5 Development

5.1 General Structure

The main purpose of this thesis is to develop an information system (IS) for visualizing wicked problems. With the previous literature review and the use of modern technologies, the IS has been developed as a progressive web application using the cloud (**wickedproblems.io**). Web applications have the ultimate advantage of not being reliant on one specific device or hardware. Thus, it is accessible over the internet from anywhere, and on any device with a browser and internet connection. Similarly, the data is stored and accessed in the cloud, meaning it is also not dependent on one device. This configuration enables permanent accessibility and availability independently from time and place. To ease conversion of new users and security, firebase authentication by Google is used. While it reduces the time and cost of development, it also enables users to access the IS with their already existing Google account, or the accounts of other providers such as GitHub. In addition, it provides production-ready packages for securing backend components.

The base of problem visualization in the developed IS relies on the IBIS notation that has been explained before. Nevertheless, the notation has been certainly extended and further developed by suggestions found in numerous literature and own ideas. Apart from that, the role of data has been integrated with separate features, taking into account the research of Bibri and Acm (2019) and Schoder et al. (2014).

5.2 Challenges

Before starting with development, some challenges have been identified. On the one hand, there are technological challenges that are addressed by the choice of technologies and type of software architecture. When it comes to a long-term wicked problem with a large amount of information, the IS must keep high performance, given that it is running in a browser on web technologies. Since the IBIS notation consists of nodes, the application must be able to handle a large amount of them in one view. High performance is especially important for scalability over time. Performance is also relevant on the backend, where the API should serve multiple sessions at once.

Furthermore, the ability to collaborate on wicked problems must be supported in a usable and efficient way. Especially when multiple users are working on the same wicked problem, the IS must ensure that the changes are synced between the sessions of different users. Again, scalability here is important as well. While 2 or 3 users are easy to handle, it will become more difficult as the number of users increases.

Finally, special attention must be given to the usability of the IS. Decision-makers and stakeholders of wicked problems should not feel overwhelmed by the IS. They must rather feel supported in their decision-making and problem visualization without having to think about usability for too long. A typical scenario of using the IS would be a group of people discussing a wicked problem and simultaneously using the IS to map their thoughts and view their ideas and information. Therefore, it must be easy for them to quickly add their inputs to the application and move back to their discussion. Having a too complicated UI will deter attention from the actual problem and cause the users to leave the application.

5.3 Features

This section will explain the features implemented in the IS that also set the objectives of the application. As mentioned before, the features are extracted from the literature found as well as own ideas.

5.3.1 Canvas view

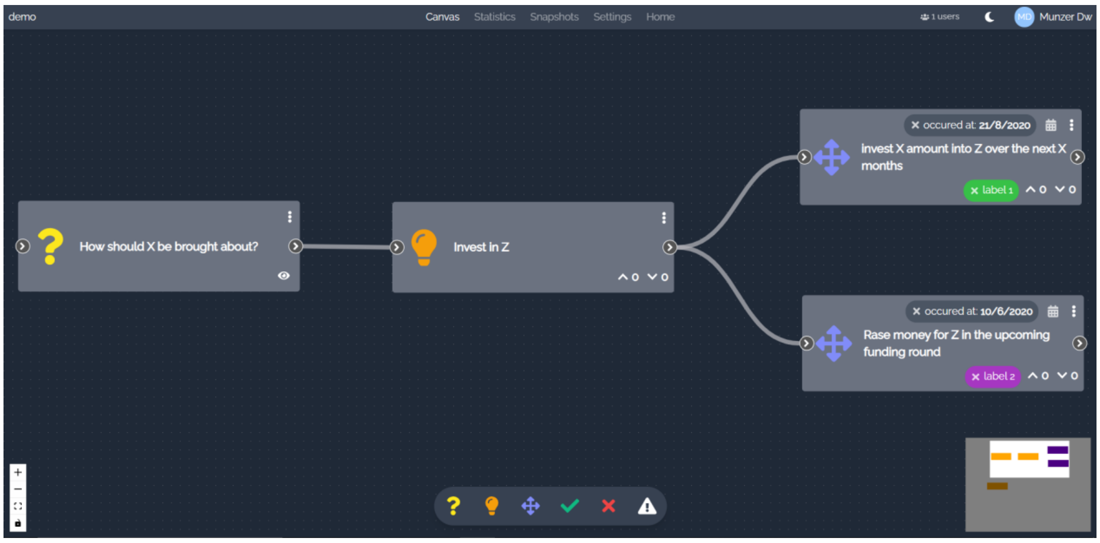


Figure 10: Canvas view
Source: Own illustration (wickedproblems.io)

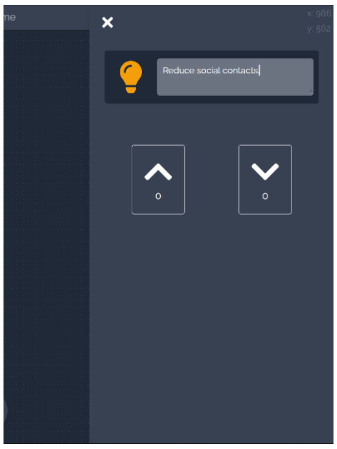


Figure 11: Node editor
Source: Own illustration (wickedproblems.io)

When starting a project (i.e. a wicked problem), the first page will render a canvas view (Figure 10) where the user can drag different types of nodes from the bottom bar and drop them into the field. As one can see, there are 5 types of nodes – question, idea, action, argument (pro or contra), and constraint. The nodes can be connected by their target and source handles on the left and right sides, respectively. The position of a node can be changed by holding down the node and dragging it to the desired position. Multiple nodes can be moved by holding SHIFT and selecting a range. To edit the text of a node, the user double-clicks on it to open the node editor on the right side (Figure 11). The app will automatically focus on the input field to enable fast editing. Other properties of a node can be adjusted in the node editor too. By clicking the 3 vertical dots on every node, the user can edit or delete the node. Selected nodes can also be deleted by simply clicking on the keyboard delete button. The canvas is endless, and it can be zoomed in and out by the mouse scroll wheel. On the bottom right side of the canvas view is the mini-map which shows an overview of all the nodes in their respective color. It also depicts the location of the user on the field. On the bottom left side of the screen, some controls are found, like zoom buttons, centering, and locking. Centering will bring the user back to the center of the canvas, while the locking will enable the user to click anywhere and move around freely without being able to interact with the nodes. Furthermore, the top menu enables navigation between the different pages. On the left, it shows the name of the project which in this case is “demo” (Figure 10). There is also an option to switch the theme between dark and light mode. While themes are usually fun customization for user preference, the light and dark modes also help improve usability during different times of the day. Dark mode will enable the user to use the application at nighttime, while light mode at daytime. Next to the user widget, the number of users is depicted. The user widget shows the user account name and enables drop-down actions such as signing out. The navigation menu is designed to be at the top with minimal height to enable as much space as possible for the canvas view.

5.3.2 Nodes

The question node, like in IBIS, illustrates an issue and has the option to hide the branch that is coming out of it, which is useful to keep an overview when the canvas view grows. The hidden property of a branch is saved in the database and the mini-map does not show hidden branches, only the starting question. Unlike IBIS, there is no UI element for the differentiation between the different types of issues. The second node is an idea, which is equal to the “position” element in IBIS. Eventually, the purpose of an idea is the same as a position, to provide an “answer” to a question. Renaming the node to “idea”, however, fits better with the third type of node “action”, which does not exist in the IBIS notation and is the new extension to it by this thesis. The reasoning behind an action node is to express a specific and concrete act that shows how an idea can be (or is) implemented. Worded differently, an action is usually what follows, after a position has been decided on. It is partly inspired by the “Climate CoLab” platform where participants enter a contest and create proposals about actions to be taken to tackle climate change. Let’s take for instance the handling of the Corona Pandemic (Schiefloe, 2021):

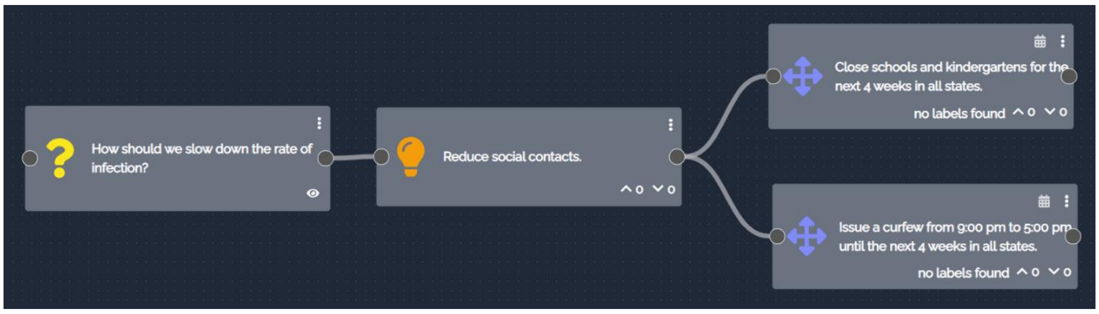

How should we slow down the rate of infection?
Reduce social contacts.
^ 0 ⌄ 0
Close schools and kindergartens for the next 4 weeks in all states.
no labels found ^ 0 ⌄ 0
Issue a curfew from 9:00 pm to 5:00 pm until the next 4 weeks in all states.
no labels found ^ 0 ⌄ 0

Figure 12: Example branch of nodes
Source: Own illustration (wickedproblems.io)

In this example (Figure 12), the issue of slowing down the rate of infection is presented as a question. It relates to the overall problem of how to "solve" the Corona pandemic problem. The first idea to the question suggests reducing social contacts, and it is only a suggestion at this point and does not provide a concrete statement on how it should be done. Here, actions 1 and 2 point out specific measures that must be taken to reduce social contact. So, in this case, the user can say what specific actions are available. The developed artifact by Noble and Rittel (1989) consists of the different types of IBIS issues: factual (Is X the case?), deontic (Should X be the case?), explanatory (What causes X?), instrumental (How can X be brought about?), and conceptual (What do you mean by X?). Furthermore, the authors use a position as an answer to these questions. In case of factual and deontic issues, where the positions can be "yes" or "no", an action can be connected to the positions and explain how they are realized. For example: "should X be the case?" would lead to the position "yes" which leads to the action "X can be the case by doing Y and Z." The other types of issues can result in a long list of positions. Here, actions are especially well suited for instrumental questions (Figure 12).
The action node is taken a step further. While having actions separated from ideas helps the overall visualization, it does not show what actions have happened. Therefore, the date feature is implemented (see the top right "calendar" icon of the action node). By clicking on the calendar icon, the user can select the specific date at which the action has taken place.

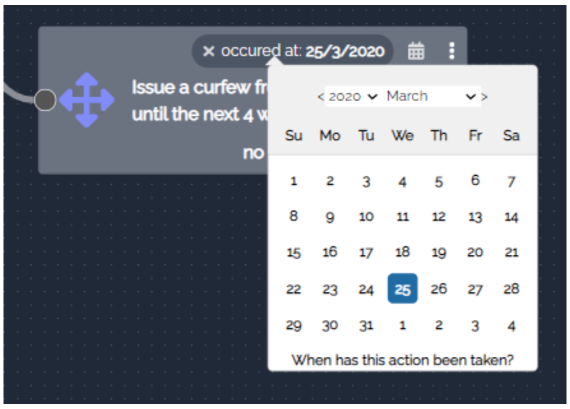


Figure 13: Action node
Source: Own illustration (wickedproblems.io)

The node will display the date next to the calendar (Figure 13), distinguishing it from action nodes without a selected date. The date value will offer great opportunities for many features, an example of which ("snapshots") will be shown later.

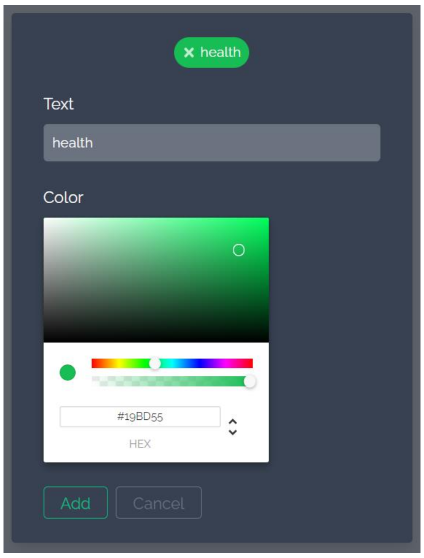


Figure 14: Label editor
Source: Own illustration (wickedproblems.io)

Action dates are useful to assign the actions to a specific time value and distinguish actions that happened from ones that did not. Nevertheless, actions can differ from each other drastically by severity, type, goal, etc. Thus, further distinguishing is needed. This is where labels come into place. Labels can be created and updated in settings (Figure 14). After creating one, the user can click on "select label" on an action node and select one from the drop-down menu. Users must determine their own way to categorize the actions with labels. Like dates, labels will allow implementing further features with actions. Since all nodes emerge from hypothetical contexts, the action node establishes a vivid connection between real-world decisions and the wicked problem visualization.

