-process the dataset, before the critical information get passed to the smart contract. It should be made clear, that the evaluation algorithm itself is still in the smart contract, deployed on the blockchain. The client pre-processing only serves the purpose to extract the needed data for the main evaluation algorithm. A user of the system can read the source code of the smart contract and calculate the resulting value by extracting the information from the dataset himself, only to come to the conclusion, that calling the function via the frontend leads to the same result. As the frontend source code is visible in the browser anyways, the user can be sure, that the frontend code does not degrade and manipulate the value of the dataset by passing information to the smart contract, that suggest the dataset to be of a inferior quality than it actually is. The user can be sure, that he is not deceived. The only remaining problem is, that the user cannot be sure, that other users may be served a different, more desirable version of the frontend code, that manipulates their datasets in a good way, by passing arguments to the evaluation algorithm, that suggest their datasets to be of a better quality than they actually are. But this problem also exists, in our ideal situation, where the frontend passes the whole csv file to the smart contract, as a user can only comprehend, that he is not being deceived, but not, that others may be favoured. Even though it would not make much sense in the standpoint of medicalvalues, the frontend code could manipulate the entire csv file of certain “vip” users in an advantageous way for them, before the file is being passed to the algorithm.
We talked about this problem with medicalvalues, and they suggested, that this problem could be solved by introducing a third party supervisor. As the machine learning algorithms are run locally at the users locations, medicalvalues suggested, that such a supervisor could be introduced to occasionally check and oversee the systems on site. This supervisor could also assure, that every user is being served the same frontend code and that it is not being manipulated, by medicalvalues or the user.

2.2.2 Calculating the dataset-value

The output of the algorithm is a concrete amount of MediCoins, that the user will get for his dataset. To calculate this concrete number, the algorithm starts by calculating a score for each of the quality attributes. The score, scaling from 0 to 100, indicates the quality of the data about the specific attribute. In total, we identified seven quality attributes. These are: information about age of the patients, information about the gender of the patients, the number of attributes in the datasets (columns), the number of patients, that the dataset contains data about, the existence of loinc data, the existence of radlex data and information about snomed. Because information about the number of attributes in the datasets, the number of patients, that the dataset contains data about and snomed

are of particularly high value, the scores of these attributes are double weighted. If we sum up the scores, we get a resulting scale from 0 to 1000. That means, that a dataset of perfect quality will get a score of 1000, while a dataset with no information content will get a total score of 0. This part of the algorithm is pure. Calling it in any point of time using the same dataset will result in the same total score.
The next step is to get the amount of MediCoins corresponding to the total score. As described in section 2.3, each disease has a “budget”, a number, that represents the remaining amount of MediCoins, that medicalvalues will pour out as reward for contributions for this particular disease. We want to pour out the reward by fraction. We do this, because earlier contributions are more valuable! A dataset about a new or fatal disease is far more valuable than a dataset about a long known disease, that can already be diagnosed and cured reliably.
For a perfect dataset, the user will receive 5% of the remaining budget of the budget. To get the percentage the dataset is worth, the total score is multiplied by 5. This means, that a perfect dataset with a score of 1000 gets a “percentage value” of

$$1000 \times 5 = 5000 ,$$

which is equivalent to 5%. A dataset with a total score of 750 would be translated to a “percentage value” of

$$750 \times 5 = 3750 ,$$

which is equivalent to 3,75%.

2.3 Diseases and MediCoins

MediCoin is a token, used to credit the users for contributing data. It is used as a proprietary currency inside Medi Systems and is considered as currency to exchange royalties and discounts for products of medicalvalues. MediCoins can be minted at any time in point by the contract owner.
We decided to not limit the total supply of MediCoins, as that would involve us to take inflation and deflation into consideration.

Every recorded disease in Medi Systems has a “budget”, represented by a number. This budget is the maximum amount of MediCoins, that can be poured out as reward for datasets about this disease. The budgets have the purpose to weigh importance to specific diseases. As written in section 2.2.2, a dataset about a new or fatal disease is far more valuable than a dataset about a long known disease, that can already be diagnosed and cured reliably. Following this principle, the user should get less reward for the same dataset, if he shares it after another user’s contribution, compared to before it.
Using the concept of adjustable budgets, medicalvalues can manually specify, what disease they want to focus on. It is expected, that users will try to share data about diseases with higher budgets, to get a higher amount of MediCoins. After all, the rewards are poured out by fraction, which means, that diseases with higher budgets will give higher amounts of MediCoins.

2.4 Transferring MediCoins

The user process flow is as follows. The smart contracts, including MediCoin get deployed on the blockchain by medicalvalues and some sort of web application gets released. A doctor accesses our web application and registers to our platform. He/She goes to the interface for contributing datasets and shares a file. The reward should be poured out instantly and automated. This last step poses a problem, because in order to pour out the reward, the concrete amount of MediCoins need to be. transferred from the deployers account to the user's account. This is not easily done, as the use cannot access a foreign account and transfer MediCoins to himself, if that would be a security flaw. Still, the reward system should be as automated as possible. The application would be poorly scalable, if medicalvalues needed carry out every MediCoin transfer for every contribution manually. The ERC20 standard has a concept of approval and allowance. A user X can approve a user Y to spend a specified amount of tokens. The `allowance` function will then return the amount of MediCoins that user Y can spend from user X.