Furthermore, both idea and action nodes can be up or downvoted by each user in the project. This feature is derived from the literature review, where similar implementations have been found (Janssen et al., 2007) (J. Introne & Iandoli, 2014) (Moraes et al., 2021) (Karacapilidis & Papadias, 2001) (Bothos et al., 2012) (Joshua Introne et al., 2012). Votes visualize the general preference of the stakeholders about an idea or action. An upvote will be displayed to the user by highlighting the up arrow in blue, and a downvote by highlighting the down arrow in red.

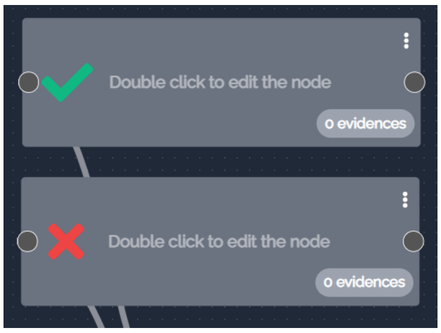


Figure 15: Argument nodes
Source: Own illustration (wickedproblems.io)

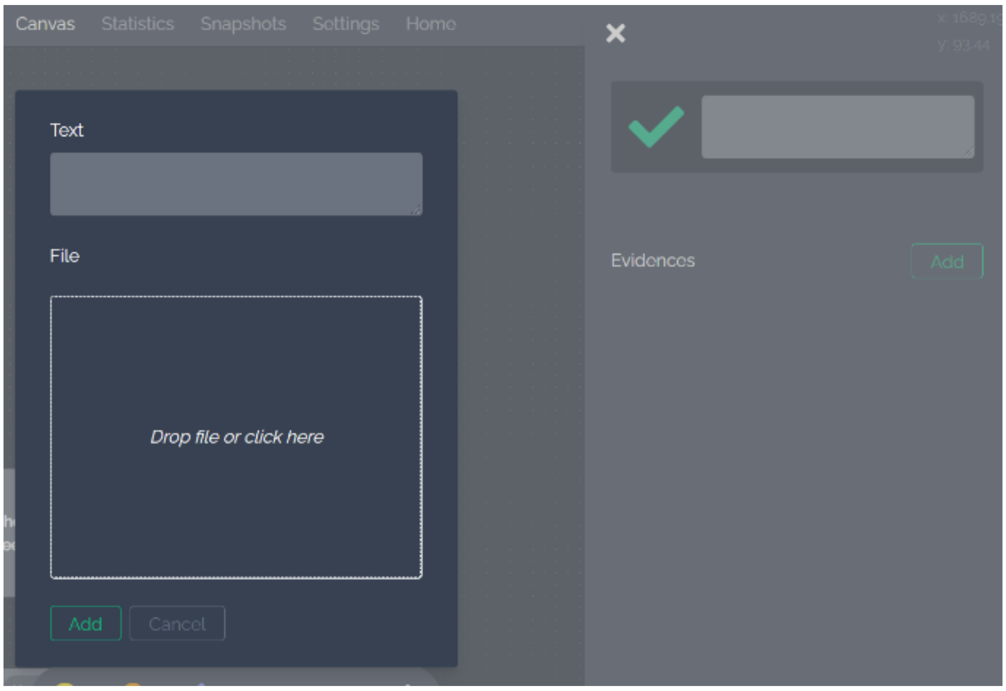


Figure 16: Evidence editor
Source: Own illustration (wickedproblems.io)

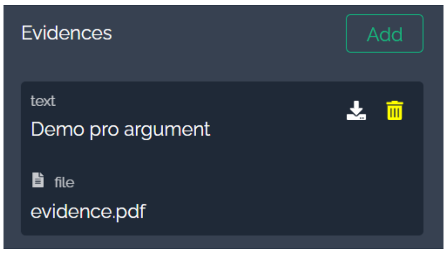


Figure 17: Evidence preview
Source: Own illustration (wickedproblems.io)

Moving on, there are 2 types of argument nodes, pro, and contra (Figure 15), just like in the IBIS notation. Argument nodes can be attached to an idea, as well as an action. Their main purpose is to provide evidence on why an action/idea is good or not. As mentioned before, double-clicking on the nodes will open the node editor. Not only the text of the node can be changed for arguments, but the user can also add evidence (Figure 16), which has 2 properties, a text, and a file. The user can choose either one or both. The file area allows adding any type of file, which can later be downloaded by all users (Figure 17). By clicking on the file name of the file, the application will open a preview in a new tab without having to download it. Since multiple evidence can be added, the user can compile all evidence under one node or add multiple nodes for each one. Each node displays the number of evidence it has (Figure 15). Arguments are helpful to validate an idea or an action between stakeholders. The ability to upload a file of any type simplifies the creation of arguments since users can upload their research, work, or findings without having to explain them in text. Similar evidence features can also be found in the previous literature review (Karacapilidis & Papadias, 2001) (Adamides & Karacapilidis, 2006).

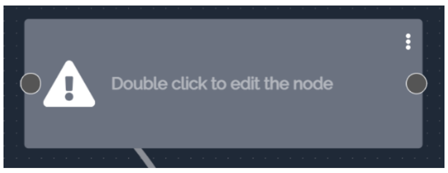


Figure 18: Constraint node
Source: Own illustration (wickedproblems.io)

The last type of node is a constraint (Figure 18), which is not included in the IBIS notation and is similar to the constraint in the Hermes system by Karacapilidis and Papadias (2001). A constraint is simply a reminder for the user about certain guidelines or restrictions to an issue (question). It can also be used as a connection to an idea or action.

Node\Features	Hide branch	Vote	Add date	Add label	Add evidence
Question	X				
Idea		X			
Action		X	X	X	
Argument For					X
Argument Against					X
Constraint					

Table 3: Summary of nodes features
Source: Own Analysis

Many of the nodes implemented in the application are derived from the IBIS notation. The extra nodes and features added enhance the notation and enable the user to extract more value out of the visualization and discussion (Table 3).

Source\Target	Question	Idea	Action	Argument For	Argument Against	Constraint
Question	X	X				X
Idea	X	X	X	X	X	X
Action	X	X	X	X	X	
Argument For	X			X	X	
Argument Against	X			X	X	
Constraint	X	X	X			X

Table 4: Possible nodes connections
Source: Own Analysis

Table 4 shows the possible connections between the nodes. Each node acts as a source and a target, while the target handle is on the left and the source handle on the right. A question node can be connected to another question node, where the target question is usually a specification of the source question. Furthermore, a question can connect to an idea (as a possible solution to the issue) and a constraint.

Idea is the only node that can connect to all other nodes as a source. An action can connect to most nodes except a constraint. Because an action represents something that already happened, it would not matter if there was a constraint to be considered. Both types of arguments can lead to a further question and connect to more arguments that are related to each other.

A question can be created from a constraint that is handled as an issue. In addition, the user can connect an idea as well as actions that may resolve the constraint.

5.3.3 Collaboration

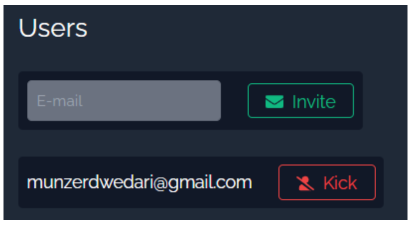
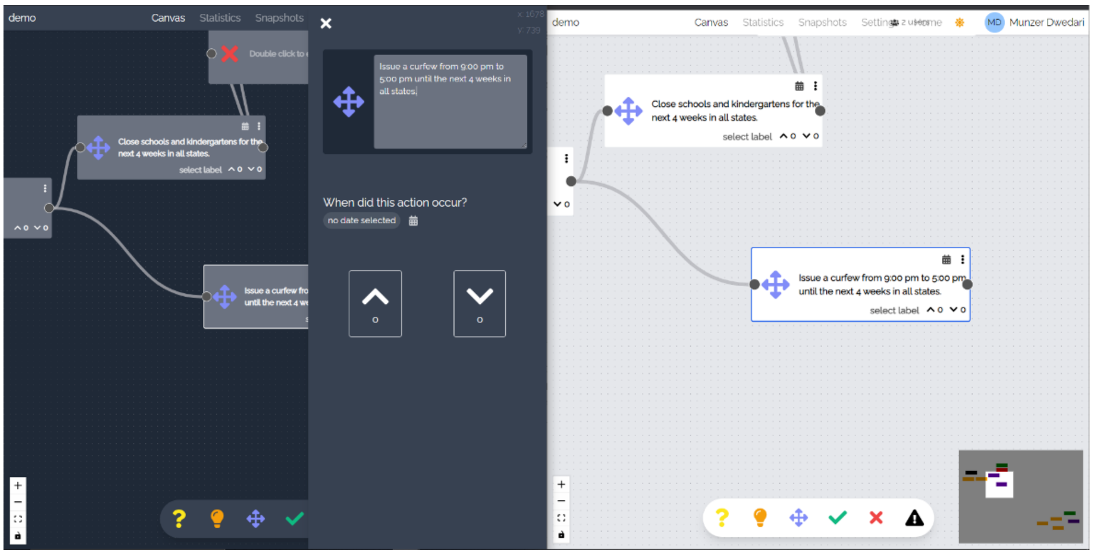


Figure 19: Invitation to project
Source: Own illustration (wickedproblems.io)

One of the especially important features in the developed information system is the collaboration of multiple users in one project. When a user creates a project, they can invite (or kick) other users under settings via their email (Figure 19). The invited user can then see the project on their home page under “Guest projects”. The invitee has access to all functionalities in the project, except inviting other users. The menu on the top will show the number of users in the project, including the admin (owner).

What is crucial about this feature is anonymity (Bothos et al., 2012). Users in a project can only identify other users by viewing the list of emails under settings. Other than that, there is no way to identify who did exactly what in the project. As pointed out in the literature review, it is important to allow users to contribute to an issue anonymously, to prevent shyness or hesitations to bring their ideas to the table. Hence, voting on ideas and actions is anonymous.



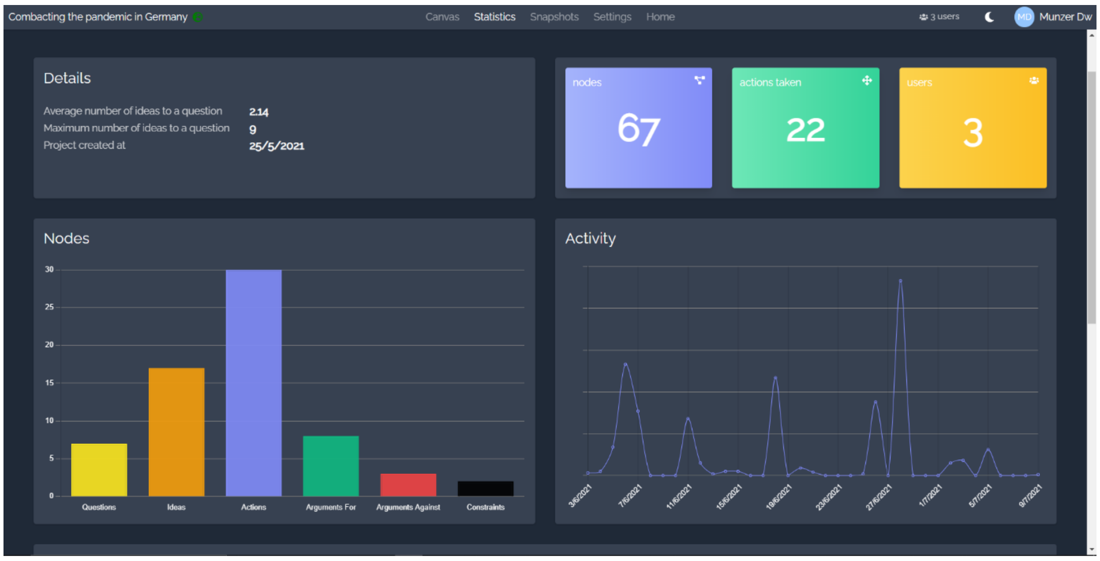
Issue a curfew from 9:00 pm to 5:00 pm until the next 4 weeks in all states

Close schools and kindergartens for the next 4 weeks in all states.

Figure 20: Collaboration view
Source: Own illustration (wickedproblems.io)

Furthermore, the collaboration feature supports real-time participation in the canvas view. When two or more users are working on the canvas, each user should be able to see the changes of others directly without having to reload the page. Therefore, the application will highlight selected nodes by other users with a blue border (Figure 20). Apart from that, when a user changes something in a node, the change will immediately get updated for all other users. This is made possible with socket.io, which will be explained later in further detail. Not only node updates are sent in real-time to all users, but also node locations, meaning when a user moves a node to a new location, it will get updated on the canvas of all other users in that session.

5.3.4 Statistics

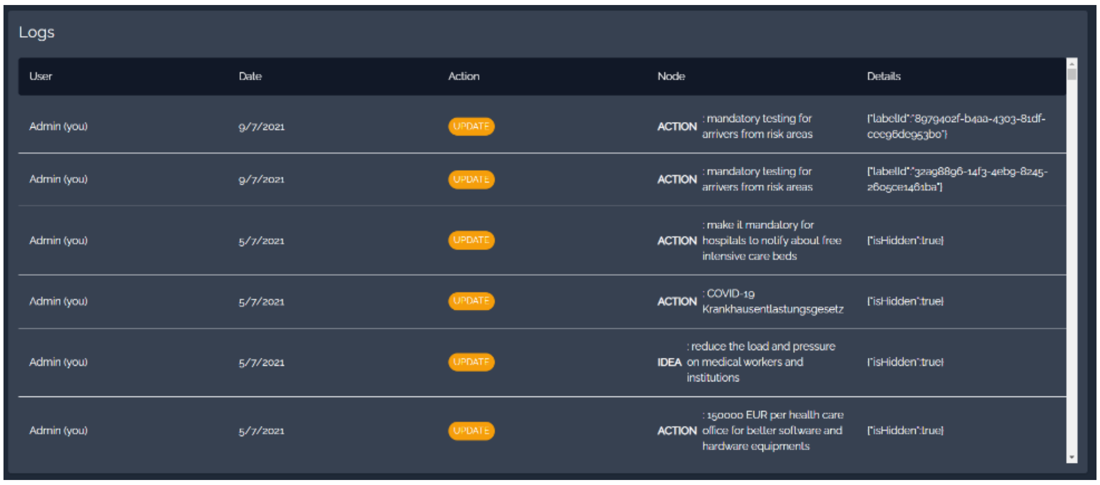


Details

Average number of ideas to a question	2.14
Maximum number of ideas to a question	9
Project created at	25/5/2021

Figure 21: Statistics view
Source: Own illustration (wickedproblems.io)

Statistics enable the user to have a broad overview of the current scope of the wicked problem (Figure 21). It visualizes how big the project has become and provides therefore some relevant key performance indicators (KPIs). On the top left, the user can see the “Average number of ideas to a question”. This can be interpreted as a metric of creativity among the group of users in a project. If the number is low, the user can suggest adding more ideas for issues (questions). On the top right side, the application depicts the number of nodes of the whole project, as well as the number of actions taken. The number of nodes will indicate the size of the project, while the number of actions taken visualizes how much has been done to tackle the wicked problem. The nodes bar chart at the bottom left shows the distribution of the different node types. Here, the user can for instance learn about an imbalance in the number of arguments for and against. Furthermore, the activity graph at the bottom right visualizes whether users have been active on the application and consequently the wicked problem or not. It shows which days the users worked most on the problem, and which days they did not. If the project is fading away slowly, then it will be seen on the activity graph.


Logs

User	Date	Action	Node	Details
Admin (you)	9/7/2021	UPDATE	**ACTION** : mandatory testing for arrivers from risk areas	{"labelId":"8979402f-b4aa-4303-81df-ccc96dc953b0"}
Admin (you)	9/7/2021	UPDATE	**ACTION** : mandatory testing for arrivers from risk areas	{"labelId":"32a9889б-14f3-4eb9-8245-2605ce1461ba"}
Admin (you)	5/7/2021	UPDATE	**ACTION** : make it mandatory for hospitals to notify about free intensive care beds	{"isHidden":true}
Admin (you)	5/7/2021	UPDATE	**ACTION** : COVID-19 Krankhausentlastungsgesetz	{"isHidden":true}
Admin (you)	5/7/2021	UPDATE	**IDEA** : reduce the load and pressure on medical workers and institutions	{"isHidden":true}
Admin (you)	5/7/2021	UPDATE	**ACTION** : 150000 EUR per health care office for better software and hardware equipments	{"isHidden":true}

Figure 22: Logs view
Source: Own illustration (wickedproblems.io)

When scrolling down on the statistics page, the user can find logs, upon which the activity graph is built. The logs show almost every activity that took place in the canvas view and indicates whether that activity is done by the admin or another user (collaborator). It contains the date, the type of activity (create, remove, update, delete), the node type and text, and the values that have been changed. The logs help find out about certain activities that occurred in the project.

To sum up, the statistics provide a visualization about the scope of the project, the distribution of the nodes, and the overall activity by the users.

5.3.5 Snapshots

After developing the canvas view with the IBIS notation, a core feature was still missing for visualizing wicked problems. While the canvas view provides an overview of the discussion flow and the thought process, it lacks two things. First, it does not show the timeline of the actions using the date attribute. Second, it does not show an adequate visualization of the performance of a wicked problem. Several papers found in the literature review mention the necessity of measuring the performance (Pacanowsky, 1995) (The-Australian-Public-Service-Commission, 2018). In our case, the question is how can the performance on a wicked problem be visualized and what are the impacts of our actions. Due to the characteristics of wicked problems, their solutions are not true or false, but rather good or bad. This means that the performance of handling a wicked problem is linked to good and bad outcomes.

Measuring performance should be initiated through the use of data. Data are a truthful source of how a current situation is and offer great value in supporting decision-making. Employing data for performance measuring is important and has been mentioned in past literature (Bibri & Acm, 2019) (Schoder et al., 2014). Bibri and Acm (2019) talk about how data can be utilized to find solutions for wicked problems about Urbanism. In their paper, they mention an example in urban transportation systems where measuring "[..] the time in between buses at each stop, possibly together with the number of passengers waiting, gives the planner the basis for a feedback control solution" (Bibri & Acm, 2019, p. 6). Meaning, by measuring the data, the planner can decide on actions to take to improve the current situation, i.e. waiting

times. The previously mentioned “Climate CoLab” platform (Joshua Introne et al., 2012) uses data models to predict the impacts of a proposed action during a contest. In their research, the authors argue that prediction models offer great support for policymakers, however, most of the time these models and simulations are built by experts, which requires their participation during the process of decision-making. Thus they integrate into their platform prediction graphs about climate change metrics for users to use when proposing actions.
The new feature “snapshots” developed in this thesis, enables data-driven visualization and performance measuring of wicked problems in combination with the action node. Its purpose is to show the positive as well as negative impacts of the past, measure the current performance, define the present situation, and compare the behavior and development of different datasets.

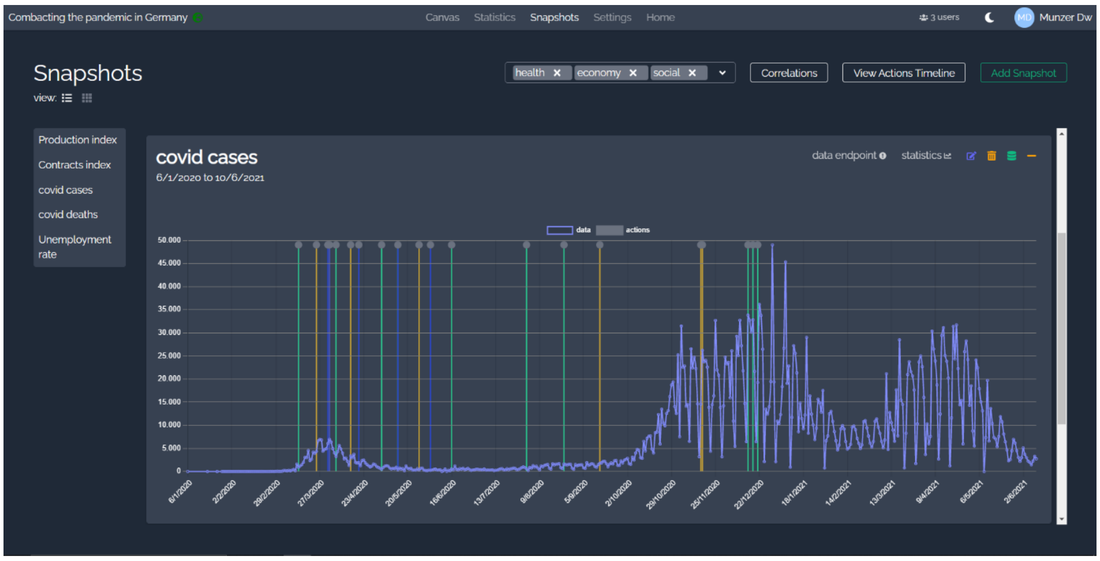


Figure 23: Snapshots view
Source: Own illustration (wickedproblems.io)

A snapshot represents a dataset. Since wicked problems emerge and develop over some time, the type of data in a snapshot is based on a time value – it is a 2-dimensional data set where the x-axis represents a date, and the y axis is a numeric value. The snapshot plots the data into a graph (Figure 23), and adds the actions too as vertical lines with the color of their label. The date feature of the actions is used to link the actions made and the moving course of the data. The connection between data and actions visualizes how the tracked value of the data performs in comparison to the actions. An example to illustrate:

Consider the Corona pandemic as a wicked problem. A key performance indicator of the overall handling of the situation is the number of daily positive cases over time. When the first wave of the pandemic happened in Germany, health officials took necessary actions to condemn the virus. These actions are plotted as vertical lines on the graph (Figure 23). The color of the lines is the same as the label that has been given to the action. The user can observe that the change in the data (positive cases) is impacted by the actions taken at that time. Among these actions is the first lockdown, which restricted social contacts and therefore reduced the number of cases, as can be seen on the graph. The graph shows the user that after taking these actions, the positive Covid cases dropped down, indicating good performance.

Nevertheless, actions can also have negative impacts. This is why the user must add as many different datasets as possible. In this case, the rate of unemployment can be plotted, which will reflect negatively through the impacts of the lockdowns. On the other hand, the number of deaths due to Covid can also be added. The production index, wellness index, national debt, number of small business closures, etc. are all possible data that can be visualized concerning the actions taken.

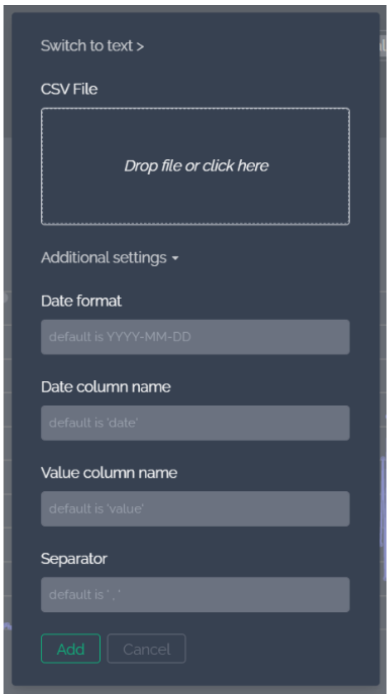
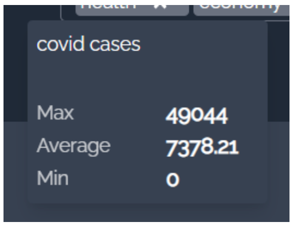


Figure 24: Importing data
Source: Own illustration (wickedproblems.io)

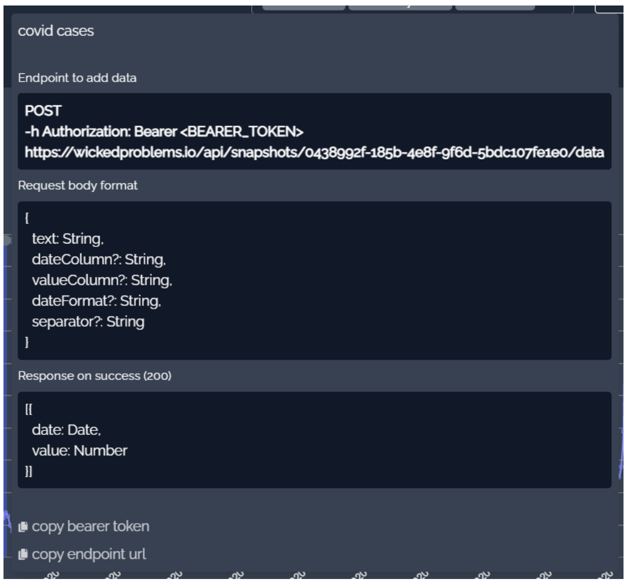
After adding a snapshot, the user clicks on the green database icon to add the data via a CSV file (Figure 24). It is also possible to enter the data as CSV text by clicking "Switch to text" at the top. In the additional settings, the user can enter the date format that is used in the data set. To parse the data correctly, the user adds the date and value column names as well as the separator string. All the additional fields have default values in case the user already fits the dataset to the application. The date value always has to be a day, since the date value of actions is in days too. If the data has values per month, the user can transform the date to the first, middle, or last day of the month. By clicking the orange minus icon, the user deletes the data of a snapshot.


covid cases

Max 49044
Average 7378.21
Min 0

Figure 25: Snapshot statistics component
Source: Own illustration (wickedproblems.io)

The statistics button will show the maximum, average, and minimum values of a dataset (Figure 25).


covid cases

Endpoint to add data

```
POST
-h Authorization: Bearer <BEARER_TOKEN>
https://wickedproblems.io/api/snapshots/0438992f-185b-4e8f-9f6d-5bdc107fe1e0/data
```

Request body format

```
{
  text: String,
  dateColumn?: String,
  valueColumn?: String,
  dateFormat?: String,
  separator?: String
}
```

Response on success (200)

```
[{
  date: Date,
  value: Number
}]
```

Figure 26: Snapshot endpoint view
Source: Own illustration (wickedproblems.io)

Usually, data sets are prepared on data processing platforms by specific programming languages such as Python. To save time on downloading a dataset as a CSV file, and then uploading it, the “data endpoint” button opens a popup and gives the user an endpoint to store the data via a POST request (Figure 26). The popup shows the necessary request format, the input format, and the output data. At the bottom, the user can copy the current bearer token for authentication and the endpoint URL, which holds the id of the snapshot. The post request returns the added data as an array of objects.