Ideally, this approval would be given automatically every time a user evaluates his dataset and calls the evaluation function. This is not possible though, as the possibility of users approving themselves imposes a security risk. This could be avoided, by reworking the access control of the token contract, though that would be beyond the scope of the project.

Instead, we decided on a simpler, albeit less elegant solution that approves the users in advance. The smart contract has an array, which stores all user addresses of users, who need approval. Right after registration, the user's address gets added to this array. The user is in a state of **registered but not approved**. In this state, the user will be kept from interacting with the application. Contributing datasets would not work properly anyway, as he would not receive the award. The contract function `approve` needs to be called from the deployer's address for every user. This means, that medicalvalues needs to approve every newly registered user manually. This happens through an Admin Page, where medicalvalues will be displayed a list of users, waiting for approval. A simple button click will approve the user a default amount of MediCoins, which he then can transfer himself after contributing datasets.
This approach of approving every users high amounts of MediCoins presents a new concern, as we have to prevent, that users randomly transfer arbitrary amounts of MediCoins to themselves. To solve this problem, we implemented the concept of **pending** datasets. Every doctor is represented by an instance of a struct Doctor. The struct contains a member called `pending`, which is of type `Dataset`. To see detailed explanation of these structs, see section 3.2.2 . The important point is, that after the value of the dataset is calculated, a `Dataset` instance is created, which has the value of the dataset, a concrete amount of MediCoins, stored inside of it. This Dataset is then stored in the `pending` attribute of the `doctor` instance, that corresponds to the calling user's address. When the user accepts the calculated amount of MediCoins as reward in the web application and he tries to transfer MediCoins from the account of medicalvalues to himself, it is first checked, if that transfer is valid. The transfer is only valid, if the desired amount of MediCoins matches the value in the `pending` dataset of the `Doctor` instance, that corresponds to the calling user.

3. Smart Contracts

3.1 MediCoin

The MediCoin smart contract inherits from the ERC20 contract from Openzeppelin. A MediCoin is a token, used to credit the users for contributing data. It is used as a proprietary currency inside Medi Systems and is considered as currency to exchange royalties and discounts for products of medicalvalues.
MediCoin consists of four contract attributes, a constructor, one modifier and seven own functions, besides the ones from the inherited parent ERC20 contract. .

3.1.1 Attributes

Attribute	Type
owner	address
minter	address
mediSystemAddress (deprecated)	address
_totalSupply (deprecated)	uint256

`owner` stores the address of the contract owner, who is the deployer of the contract. Currently, there is no way to change the owner address after contract creation, therefore it is not possible to change the owner of the smart contract. For future versions, it should be considered, whether or not is is useful to implement. As the project was developed for medicalvalues, we decided to not make the owner changeable.
In the following of section 2.1, the terms owner and deployer are used interchangeably. Both refer to the deployer of the contract, and are meant to be the address of medicalvalues.

`minter` stores the address of the account, which is allowed to mint new MediCoin tokens.

`mediSystemAddress` (deprecated) should store the address of the MediSystem smart contract. This attribute is used in the modifier *hasAccess* , as the methods inside the MediCoin contract should be callable from the MediSystem contract.
This attribute is now deprecated, as the MediSystem contract inherits from the MediCoin contract, making the access modifier redundant. The attribute will be removed.

`_totalSupply` is unused and will be removed. The total supply of MediCoins is accessible via the super function of the ERC20 token contract.

3.1.2 Modifier

Modifier	hasAccess

`hasAccess` restricts the access to functions for users other than the contract owner. Functions with this modifier in their signature can only be called from the contract owner, which in this case is always the deployer.

3.1.3 Constructor

The constructor calls the super constructor of the ERC20 contract that MediCoin inherits from, sets the name of the token to “MediCoin” and the symbol to “MDC”. It then assigns the deployers address to the attributes owner and minter. It also mints a provisional amount of 1000 MediCoins and assigns them to the owner.

3.1.4 Functions

Function	Parameters	Returns
setMediSystemAddress	void	void
defaultApprove	spender	bool
approve	spender, amount	bool
decreaseAllowance	spender, subtractedValue	bool
increaseAllowance	spender, addedValue	bool
transfer	recipient, amount	bool
mint	amount	bool
Inherited Functions		
transferFrom	sender, recipient, amount	bool
_mint	account, amount	void

`setMediSystemAddress` (deprecated) is a function, takes an address «_mediSystemAddress» as a parameter and assigns it to the attribute mediSystemAddress. As the attribute itself is deprecated, so is this function. This function will be removed in the future.