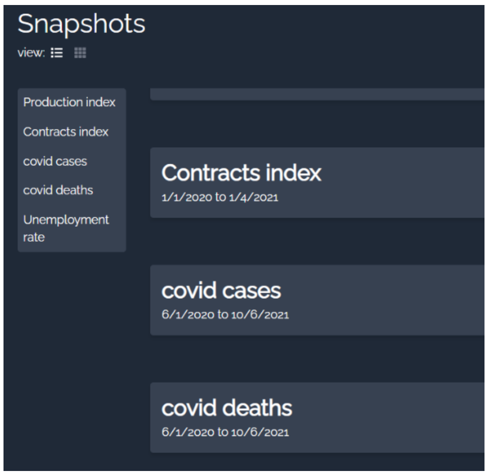


Figure 27: Snapshots navigation
Source: Own illustration (wickedproblems.io)

On the left side of the page, there is a list of the snapshots available, which the user can use for navigation (Figure 27). Evey snapshot graph can be folded open or closed by clicking on the title.

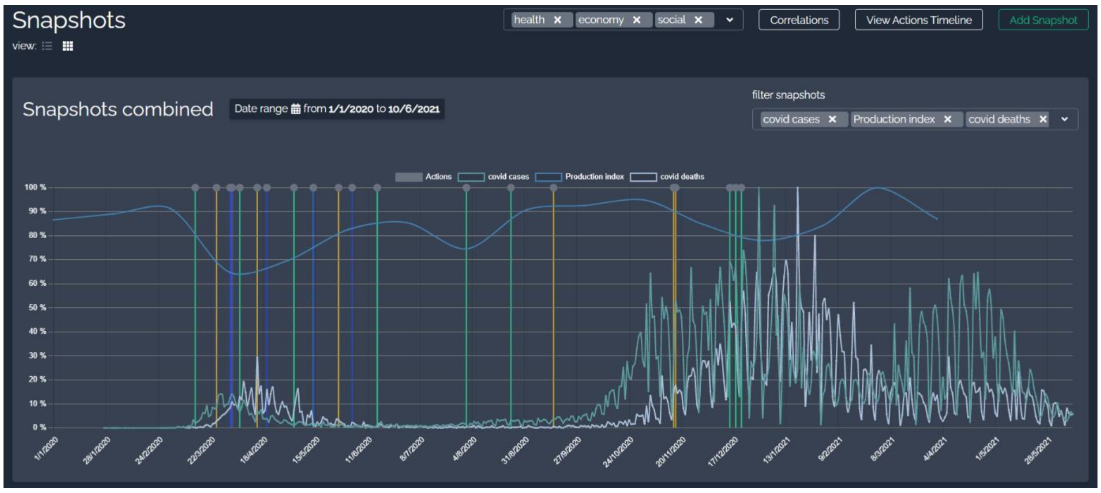
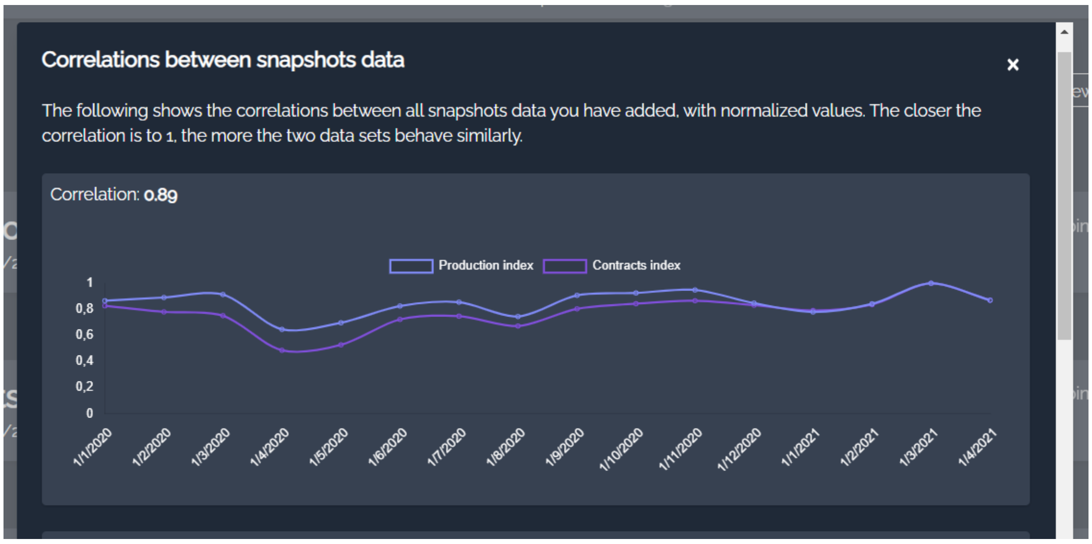


Figure 28: Snapshots combined view
Source: Own illustration (wickedproblems.io)

What about comparing different snapshots together with the actions? The Snapshots combined view provides one graph where the user can choose which snapshots to show through the upper right filter (Figure 28). The application normalizes all the data values to bring them on the same level from 0% to 100%. The normalization simplifies the illustration of the data and enables the comparison of their behavior. The user can also filter the time range as well as the type of actions (labels) on the graph. To get back to the list of snapshots, the user can change the view under the page title in the upper left corner.


Correlations between snapshots data

The following shows the correlations between all snapshots data you have added, with normalized values. The closer the correlation is to 1, the more the two data sets behave similarly.

Correlation: **0.89**

Figure 29: Snapshots correlations view
Source: Own illustration (wickedproblems.io)

Sometimes, the stakeholders of a wicked problem will use a dataset to help identify the next course of action. Thus, they influence the outcome of the data in that particular snapshot. There is a chance that another dataset (snapshot) behaves similarly and is therefore equally influenced by the actions. The correlations view calculates the correlations between all datasets to see whether they behave similarly or not (Figure 29). The correlation is calculated by:

$$R = \frac{\Sigma[(x - \bar{x}) * (y - \bar{y})]}{\sqrt{\Sigma[(x - \bar{x})^2 * (y - \bar{y})^2]}}$$

Equation 1: Correlation equation

where x and y are the values of 2 different data sets. The result will show the datasets with the highest absolute R-value (1) first.

5.4 Implementation

While the features of the application are important to the research question, it is also interesting to look at the implementation design and architecture of the information system. The software consists of 3 components, a frontend client, a backend application programming interface (API), and a PostgreSQL database. The following sections will explain in detail how the architecture is designed.

5.4.1 Tech stack

Component	Libraries/Frameworks
Frontend	React, Mobx, Socket.io, Tailwindcss, Serve
Backend	Express JS, Sequalize, Socket.io
Database	PostgreSQL

Table 5: Tech stack
Source: Own Analysis

The frontend of the application is developed in a NodeJS environment with React as a frontend library. NodeJS "is a JavaScript runtime built on Chrome's V8 JavaScript engine." (nodejs.org). With its asynchronous, event-driven JavaScript runtime, it enables programmers to build and scale network applications. Furthermore, React (reactjs.org) is a JavaScript library for building user interfaces for web as well as mobile applications. It is developed by Facebook and has a declarative architecture where views consist of components that render based on a defined state. After developing the UI, react builds all the views and logic into static Html, CSS, and JavaScript files, which are then served over the web by the "serve" library. Moreover, the state management is implemented with Mobx (mobx.js.org), which has support for React. Mobx is a simple well-supported library, where state management can be completely done separately from the UI. The styling of the application is done completely with Tailwindcss, providing great flexibility and support for different themes (like a dark and light theme).

Moving on to the backend, which is also running on NodeJs, ExpressJS is a popular framework that enables the creation of NodeJS servers based on HTTP protocols. Sequalize is a library that communicates with an SQL database without having to write SQL queries. It can be used to define database models as well as associations and use them to simply call methods instead of writing complicated queries. Socket.io is used for "real-time, bidirectional and event-driven communication" (socket.io) between the frontend and backend. It is employed to enable the multiuser feature, where users get updates from other participants in real-time in the canvas view.

5.4.2 Analysis Object Model

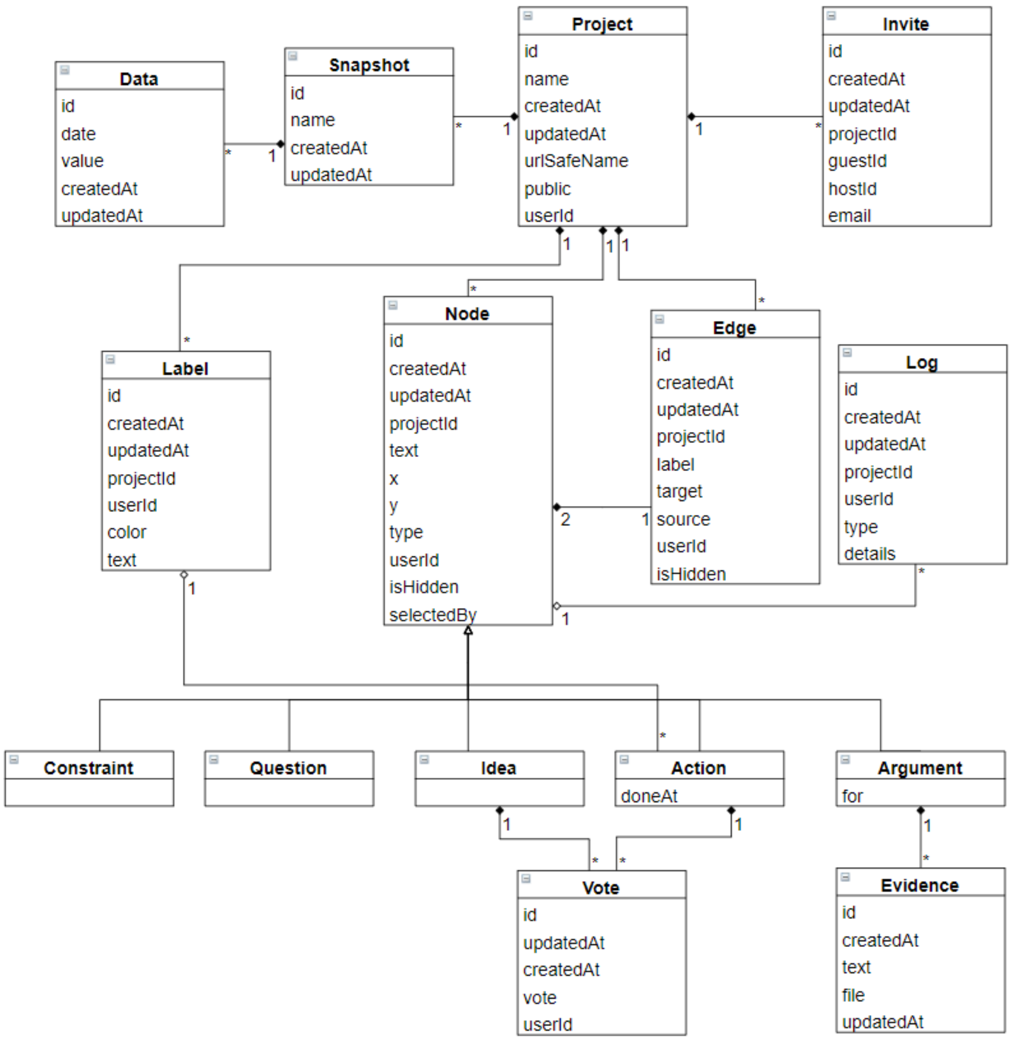


Figure 30: Analysis object model (excluding class methods)
Source: Own illustration

The analysis object model represents the structure of the application and is used to map the frontend UI and backend architecture as well as the database (Figure 30).
To begin with, the project object has a unique id and two date attributes (createdAt, and updatedAt) that are maintained automatically by the database. The three attributes are found in all other object types too. A project is assigned to an individual user (owner) by the "userId" and has a simple name attribute. The name is automatically stored as a url-safe name, which is used to provide a human-readable link to the project (instead of having to use the id for instance).

Next is the node interface. Many nodes belong to one project, as can be seen by the composition line. Every type of node has a text, position x & y, a type, and a Boolean attribute to determine if it is hidden on the canvas or not. To show which user is currently working on the node, the "selectedBy" attribute is used.

2 nodes are connected through one edge, one as a source, one as a target. The edge object has a label, which is set as an empty string by default, and a Boolean attribute "isHidden" too. The connection between the Edge object and project is the same as the node.

The different node object types are derived from the interface. Questions and constraints do not provide an extra attribute to the interface, they only contain simple text. Both ideas and actions are connected to many votes. A vote object has a "userId" attribute of the voter, and the actual vote which has three values: "true" for an upvote, "false" for a downvote, and null for no vote. The action node has the extra date attribute "doneAt". The argument has the extra attribute "for" which is a Boolean value. True represents an argument for and false an argument against. The argument node has multiple pieces of evidence, which contain both a text and/or a file.

Furthermore, the log object represents a change in the node, ignoring the x & y and "isHidden" attributes, since they do not provide important information. One node can have many logs. A log has a type that can be "create", "update" or "delete". The details attribute contains the changed attributes of a node in JSON format, while "userId" in this case is not the owner of the project, but the user who committed an update on the node, which resulted in a log entry.

Apart from that, labels are used to differentiate action nodes. Every project has many labels. A label has a text, a color, and can be connected to many actions. Moreover, an invite connects a project and its owner to a guest user. When a guest user tries to access the project data, the invite object will be used to determine whether the user is allowed to have access or not. Finally, a snapshot is an object type with simply a name. Many snapshots can connect to a project and the data object model represents one data value that can be seen on a graph with the date on the x-axis and the actual value on the y-axis.

5.4.3 Database

The database structure is similar to the Analysis Object model (Figure 30). In the database, the "Node" interface is represented in one table with all the attributes combined. While there are multiple ways to integrate inheritance into the database, "single table inheritance" is one of the easiest. The drawbacks of this method are:

1. Enforcing "not null" on attributes is not possible and will have to be done by the application logic since one database entry contains attributes of all types.
2. If a new type is going to be added, the table format will have to be altered, which can create some complications.

These drawbacks have been considered and concluded to be insignificant. None of the node types has a unique attribute that must not be null, and for now, no new node types are planned to be added. More node types might also make the usability complicated. Apart from that, having all node types in one table will enable switching node types easily, and the data for all types is persisted, no matter what the type attribute changes to.

Furthermore, the foreign key relationships to projects (attribute "projectId") have "cascade" on delete, meaning all the entities that are associated with a project will be deleted if the project is deleted. The same kind of relationship is also found in "Votes" & "Evidence" to

"Nodes" and "Labels" to "Nodes", and "Edges" to "Nodes", where edges are automatically deleted if one of the nodes does not exist anymore. Snapshot data are also deleted automatically when a snapshot is removed from the database.

5.4.4 Authentication

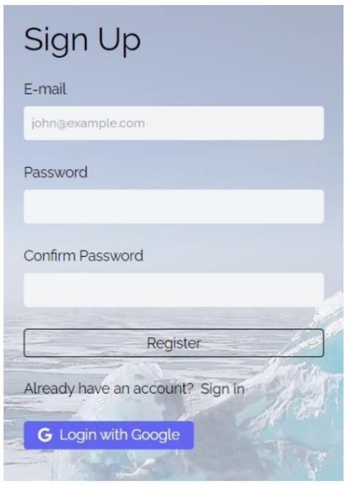


Figure 31: Signup view
Source: Own illustration (wickedproblems.io)

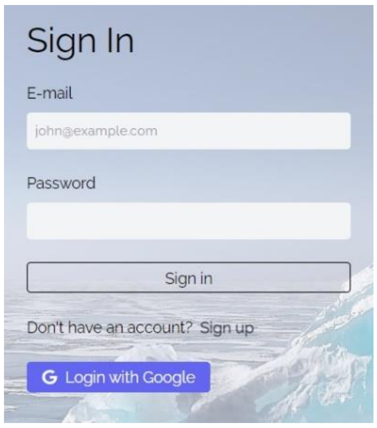
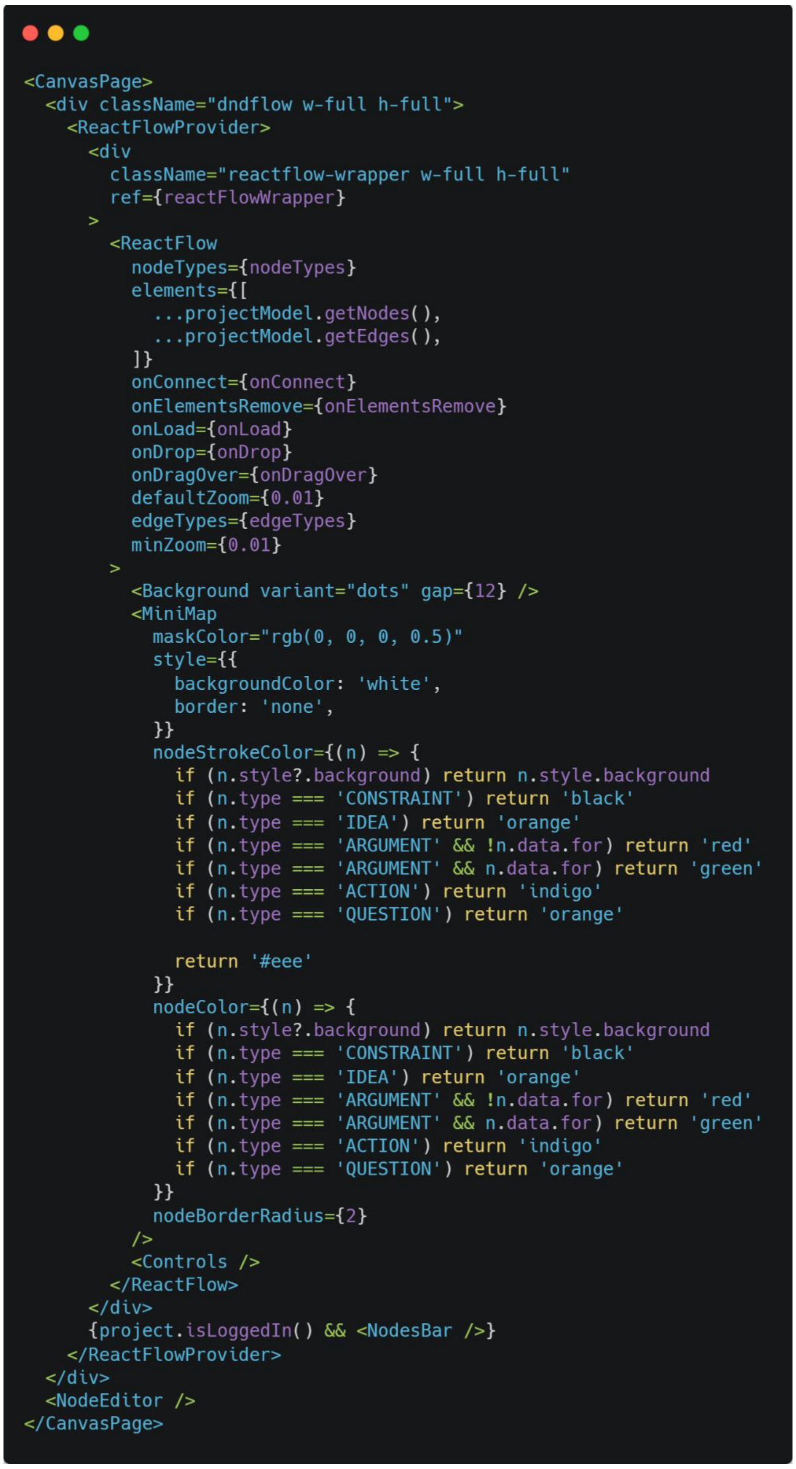


Figure 32: Sign in view
Source: Own illustration (wickedproblems.io)

To store private data, enable collaboration, and improve usability, authentication is crucial for the information system. Therefore, Firebase authentication is implemented. Firebase is a platform created by Google for mobile and web applications. One of its services is authentication. What is special about it, is that users can log in with their accounts of popular services like GitHub, Facebook, Microsoft, Twitter, Google, and more (for this project, only Google is enabled) (Figure 31, Figure 32). Apart from that, Firebase has the option to sign up with an email and password (Figure 31) and takes care of authentication workflows such as email verification. In addition, the Firebase console provides a great management dashboard for authentication settings and other features such as analytics. The integration of the application into Firebase was simple thanks to the available libraries for Nodejs.

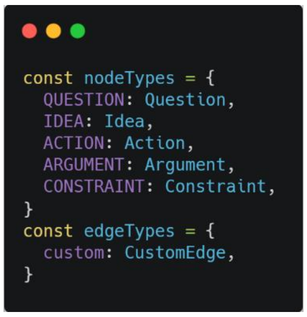
On the client's side (frontend), Firebase provides a method for users to call with their credentials to log in. The method returns authentication data with a bearer access token. The token is then stored in the application to use for all requests. Firebase stores authentication in the browser, so users are kept logged in. In addition to the client-side, firebase provides functionality to write middleware to authenticate incoming requests. The Firebase middleware uses the incoming bearer token from each request and returns a user object to the route resolver. The user object contains the user id, which is used to associate data with a user account. None of the user data is stored in the PostgreSQL database. All the authentication data is stored in the Firebase cloud.

5.4.5 Reactflow



```
<CanvasPage>
  <div className="dndflow w-full h-full">
    <ReactFlowProvider>
      <div
        className="reactflow-wrapper w-full h-full"
        ref={reactFlowWrapper}
      >
        <ReactFlow
          nodeTypes={nodeTypes}
          elements={[
            ...projectModel.getNodes(),
            ...projectModel.getEdges(),
          ]}
          onConnect={onConnect}
          onElementsRemove={onElementsRemove}
          onLoad={onLoad}
          onDrop={onDrop}
          onDragOver={onDragOver}
          defaultZoom={0.01}
          edgeTypes={edgeTypes}
          minZoom={0.01}
        >
          <Background variant="dots" gap={12} />
          <MiniMap
            maskColor="rgb(0, 0, 0, 0.5)"
            style={{
              backgroundColor: 'white',
              border: 'none',
            }}
            nodeStrokeColor={(n) => {
              if (n.style?.background) return n.style.background
              if (n.type === 'CONSTRAINT') return 'black'
              if (n.type === 'IDEA') return 'orange'
              if (n.type === 'ARGUMENT' && !n.data.for) return 'red'
              if (n.type === 'ARGUMENT' && n.data.for) return 'green'
              if (n.type === 'ACTION') return 'indigo'
              if (n.type === 'QUESTION') return 'orange'

              return '#eee'
            }}
            nodeColor={(n) => {
              if (n.style?.background) return n.style.background
              if (n.type === 'CONSTRAINT') return 'black'
              if (n.type === 'IDEA') return 'orange'
              if (n.type === 'ARGUMENT' && !n.data.for) return 'red'
              if (n.type === 'ARGUMENT' && n.data.for) return 'green'
              if (n.type === 'ACTION') return 'indigo'
              if (n.type === 'QUESTION') return 'orange'
            }}
            nodeBorderRadius={2}
          />
          <Controls />
        </ReactFlow>
      </div>
      {project.isLoggedIn() && <NodesBar />}
    </ReactFlowProvider>
  </div>
  <NodeEditor />
</CanvasPage>
```

Figure 33: Canvas view frontend code snippet
Source: Own illustration
Image created with: carbon.now.sh



```
const nodeTypes = {
  QUESTION: Question,
  IDEA: Idea,
  ACTION: Action,
  ARGUMENT: Argument,
  CONSTRAINT: Constraint,
}
const edgeTypes = {
  custom: CustomEdge,
}
```

Figure 34: Frontend objects types for nodes & edges for Reactflow
Source: Own illustration
Image created with: carbon.now.sh

The canvas view is based on an open-source library called "React Flow" (reactflow.dev). The library is developed by webkid.io, an agency based in Germany that deals with interactive news and data visualization. The library can be used to create flow chart views with custom nodes. Figure 33 shows the code of the canvas page view developed with the library in React. The "ReactFlowProvider" allows access to the internal state and actions of the library outside the flow chart component "ReactFlow". Here, the first property that "ReactFlow" takes is an object which maps the node types to their specific React components (Figure 34). These node components get properties like position and data from "ReactFlow" and implement their style for their node type. Likewise, it also accepts a custom edge type with custom styling. The elements of the canvas are provided through an array of objects, where nodes and edges are fetched by the project state model and restructured according to the "ReactFlow" interface (more on the state models later). Moreover, the library accepts callback functions to handle connections between nodes ("onConnect"), removing nodes or edges ("onElementsRemove"), and adding new nodes ("onDrop").