`defaultApprove` is a function, that takes an address «spender» as parameter and approves the corresponding account to transfer a maximum amount of 1000 MediCoins from the deployer account, who initially holds all of the MediCoins.
The function has the modifier «hasAccess», which ensures, that only the contract owner can call the function. This makes sense, as only medicalvalues should be able to approve users to transfer MediCoins from medicalvalues’ account. Without this modifier, all users could approve each other to transfer, and therefore sharing each others tokens, which is not intended. For this reason, the func-

tion transfer also has the modifier «hasAccess», which prevents users to sending MediCoins to each other. Internally, the function calls the «approve» function from the super contract, and passes the «spender» as the to be approved account, and 1000 MediCoins as the amount, that the spender can transfer from the deployers account.
If the function is called when the given account already has allowance, the remaining allowance will be overwritten by the newly approved 1000 MediCoins. The function returns true, if approval was successful.

`approve` is a function, that takes an address «spender» and an uint «amount» as parameters. This function overrides the super function. It does almost the same as defaultApprove, except for the amount of allowance, which in this case can be specified using the «amount» parameter.
If the function is called when the given account already has allowance, the remaining allowance will be overwritten by the given «amount» of MediCoins. The function returns true, if approval was successful.

`decreaseAllowance` is a function, that takes an address «spender» and an uint «subtractedValue» as parameters. It decreases the allowance of the «spender» by the given amount «subtractedValue». The function has the modifier «hasAccess», which ensures, that only the contract owner can decrease the allowance of other users. As other users cannot approve allowance to each other anyway, this function is inaccessible to them.
The function returns true, if the allowance has been successfully decreased.

`increaseAllowance` is a function, that takes an address «spender» and an uint «addedValue» as parameters. It increases the allowance of the «spender» by the given amount «addedValue».
The function has the modifier «hasAccess», which ensures, that only the contract owner can increase the allowance of other users. As other users cannot approve allowance to each other anyway, this function is inaccessible to them.
The function returns true, if the allowance has been successfully increased.

`transfer` is a function, that takes an address «recipient» and an uint «amount» as parameters and transfers the specified «amount» of MediCoins from the callers account to the one of the «recipient». It overrides the super function and only differs from it in the access modifier.
The function has the modifier «hasAccess», which ensures, that only the contract owner can transfer MediCoins to other accounts. The function returns true, if the «amount» of MediCoins have been successfully transferred from the callers address, in this case medicalvalues, to the «recipient».

`mint` is a function, that takes an uint «amount» as parameter and mints that «amount» of new MediCoins and assigns them to the contract owner, in this case medicalvalues. Internally, it calls the super mint function and passes the address owner and the given «amount» to it.
The function returns true, if the specified «amount» of MediCoins has been minted successfully.

`transferFrom` is an inherited function. It takes two addresses «sender», «recipient» and an uint256 «amount» and transfers the given «amount» from the account of the «sender» to the «recipient».

_mint is an inherited function. It takes an addresses «account» and an uint256 «amount», creates new MediCoins and assigns them to the the «account».

3.2 MediSystem

The MediSystem smart contract currently inherits from the MediCoin smart contract. It is the main contract, that users and web-applications interact with.

3.2.1 State Variables

State Variables	Type
...{inherited variables from MediCoin}	
diseasesNames	string
allDoctorAddress	address
unapprovedDoctors	address
doctors	mapping(address => Doctor)
diseases	mapping(string => Disease)

diseasesNames stores all diseases added by the users.

allDoctorAddress stores all addresses of doctors, which are already checked-in.

unapprovedDoctors stores all addresses of doctors, which are not yet approved.

doctors is a mapping, which maps addresses to instances of the Doctors struct.

diseases is a mapping, which maps disease names, in the form of strings, to their corresponding Disease structs.

3.2.2 Structs

Doctor	
doctorAccount	address
doctorName	string
contributedData	bytes32[]
isPendingDatasetExist	bool
pending	Dataset
isExist	bool

Dataset	
fileHash	bytes32
value	uint
numberOfPatientsData	uint

Disease	
budget	uint
numberOFPatientsData	uint
name	string

Doctor is a struct, which contains 6 attributes.

doctorAccount stores the address of the doctor.
doctorName stores the name of the doctor.
contributedData is a bytes32 array, which stores the hash values of the files, that have been contributed by the doctor.
isPendingDatasetExist stores a bool value, which tells us whether a file is in the state of pending or not.
pending stores a pending dataset, if isPendingDatasetExist is true, the pending dataset has valid data.
isExist stores a bool value, which shows whether this doctor exists or not. If the value is set to true, the doctor is already successfully registered.

Dataset is a struct, which contains 3 attributes.

fileHash stores the hash value of the file.
value stores the value of the dataset, expressed in MediCoins.
numberOfPatientsData stores how many patients the dataset contains information about.

Disease is a struct, which contains 3 attributes.

budget every disease has a budget, which implements the concept described in section 2.3.
numberOfPatientsData stores how many patients are in the dataset.
name every time a disease is instantiated, a disease name is required.

3.2.3 Modifier

Modifier	isOwner	isRegistered

isOwner check whether the caller is the owner, here deployer or not. This modifier restricts non-developers from accessing certain functions.
isRegistered check whether a doctor is already registered.

3.2.4 Constructor

The constructor assigns the deployers address to the inherited state variable **owner**.

3.2.5 Functions

Function	Parameters	Returns
getAllUnapprovedDoctors		address
approveDoctor	doctor	void
registerDoctor	doctorName	void
addDisease	budget, name	void
getMyName	account	string
getDoctorName	_doctorAddress	string
evaluation_attritube	_attributeAmount	uint
parseStringToUint	_string	uint
getGenderValue	gender	uint
getAgeValue	age	uint256
getNumberOfPatientsValue	numberOfPatients	uint256
getLoincVal	loinc	uint256
getRadlexVal	radlex	uint256
getSnomedVal	snomed	uint256
calculateDatasetValue	account, disease, numberOfPatients, age, gender, numberOfAttributes, loinc, radlex, _fileHash, snomed	void
addPendingDataset	account, _fileHash, amount, numberOfPatients	void
getDataSetValue	account	uint256
abortContribution	account, diseaseName	
getTotalDatasetValuePercentage	numberOfPatients, age, gender, numberOfAttributes, loinc, radlex, snomed	void
getIsDiseaseExists	content	bool
contributeData	_fileHash, _address, amount	void
getIsApproved	_address	bool
getDiseaseBudget	diseaseName	uint
getDiseaseNumberOfPatientsData	diseaseName	uint
getIsPendingDataExists	_address	bool
Inherited Functions		
transfer	recipient, amount	bool

`getAllUnapprovedDoctors` is a function, that does not take any parameters.

The function will return all addresses of all unapproved doctors in an string array.

`approveDoctor` is a function, that takes an address «doctor», which is the address of an instantiated doctor.
The function has the modifier «isOwner», which ensures, whether the user of the function is owner or not and then removes the doctor with the given address from the array unapprovedDoctors.

`registerDoctor` is a function, that takes a string «doctorName», which is a name of doctor.
The function instantiate a doctor by a name, after a doctor instantiated, it will be pushed into an array of unapprovedDoctors.

`addDisease` is a function, that takes a uint «budget», which is the initial budget and a string «name», which is the name of the disease.
The function instantiates a disease, and adds it to the mapping `diseases`.

`getMyName` is a function, that takes an address «account».
The function returns the name of the user corresponding to the given «account».

`getDoctorName` is a function, that takes an address «account» and returns the name of the doctor corresponding to the given «account».
The function has the modifier «isOwner», which restricts, that only owner can get the name of doctor.

`evaluation_attribute` is a function, that takes a uint «_attributeAmount», which means how many attributes are in the dataset.
The table below shows the corresponding scores for the number of attributes in the dataset.

Number of Attributes in the Dataset	<7	(7, 12]	(12, 18]	(18, 24]	(24, 30]	>30
Resulting score	0	20	40	60	80	100

`parseStringToUint` is a utility function, that takes a string «_string», and transforms it to an `uint`. «_string» is expected to be a string only containing numbers.
The function contains four loops. The first loop assigns the numbers from 48 to 57 in an array named `array1`. The second for loop loops through the parameter «string» and transforms every character to a `bytes1` type. Afterwards it transforms those `bytes1` into a `uint8` number, that means the number from 48 to 57 and stores these number into an second array `array2`. The third loop compares the two arrays. If the elements are the same, then it assigns the index into the array2. The last loop combine all of the single numbers into one number.

`getGenderValue` is a function, that takes a string array «gender», which is expected to have at least 6 elements. The first index is expected to contain any expression for the “male” gender, the second to contain the occurrences of male. The third index is expected to contain the any expression for the

“female” gender, the fourth index to contain its occurrences. Indices 5 and 6 follow the same principle for “transsexual” gender. If there are more than 6 elements, the rest of elements are false value. The array is structured in such a way that the even indices contain the gender expressions. The odd indices following the respective expressions contain the number of occurrences.
The function returns a score (uint256) between 0 and 100.

Amount of falsy gender values	0	<6	(6, 12]	(12, 18]	(18, 24]	>24
Resulting score	100	80	60	40	20	10

`getAgeValue` is a function, that takes a string array «age» , which contains 3 elements max value, min value and the number of false values. The function returns a score (uint256) between 0 and 100.
If the min age value of the data set is less than 0 or the max age value is greater than 150, the function returns a score of 0. This allows us to check whether the rough age range has been fulfilled. If this is the case, the function continues with the incorrect age entries. The table below shows the corresponding scores for the number of falsy age values.

Amount of falsy age values	<6	(6, 10]	(10, 20]	(20, 30]	>30
Resulting score	100	70	40	10	0

`getNumberOfPatientsValue` is a function, that takes a uint256 «numberOfPatients».
The function returns a score (uint256) between 40 and 100.
The table below shows the corresponding scores for the number of patients.

Number of patients	<= 200	(200, 400]	(400, 600]	(600, 800]	(800, 1000]	>1000
Resulting score	40	60	70	80	90	1000

`getLoincVal` is a function, that takes a bool value «loinc», which is true or false. The function returns a score (uint256) of either 0 or 100.
In case the Loinc value is true, the function will return a score of 100. Is this not the case the function will return a score of 0.