Furthermore, the React Flow library provides components and more customizations to the canvas view. The "Background" component shows the background dots, which can be changed to a different pattern or have different gap sizes in-between. In addition, the minimap on the bottom right is also provided in a separate component. Here, the style, colors, as well as representation of the nodes, can be customized. Last but not least is the component for the controls, which in this case is being used with its default properties.

5.4.6 State Management

State management is a crucial part of developing the information system, especially for performance. In React, components/views are bound to a state. When the state changes, React re-renders the components/views. The re-render process increases the RAM usage of the application, so the bigger the render, the more computing power is required by the browser. In this case of the information system, the canvas view can grow endlessly, meaning the view can get excessively big and therefore the re-renders as well. To overcome heavy re-renders, the state management is designed to minimize re-renders as much as possible. For example, every node is connected to its state, and a change in that state will not trigger a re-render of the whole canvas view, but the node only. When moving a node around, only the state of that node is changed. Similarly, when editing a node, adding evidence, voting, etc., only the node's state is changed. The canvas view is only re-rendered when the user adds, connects, or deletes nodes/edges.

After searching for a state management library for React, the decision was made to use Mobx. Mobx can be implemented in a truly clear model-view-architecture. Let us use the user's home page for example.

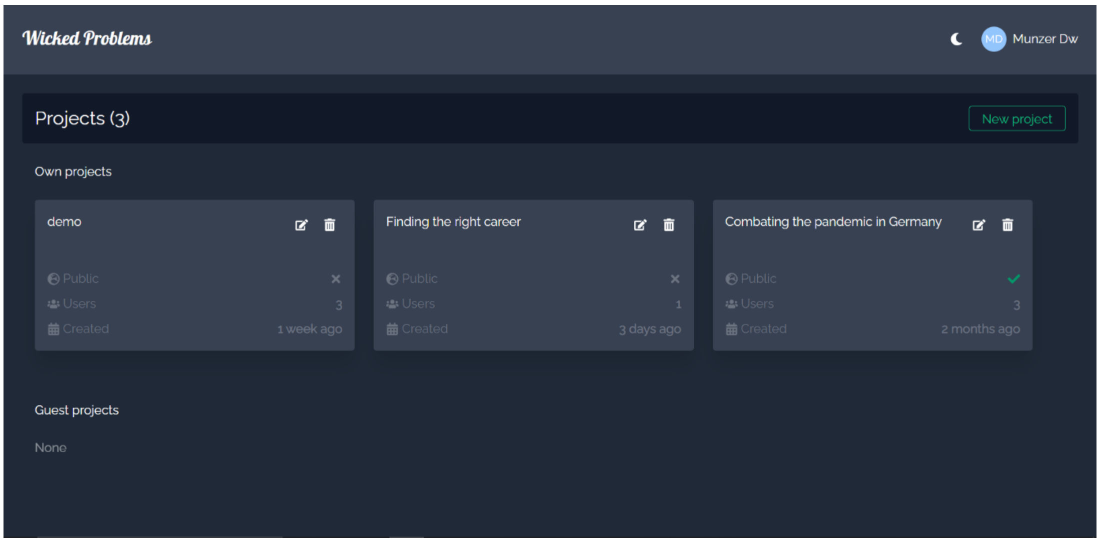


Figure 35: User's home page
Source: Own illustration (wickedproblems.io)

The home page UI lists the projects associated with the user in a horizontal list view. Each project is represented in a project card component (Figure 35). Starting with the model, the class Projects represents the state of the view. It is set as an observable object for Mobx in the constructor. The class has the array field "projects" and provides state methods that are used to interact with the state, while the API methods are used for communication with the backend service. In the end, an instance of the model class is initiated and exported for components to use.

The "Projects" React component acts as a controller and a view. It initiates a call of the API function of the model on the first render of the page to fetch the projects. The whole view of the projects page is wrapped in an "observer", which is provided by Mobx. The observer component listens to state changes by the model and re-renders accordingly. The button "Add new project" for instance acts as a controller for the project editor model, which is part of the input form to create or edit a project. The state model attribute "projects" is mapped into the project card components, which only receives an id of the project and gets the project data from the model through the "findProject" method. Similar to the nodes, when a project object is updated, the project's view does not re-render, only the project card component.

5.4.7 Socket.io

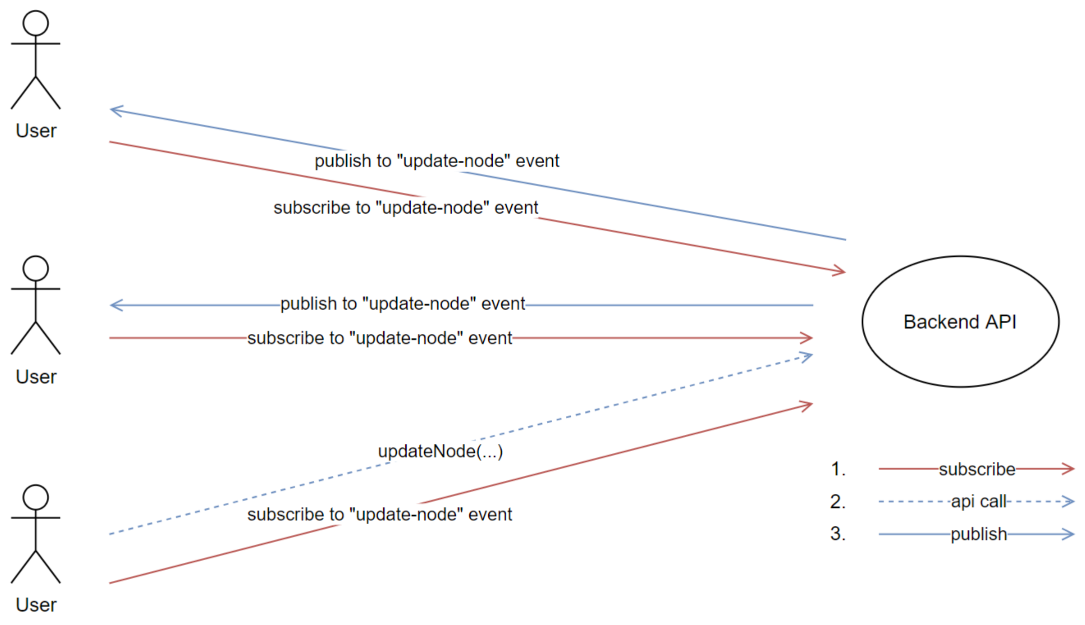
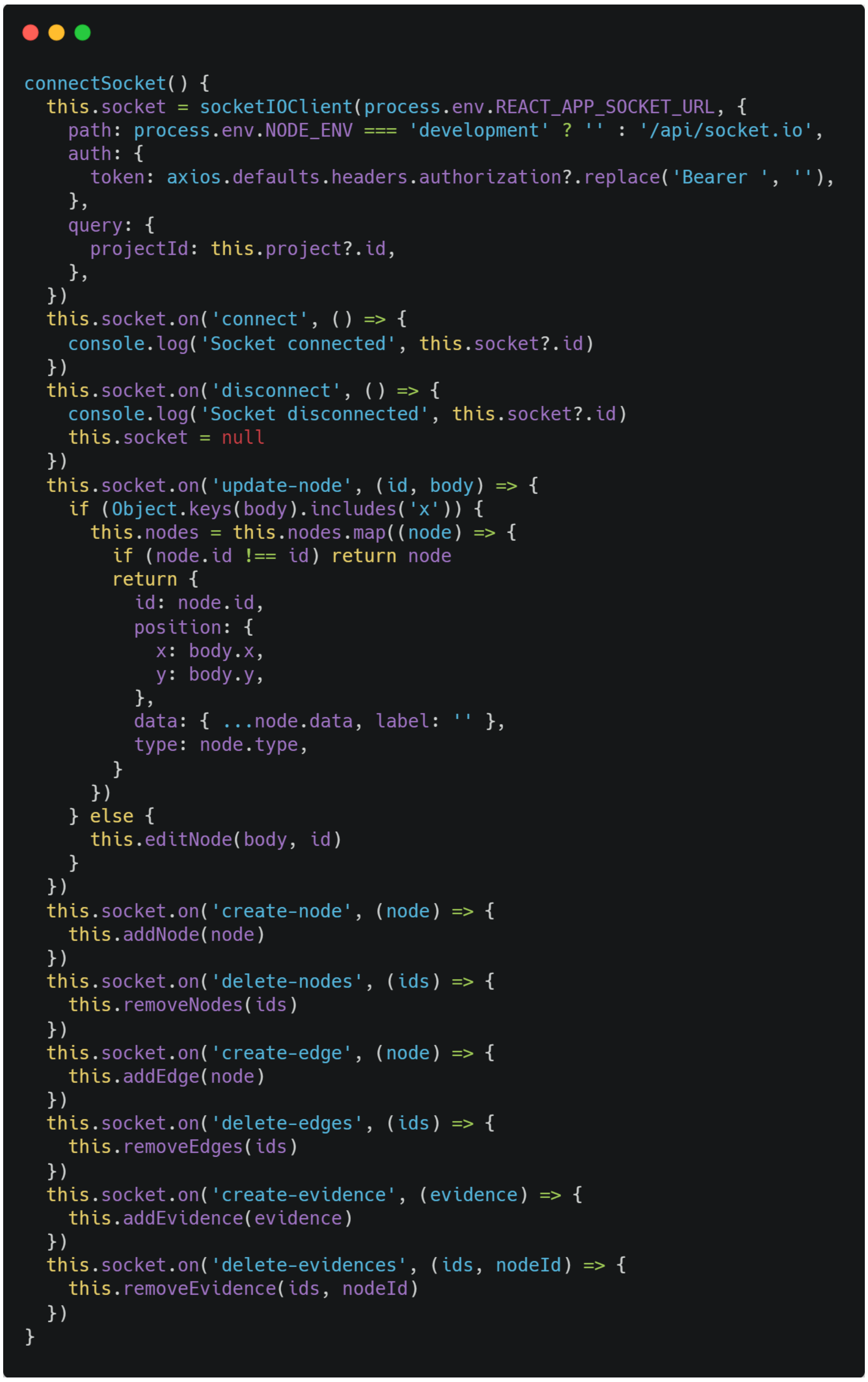

User
publish to "update-node" event
subscribe to "update-node" event
User
publish to "update-node" event
subscribe to "update-node" event
Backend API
updateNode(...)
subscribe to "update-node" event
User
1. subscribe
2. api call
3. publish

Figure 36: Backend socket.io design
Source: Own illustration

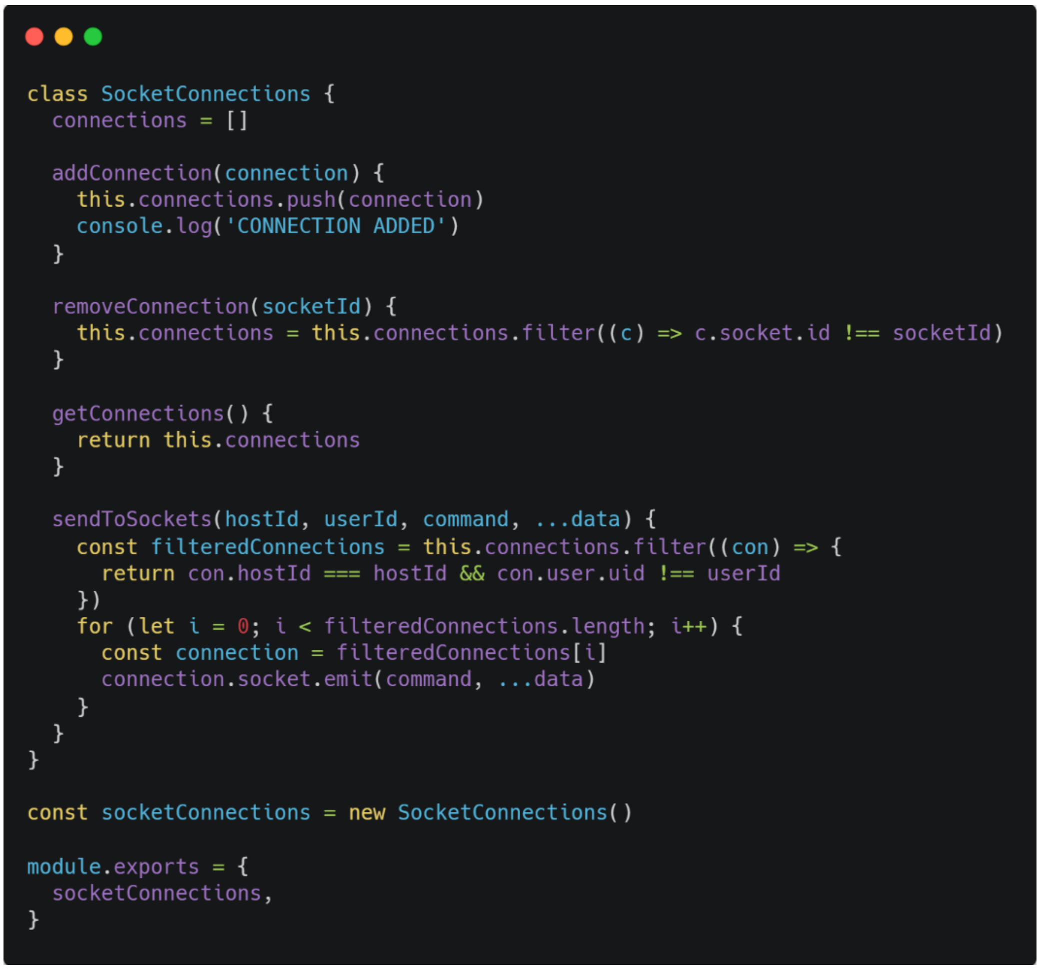
Socket.io has enabled implementing collaboration in real-time. This means when a user opens the canvas page of a project, the application frontend client will automatically establish a connection to the backend API and listen to specific events (Figure 36).



```
connectSocket() {
  this.socket = socketIOClient(process.env.REACT_APP_SOCKET_URL, {
    path: process.env.NODE_ENV === 'development' ? '' : '/api/socket.io',
    auth: {
      token: axios.defaults.headers.authorization?.replace('Bearer ', ''),
    },
    query: {
      projectId: this.project?.id,
    },
  })
  this.socket.on('connect', () => {
    console.log('Socket connected', this.socket?.id)
  })
  this.socket.on('disconnect', () => {
    console.log('Socket disconnected', this.socket?.id)
    this.socket = null
  })
  this.socket.on('update-node', (id, body) => {
    if (Object.keys(body).includes('x')) {
      this.nodes = this.nodes.map((node) => {
        if (node.id !== id) return node
        return {
          id: node.id,
          position: {
            x: body.x,
            y: body.y,
          },
          data: { ...node.data, label: '' },
          type: node.type,
        }
      })
    } else {
      this.editNode(body, id)
    }
  })
  this.socket.on('create-node', (node) => {
    this.addNode(node)
  })
  this.socket.on('delete-nodes', (ids) => {
    this.removeNodes(ids)
  })
  this.socket.on('create-edge', (node) => {
    this.addEdge(node)
  })
  this.socket.on('delete-edges', (ids) => {
    this.removeEdges(ids)
  })
  this.socket.on('create-evidence', (evidence) => {
    this.addEvidence(evidence)
  })
  this.socket.on('delete-evidences', (ids, nodeId) => {
    this.removeEvidence(ids, nodeId)
  })
}
```

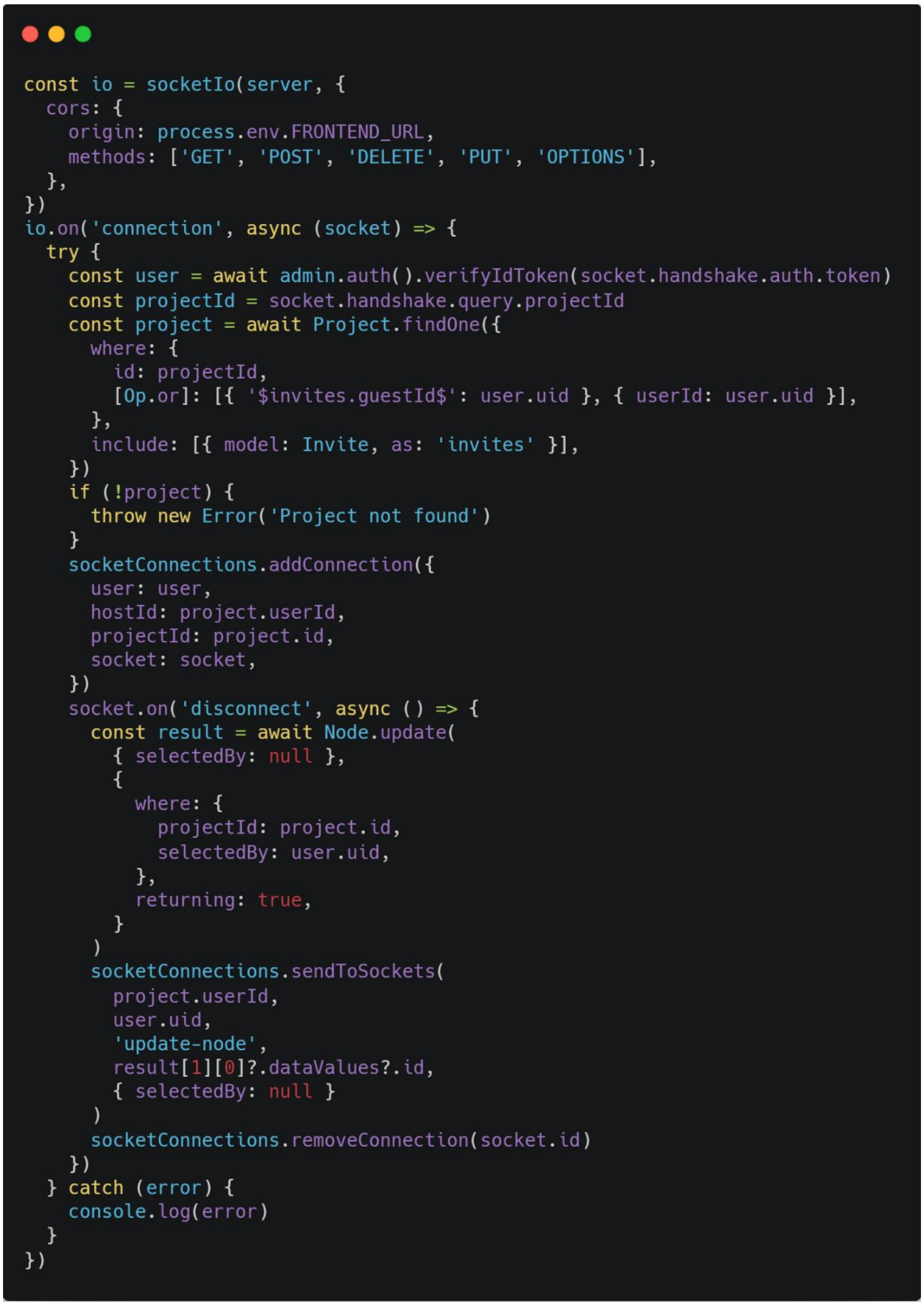
Figure 37: Frontend socket.io connect method
Source: Own illustration
Image created with: carbon.now.sh

The connect function is a class method of the Project state model, and the connection is saved as a class attribute (Figure 37). As can be seen, the connection is authenticated with the Bearer token of Firebase. The project ID is also sent as a query parameter to identify which project the user wants to subscribe to. After establishing the connection, multiple event callbacks are implemented.



```
class SocketConnections {
  connections = []

  addConnection(connection) {
    this.connections.push(connection)
    console.log('CONNECTION ADDED')
  }

  removeConnection(socketId) {
    this.connections = this.connections.filter((c) => c.socket.id !== socketId)
  }

  getConnections() {
    return this.connections
  }

  sendToSockets(hostId, userId, command, ...data) {
    const filteredConnections = this.connections.filter((con) => {
      return con.hostId === hostId && con.user.uid !== userId
    })
    for (let i = 0; i < filteredConnections.length; i++) {
      const connection = filteredConnections[i]
      connection.socket.emit(command, ...data)
    }
  }
}

const socketConnections = new SocketConnections()

module.exports = {
  socketConnections,
}
```

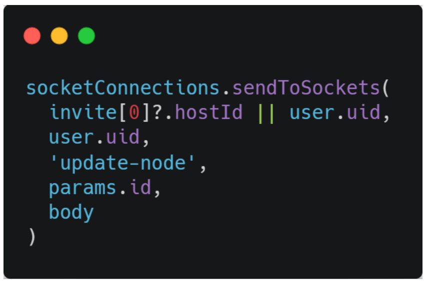
Figure 38: Backend “SocketConnections” class
Source: Own illustration
Image created with: carbon.now.sh



```javascript
const io = socketIo(server, {
  cors: {
    origin: process.env.FRONTEND_URL,
    methods: ['GET', 'POST', 'DELETE', 'PUT', 'OPTIONS'],
  },
})
io.on('connection', async (socket) => {
  try {
    const user = await admin.auth().verifyIdToken(socket.handshake.auth.token)
    const projectId = socket.handshake.query.projectId
    const project = await Project.findOne({
      where: {
        id: projectId,
        [Op.or]: [{ '$invites.guestId$': user.uid }, { userId: user.uid }],
      },
      include: [{ model: Invite, as: 'invites' }],
    })
    if (!project) {
      throw new Error('Project not found')
    }
    socketConnections.addConnection({
      user: user,
      hostId: project.userId,
      projectId: project.id,
      socket: socket,
    })
    socket.on('disconnect', async () => {
      const result = await Node.update(
        { selectedBy: null },
        {
          where: {
            projectId: project.id,
            selectedBy: user.uid,
          },
          returning: true,
        }
      )
      socketConnections.sendToSockets(
        project.userId,
        user.uid,
        'update-node',
        result[1][0]?.dataValues?.id,
        { selectedBy: null }
      )
      socketConnections.removeConnection(socket.id)
    })
  } catch (error) {
    console.log(error)
  }
})
```

Figure 39: Backend socket.io instance and connection handler
Source: Own illustration
Image created with: carbon.now.sh

On the backend, there is the “SocketConnections” class (Figure 38), which stores connection objects and offers some methods on the array of connections. Before accepting connections, a Socket.io instance is initiated with CORS options to increase security and allow connections from the domain origin only (Figure 39). The “connection” callback first verifies the user via the sent authentication token and checks whether there is a project where the user is either the owner or a guest. If the project exists and the user is verified, the connection is stored in the “SocketConnections” instance. A connection contains the user who created the connection, the user ID of the project owner (“hostId”), the project ID, and the socket object.



```
socketConnections.sendToSockets(
  invite[0]?.hostId || user.uid,
  user.uid,
  'update-node',
  params.id,
  body
)
```

Figure 40: Backend sending node update to socket connections
Source: Own illustration
Image created with: https://carbon.now.sh/

Now, when another user, who is also connected, updates the text of a node, for instance, the backend API route handler will send the updated values to all listeners of the event “update-node” (Figure 40). The arguments are the owner ID of the project, the user ID who initiated the update, the event name, the ID of the node, and the changed attributes (body). The “SocketConnections” class filters the connections where the host ID of the project is the same, but the connected user has a different ID than the user who initiated the node update. After that, the event is published to the filtered sockets with the necessary data (node ID, and body). The frontend client will then update the state of the node with the new values. With Socket.io, there are many possibilities to make collaboration easier and improve usability to the user.

5.4.8 Hosting

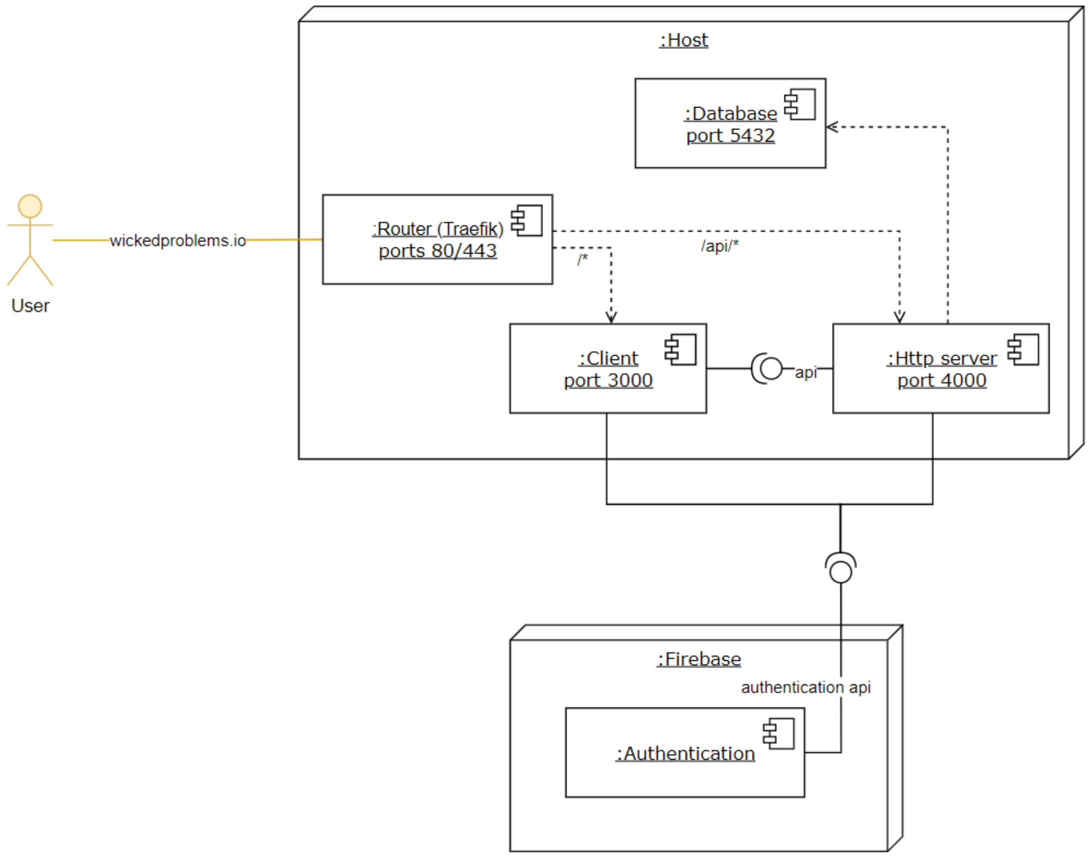


Figure 41: (UML) Combination of deployment and component diagram
Source: Own illustration

The hosting of the application is straightforward. There are 4 components: frontend client, HTTP server, database, and proxy router (Traefik), all of which are running on one host (Figure 41). Every component is running in its own docker environment, separate from the host machine and other components. Docker exposes one port for each component to communicate with the other ones. The client is running on port 3000, the HTTP server on 4000, and the Postgres database on 5432. The Traefik container accepts connections on default HTTP(s) ports on 80 and 443 from the domain "wickedproblems.io".