`getRadlexVal` is a function, that takes a bool value «radlex», which is true or false. The function returns a score (uint256) of either 0 or 100.
In case the Radlex value is true, the function will return a score of 100. Is this not the case the function will return a score of 0.

`getSnomedVal` is a function, that takes a string array «snomed», which contains 2 elements, the first is “true” or “false”, the second is count of false value. The function returns a score (uint256) between 0 and 100.

So the function first checks via the boolean if snomed entries exists. If a snomed entry exists the function will continue, if not it will be aborted and returns 0 as the score. Then the function checks the number of falsy snomed values. The table below shows the corresponding scores for the number of falsy snomed values.

Amount of falsy snomed values	<6	(5, 10]	(10, 15]	(15, 20]	(20, 25]	>25
Resulting score	100	80	60	40	20	0

`calculateDatasetValue` is a function, that takes an address «account», that is the senders address, a string «disease», which is the name of the disease, that the dataset is about, an uint «numberOfPatients», which is the number of patients the dataset contains information about, a string array «age» with information about the age of the patients in the dataset, a string array «gender» with information about the gender of the patients in the dataset, a uint «numberOfAttributes» which is the number of columns in the dataset, a bool «loinc» that indicates the existence of loinc data in the dataset, a bool «radlex» that indicates the existence of radlex data in the dataset, a bytes32 «_fileHash» with the hash of the dataset and a string array «snomed» with information about snomed data in the dataset as parameters.
The function implements the concept of calculating the value of datasets, described in section 2.2.2. First, the helper function `getTotalDatasetValuePercentage` is called. The returned value, which is referred to as the **total score** in section 2.2.2 and ranges from 0 to 1000, is multiplied by 5. This is a value ranging from 0 to 5000. To get the the correct percentage, this number is divided by the conversion rate 100000. This results in a total of 5% of the Medicoins assigned to the disease if all points are reached. To get the amount of MediCoins that the dataset is worth at the that particular point in time, this percentage number is multiplied by the budget of the disease. The calculation is as follows:

$$\frac{\text{total score} \times 5}{100000} \times \text{budget}$$

The problem is, that

$$\frac{\text{total score} \times 5}{100000}$$

can lead to a decimal number less than one, which in solidity results in 0. To avoid the problem, we execute the calculation in the following sequence:

$$\text{total score} \times 5 \times \text{budget} : 100000$$

Because the budget is a number with 18 positions dedicated for decimals, the length of the number should be very large. After multiplying it by 5 and then the total score, which ranges between 0 and 5000, the number should be so large, that dividing it by 1000000 should be no problem. It may happen that in case of the lowest rating of a dataset and from a budget of a disease below 2000 MediCoins, no MediCoins will be paid for the dataset. For a record with such a bad rating as the 10 out

of possible 1000 points, no MediCoins should be paid either. Therefore, this problem can be neglected.
After the value has been calculated, it will be subtracted from the budget of the disease. Then, the attribute `isPendingDatasetExist` of the Doctor struct, that corresponds to the «account», which should be the currently logged in user, is set to `true`. A Dataset struct is created for this to be contributed dataset and stored in the `pending` attribute of the Doctor struct.

`addPendingDataset` is a function, that takes an address «account», which is an address of an account; a bytes32 «_fileHash», which is a hash value of a file; a uin256 «amount», which is an amount of files; a uint256 «numberOfPatients», which is count of patients.
The function instanced a Dataset and the set this dataset in corresponds doctor as pending.

`getDataSetValue` is a function, that takes an address «account», which is an account of a doctor.
The function will return the value of the pending dataset in this account.

`abortContribution` is a function, that takes an address «account»; a string «diseaseName».
The function will judge whether if in this account a dataset exist, that can be aborted. if exist, returned the corresponds value back to disease and then delete the pending dataset.

`getTotalDatasetValuePercentage` is a function, that takes a uint «numberOfPatients», which is the number of patients the dataset contains information about; a string array «age» with information about the age of the patients in the dataset; a string array «gender» with information about the gender of the patients in the dataset; a uint «numberOfAttributes» which is the number of columns in the dataset; a bool «loinc» that indicates the existence of loinc data in the dataset; a bool «radlex» that indicates the existence of radlex data in the dataset and a string array «snomed» with information about snomed data in the dataset as parameters.
The function adds up the results of all the following functions getNumberOfPatientsValue, evalueation_attribute, getSnomedVal, getAgeValue, getGenderValue, getLoincVal, getRadlexVal.

`getIsDiseaseExist` is a function, that takes string «content», which is the name of a disease.
The function checks if a disease exists in the system.

`contributeData` is a function, that takes a bytes32 «_fileHash», which is a hash value, it can uniquely identify a file; an address «_address», that is an address of doctor; a uint «amount», which is amount of Dataset, that the doctor want to contribute.
The function at first set up three conditions, that are `doctorsIsPendingDatasetExist ==` true, `pending.value ==` «amount», `pending.fileHash ==` «_fileHash», if all of these conditions are satisfied, implement the function next step transfer, after successfully transfer, the function add new number of patient Dataset into corresponds disease and delete pending dataset.

`getIsApproved` is a function, that takes an address «_address» which is an address of a doctor.
The function will traverse all elements in the array `unapprovedDoctors`, if an address is not in this array, then return true, otherwise return false.

`getDiseaseBudget` is a function, that takes a string «diseaseName», which is name of disease.
The function return the budget of a disease.

`getDiseaseNumberOfPatientsData` is a function, that takes a string «diseaseName», which is name of disease. The function return the number of patient dataset.

`getIsPendingDataExist` is a function, that takes an address «_address», which is the address of a doctor. The function return the state of `isPendingDatasetExist`.

transfer is a function, that takes an address recipient, which the address can get the MediCoins;
a uint256 «amount», which restricts the amount of the MediCoins.
The function override transfer function from MediCoin.sol, the allowance is also very important, if the allowance is low, add his address to the unapprovedDoctors array. so that medicalvalues can refill his allowance.

4. Web-Application

The Medi System web application(web app) currently consists of a single frontend module, which can be located in the project directory `blockchain_hackathon_2021/Frontend`. The web app allows users to interact with the smart contract, such as contributing datasets, reading their MediCoin balances, etc.

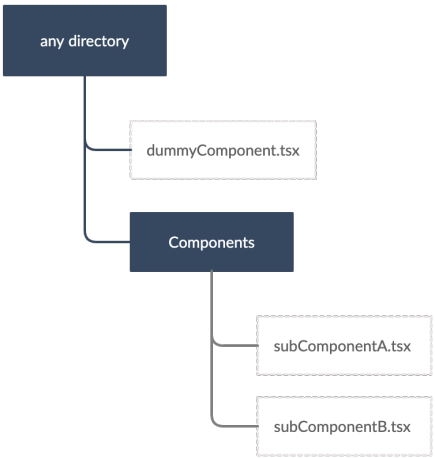
4.1 Architecture and Structure

The frontend is built using React.Js and a Redux store for global state management. Most of the code is written in Typescript for type-safety. The project uses Webpack as module bundler.
For calling functions of the MediSystem smart contract, the framework useDApp is used.

4.1.1 File Structure

Each React component (in the following just “component”) is located in its own typescript file. The file and the contained component share the exact same name, with the difference, that the file is written in camel-case, and the function itself is named in pascal-case. If the component uses a sub-component, that is only used in in this component, the sub-component should be placed in a directory `Components/` at the same level the file is located.
The file itself has all the imports at the top. All external library imports, like `React`, `useDispatch`, `useHistory`, etc are listed first. Separated by an empty line, our own imports are listed below.

If the component takes props, the props type is specified directly above the component. The name of the type is prefixed with the name of the component and ends with the suffix “`Props`“ .



Inside the function component, the separators `// – STATE –`, `// – HELPERS –`, `// – EFFECTS –`, `// – RENDER –` , etc. help keeping the code readable and organised.
Styles are written as typescript objects and type-checked by the React.CSSProperties interface, either in the same file, inline in JSX or imported from a separate typescript helper/utils file. In the future, this needs to be cleaned up and made more consistent.

4.1.2 Folder Structure and Naming

All frontend source code is located in the directory `blockchain_hackathon_2021/Frontend/src/` . This directory is split into subdirectories named `BaseComponents`, `Illustrations`, `Pages`, `State` and `Utils`.

`BaseComponents/` contains components, that are used across the application.

`Illustrations/` contains all illustrations and images, that are used across the application.

`Pages/` contains subdirectories. Each of them represents one page of the application. Each page subdirectory contains a typescript file, named the same as the subdirectory but in camel-case instead of pascal-case. This can be seen as an index file for the specific page. If the page has components, specific to it, they are placed inside a directory `Components/` inside the page subdirectory, following the same principle as described in section 4.1.1.

`State/` contains the code for the redux store, reducers and actions.
The subdirectory **`State/Actions`** contains files for action -creators, -types and -interfaces.
The subdirectory **`State/Reducers`** contains files for the reducers. In total, the web app uses two reducers. One for modal states, and one for the Contribute-Data-Page.

`Utils/` contains styles, enums, hooks, util functions, etc, that are used globally across the project.

`Contracts/` is configured as the build directory in truffle configuration. All the Application binary Interfaces are built to this directory. The `MediSystem.json` file, is being default imported in `hooks.ts`, as `mediSysAbi`, which is then used to instantiate an `ethers.js` Interface.

As written above in 3.1.1 File Structure, a sub-component specific to a component should be placed inside a directory named `Components`. The same principle applies to util files like styles or helper files. Constants, styles or utility functions, that are only used in a single component, should be stored in their own **`utils.ts`** or **`styles.ts`** at the same level as the main component. If multiple util files are needed, they should be placed in their own directory named **`Utils`** .