The HTTP service component provides the API of the application via HTTP requests. This is used by the client, or directly by the user. As mentioned before, Firebase is connected for authentication in the application. It enables the client component to log in with an account and generate an access token. On the other hand, it enables the HTTP server to authenticate incoming requests and get user data. Firebase authentication is running on a separate server.

Furthermore, the entry points for the user are ports 80 (HTTP) and 443 (HTTPS). The proxy router Traefik (traefik.io) here plays a significant role. It runs on a docker container too and redirects the incoming requests from the two HTTP(s) ports through the domain "wickedproblems.io” to the client docker container on port 3000. Likewise, it redirects requests with path-name prefix “/api” to the HTTP server docker container on port 4000. Traefik also enables setting up a free SSL certificate with “let’s encrypt” (letsencrypt.org).

5.5 Demonstration

Now it is time to demonstrate the application on a wicked problem. Therefore, we can use the Corona pandemic that is already mentioned multiple times before. The goal of the demonstration is to show how the wicked problem can be visualized. Schiefloe (2021) explains how the Corona pandemic matches with the characteristics of wicked problems that have been defined by Rittel and Webber (1973). Apart from the necessary characteristics, the Corona pandemic is an especially good example to illustrate the application because it is a new problem (since 2020) and thus a lot of what has been done can be traced back to the beginning and found online. Furthermore, the Corona pandemic is also still active as of the time of writing, and because it has a great impact on society, it would be very interesting to see how it can be plotted into the canvas view and snapshots. To narrow down the problem and make it easier to research information, the demonstration will only focus on the handling of the government in Germany. The topic of the project is “Combating the Corona pandemic in Germany” and can be viewed as a public project on the home page of the application (wickedproblems.io).

To make the demonstration as accurate as possible, we can look at the websites of the government and extract data and blog posts on the pandemic. Luckily, the government has published a PDF document containing all the actions and goals they set to condemn the pandemic (German-Federal-Government, 2020). The document is published at the beginning of the pandemic and is structured into 3 objectives:

Objectives	
1	Protecting health and safeguarding the effectiveness of our healthcare system
2	Cushioning the impact on citizens, employees and companies
3	Overcoming the pandemic through international cooperation

Table 6: Objectives to condemn the pandemic
Source: German-Federal-Government (2020)

For the sake of demonstration, only the first objective is further investigated and is plotted in the canvas view as a question:

“How do we protect the health and safeguard the effectiveness of our healthcare system?”.

Furthermore, under each objective, the document lists multiple points to specific topics that were considered. These topics can be plotted as further questions (issues) that branch from the main question (objective). Example:

	Topic	Question
1	Strengthening the healthcare system	How can we strengthen the healthcare system?
2	Slowing down the rate of infection	How do we slow down the rate of infection?

Table 7: Topics related to the objective
Source: German-Federal-Government (2020)

After that, the text is analyzed, and ideas are extracted and illustrated as nodes in the canvas view. Some actions are derived from the document, however, since the date is very early, many actions that followed during the year 2020 are not included. Because the objective that is demonstrated is related to health, one can look at the website of the "Federal Ministry of Health" in Germany. There, daily news about the Corona pandemic is published in German and translated to English for the demonstration (Federal-Ministry-Of-Health, 2020). The ministry communicates information about the virus and the actions that are taken to condemn it. Since the information is published daily, it is very easy to extract actions and assign dates to them.

Using the document from the beginning of the pandemic and the website of the ministry of health, we can plot the "discussion" and planning process that took place in the year 2020.

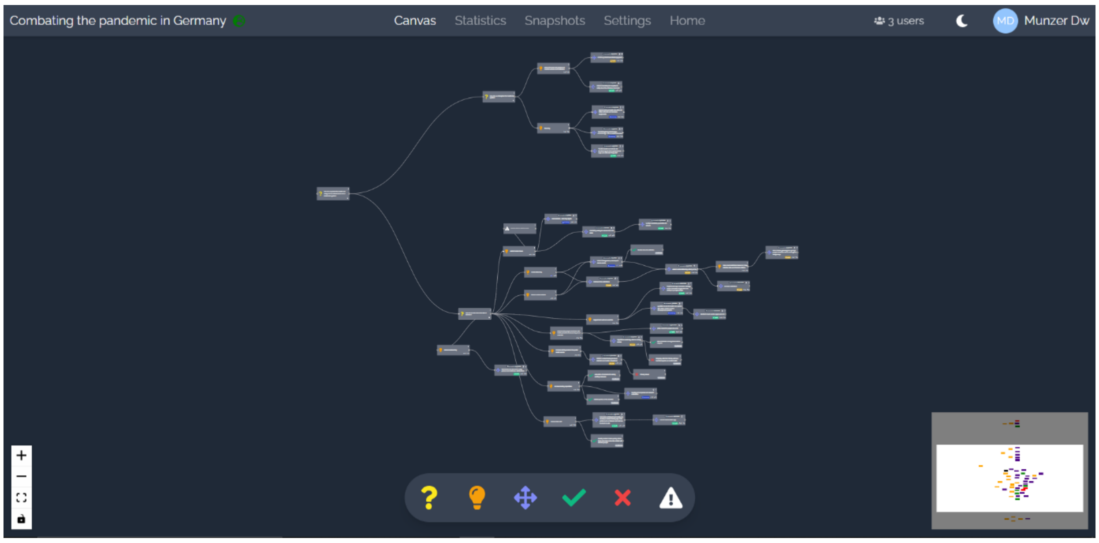


Figure 42: Canvas view demonstration
Source: Own illustration (wickedproblems.io)

Figure 42 shows the resulting canvas view of our case. Of course, many more nodes can be added and structured differently. Though for the demonstration one can see how the Corona pandemic can branch off in many directions and grow rapidly. It also illustrates the diversity of the problem and the different perspectives there are to look at it (i.e. there are many different questions to be asked). Those perspectives interplay with each other and influence the outcomes of the actions that are taken.

Furthermore, it is possible to create 3 labels for the actions in this wicked problem: social, economic, and health. The label "social" in yellow will be assigned to actions taken to improve the situation from a social perspective, like funding nursing homes to buy digital equipment to host video calls with relatives, since visiting was not possible anymore. This can also be labeled as "health" (light green) because it relates to mental health. The same logic applies to the other labels. Of course, this choice of label is not mandatory, one can choose other sets of labels for this case.

Coming to the snapshots feature, many possible datasets can be used to track the situation and performance of the actions. For this demonstration, we can use the number of daily cases, the number of daily deaths (ourworldindata.org), the monthly unemployment rate, the monthly production index ("Produktionsindex"), and the monthly contracts index ("Auftragsindex") (destatis.de). The data is downloaded as a CSV file, filtered and processed in a Python Jupiter notebook, and then added to the snapshots via the endpoints (see an example of "unemployment rate" in appendix 10.3). The number of daily cases and deaths provide a measurement of the effectiveness of handling the coronavirus from a health perspective, while the other metrics show the impact of the crisis from an economic point of view. Here is the example of snapshots combined again as mentioned before in the features section, showing all actions, number of cases, and deaths:

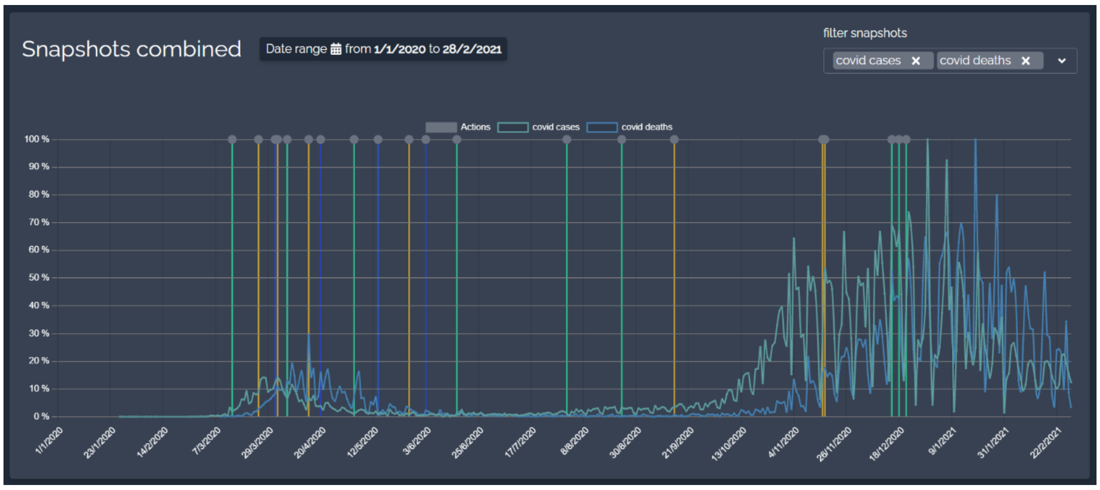


Figure 43: Snapshots combined view demonstration
Source: Own illustration (wickedproblems.io)

One can see how at the beginning the necessary measurements that were taken by the government drastically reduced the number of cases and deaths (first wave). Later, one can see that the actions taken to launch the vaccination campaign at the end of 2020 have been effective as the number of deaths declined while the number of cases is still up. This shows that prioritizing vaccinations groups likely have resulted in much fewer deaths.

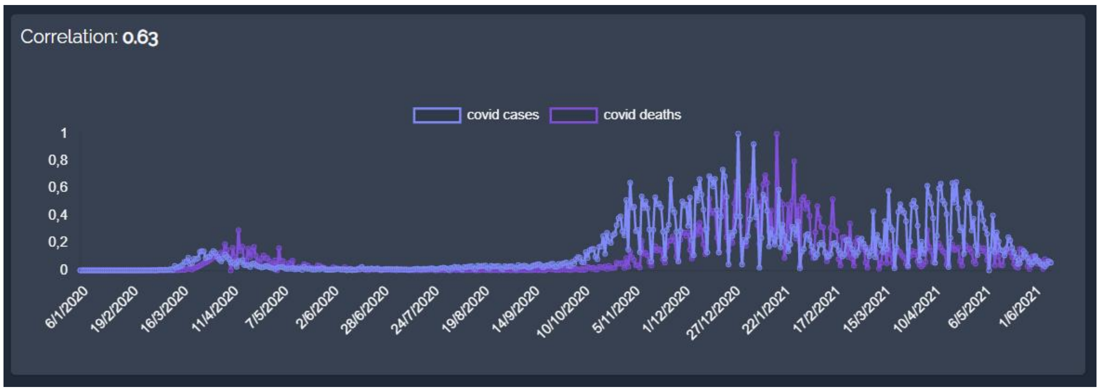


Figure 44: Demonstration of correlations feature
Source: Own illustration (wickedproblems.io)

The correlations feature also shows interesting insights (Figure 44). Here it depicts that the value of daily cases and deaths are not closely correlated, especially at the end. Deaths usually behave in the same direction as the daily cases, but they seem to be shifted to multiple days after. In the end, the number of deaths is not increasing as the number of cases anymore.

This demonstration case visualizes the resolution process to combat the Corona pandemic. It plots the objectives, ideas, and actions of the government and makes clear how these actions have affected the situation of the pandemic by showing multiple datasets as key performance indicators.