4.1.3 Routing

Routing is done client side using the React-Router module. All used paths can be found in enums in `blockchain_hackathon_2021/Frontend/src/Utils/paths.ts` .

4.1.4 Pages

The web app has the following pages, that can be accessed by the user: Landing-Page, Demo-Page, Admin-Page, Contribute-Data-Page, Account-and-History-Page. For each page, there is a React component named accordingly, suffixed with `page.tsx` .

4.1.5 Hooks and useDApp

All custom hooks are located in `blockchain_hackathon_2021/Frontend/src/Utils/hooks.ts` . The most important ones are described down below.

The **useContractCall** hook from the useDApp API is used to call read-only functions in smart contracts. In Medi System, it is currently used for the custom hooks `useMyName`, `useGetDatasetValue`, `useGetMyMediCoinBalance`, `useGetMediCoinAddress`, `useGetIsOwner`, `useGetGenderValue`, `useGetAllUnapprovedDoctors` and `useGetAmIApproved`.

`useMyName` returns the the registered name of the currently logged in user. The hook calls the function `getMyName` of the MediSystem contract. Currently, if the user is in any other login-state different from “logged in”, the hook returns a falsy value, typed as `any`. This should be fixed in the future to ensure correct type guarding. Like some of the other following hooks, the return value of falsy values should be typed as `null` or `undefined`.

`useGetDatasetValue` returns the value of the uploaded dataset, expressed in MediCoins. The returned value of type string only contains numbers. Because solidity cannot deal with decimal numbers, the last 18 characters of the string represent the decimals. As a result, the return value of the hook should be parsed using **`parseMediCoin`** located in `blockchain_hackathon_2021/Frontend/src/Utils/utils.ts` . The hook has a state variable, initially set to null. If the return value of the

contract call to the `getDataSetValue` function of the MediSystem contract changes, an effect is triggered. If the returned value is falsy, the effect does nothing. If the returned value is a non falsy value, the effect sets the state variable to the value. After that, the state variable is returned, which means, that the hook is properly typed, returning either a string, or null.

`useGetMyMediCoinBalance` returns the MediCoin balance of the currently logged in user as a string, if one exists, else it returns null. Internally, the hook uses one state variable, initially set to null. An effect is triggered, every time the return value of the contract call to the `balanceOf` function in the MediSystem contract changes. If the returned value is falsy **and** the does not equal the BigNumber representation of the number, the effect returns without any changes. The second condition is needed, as 0 is a falsy value in typescript, but a valid balance in our application. If the condition does not apply, the state variable is updated to the returned value.

`useGetIsOwner` returns true, if the logged in user is the contract deployer, medicalvalues. Internally, it calls the getter of the public contract attribute `owner` to retrieve the contract deployer's address. It then returns the

`useGetMediCoinAddress` is deprecated, as the MediSystem contract currently inherits from the MediCoin contract. This hook was initially planned to be used in the Admin Page, where the admin user could control and set the address to the MediCoin contract.

`useGetIsOwner` returns true, if the currently logged in user is the contract owner, medicalvalues, and returns false, if not or no user is logged in. Internally it calls the getter function of the public contract attribute owner to retrieve the address of the contract deployer. Then, the address is compared to the address of the current user. The hook is type safe.

`useGetAllUnapprovedDoctors` returns a string array with the addresses of all users, who are not approved (have no allowance) in the MediSystem smart contract. Internally it calls the contract function `getAllUnapprovedDoctors`. Because that array may contain zero addresses, that are caused by deleting array fields in solidity, an effect edits the array by deleting those addresses before returning it.

The **useContractFunction** hook from the useDApp API is used to call state-changing functions in smart contracts. In Medi System, it is currently used for the custom hooks `useMediSysMethod` .

`useMediSysMethod` takes a string «functionName» as parameter and returns an object containing a variable `state` of type TransactionStatus and a function `send`, that will call the MediSystem contract function named «functionName» when called. This returned function takes any arguments the called contract function takes and maps them one to one. When the this function is called, Metamask will open a pop-up window asking the user to confirm the transaction.

4.1.6 Redux Store

The redux store has two reducers and two states. One reducer is dedicated for the `ModalState`, and one for the `ContributeDataPageState`. The ModalState, which can be found in the directory `blockchain_hackathon_2021/Frontend/src/State/modalReducer.ts` has two variables. The variable `isRegistrationModalOpen` is a boolean, that the `RegistrationModal` component uses to save its state. Depending on the boolean value, the `RegistrationModal` is either opened or closed. Same applies to `isSettingsModalOpen`. Though, a settings modal component is yet to be implemented.
The `ContributeDataState`, can be found in the directory `blockchain_hackathon_2021/Frontend/src/State/contributeDataPageReducer.ts` . It is used to store the important information, that the `FileUploader` component of the `ContributeDataPage` extracts from the parsed csv file. The stored data is partly used on the contributeDataPageDataSetValueDetailsPage page, which selects the information from the store and passes them to the smart contract helper functions to get normalised evaluation scores for each dataset quality attribute. These are then used to give the user a visual representation of the quality of his dataset.

4.2 Functionality

4.2.1 Log In and Approval

There are three states, a user can be situated in. If the web application is disconnected from Metamask, the user is considered to be **logged out**. If Metamask is connected, but the user is not registered in MediSystems, the user is considered as a **guest**.
If Metamask is connected and the user is registered, but not approved, the user is considered **registered but not approved**. In this state, users can log into the web application, but will see a not closable modal, informing the him that his account needs to be approved before he can use it. Users, who have completed several contributions already, will find, that their allowance decreased. If the allowance of the user falls short of a threshold, the user is situated in this same state again.
If Metamask is connected and the user is both registered and approved, he is considered **logged in**.

4.2.2 Contribute Datasets

Contributing datasets is the most critical functionality of Medi Systems. By contributing datasets, doctors improve the knowledge graph of medicalvalues, as their data is used for training the graph using machine learning algorithms.

To contribute datasets, users navigate to the Contribute Data Page using the navigation bar. First, they specify the disease, their dataset is providing data for. In the future, this will be a searchable dropdown menu, so that duplicated disease names for a single disease will be avoided.

Underneath the input field, there is a droppable area, where users can drag and drop their datasets. Alternatively, clicking on the area opens the file explorer, where the file can be selected. Currently, only csv files are supported. After the file has been selected, Papa Parse parses the file and the resulting array is stored in the state variable `selectedFile`. The change triggers an effect, that runs every time `selectedFile` changes and processes the array. It extracts information like number

When the name of the disease and the dataset are provided, a button labelled Calculate Value will appear. Clicking on it will execute the callback function handleCalculateValue, which internally executes the function `calculateValue` and passes in the parsed information from the csv file. This function is returned by the `useMediSysMethod` hook. The function `calculateDatasetValue` in the MediSystem contract will be triggered, which calculates the dataset value and adds a `pending` dataset to our user. During this contract function call, an effect is run, every time the transaction state `calculateValueState` or `datasetValue` changes. The variable `datasetValue` stores the result of the hook `useGetDatasetValue`. If the dataset has been successfully evaluated and the web application has retrieved the value of the dataset, the effect pushes the path of the DataSetValueDetailsPage onto the history stack.
On this page, the user will see a visual representation of the quality of his dataset. Currently, some of the data with absolute values, on which the quality is evaluated on, is coming directly from the redux store.

A better approach is to call the helper functions of the MediSystem smart contract, such as `getGenderValue` and pass in the needed data from the redux store. These helper functions will return normalised values, ranging between 0 and 100. Which means, that the frontend does not need to interpret the absolute numbers, but rather can work with percentages. Most importantly, the value of the dataset is displayed below.
The user now sees the calculated value of his dataset for the first time and can decide between aborting the contribution, or accepting the offer.
Aborting the contribution will currently call the callback `handleGoToFileUploader` , which redirects to the first Page of the ContributeDataPage. In the future, the redux store will be reset to its initial state, and the abortContribution contract function needs to be called. These steps also need to be executed, when the page component unmounts, e.g. when the user navigates to a different page using the navigation bar. Otherwise, the states will get inconsistent.
Submitting the contribution runs the callback function `handleContributeData`, which calls the contract function `contributeData` using the `useMediSysMethod` hook.

4.2.3 Admin Panel

The Admin Page is only accessible to the smart contract deployer, medicalvalues. Access control is done in the NavBar component using the **`useGetIsOwner`** hook. Depending on wether the hook returns true or false, the component is displaying the Admin Page as an option, or not. In the future,

some sort of access control is also needed in the `AdminPage` component itself too, as the page is still accessible to anyone navigating by url.
The Admin Page currently only displays the `AdminPageApproveDoctorsTable` component. This component uses the **`useGetAllUnapprovedDoctors`** hook to retrieve an array of all users, that are registered, but not approved. Using this component, medicalvalues can approve doctors by clicking the button next to the users address. After approval, the user can use the system normally.

5. Limitations and ToDos

With regard to the limitations, adjustments must be made above all in the approving of the users. Because currently every new registered user has to be approved by the owner. Only then can users upload datasets, have them evaluated and, if necessary, contribute them. Currently, the MediCoin budgets must also be minted manually for new diseases. In the future, this will happen automatically if there is no budget for the corresponding disease. This means that there were no datasets for the corresponding disease before. Overall the balancing between the minting and totalSupply of MediCoins, amount of allowances and the size of the budgets of each disease need to be tweaked heavily.
Currently, the whole Account and History Page is static. The concept is, to read the events, which contain the hash values of the datasets, that the user has contributed in the past. After each contribution, the csv-file gets should be moved to a local directory, as those files cannot leave the local facility. Using the hash values retrieved from the events, the web application should search the file in the local direectory, parse it, and the call the helper functions in the smart contract, such as `getGenderValue` and format the data to display them on the Details Pane for each dataset.
In order to do this, a traditional backend server running locally is needed. The server can access local directories and serve them to the frontend via an API. This way, the user does not need to open the file explorer and select all files manually. Also, a server is needed to move the csv file to the local directory.