



Software Design Document (SDD)

1. Introduction

This System Design Document (SDD) presents the technical details behind the PoliSAM system design. It starts with a mentioning of our main design goals, then it moves to describing the general architecture (high-level description) along with its components, and finally it gives a step-by-step explanation of the implementation process (low-level description) along with the proper justifications.

2. Design Goals

Apart from making the PoliSAM App a properly functioning server which satisfies the specifications introduced in the previous document, we had a couple of design goals in mind:

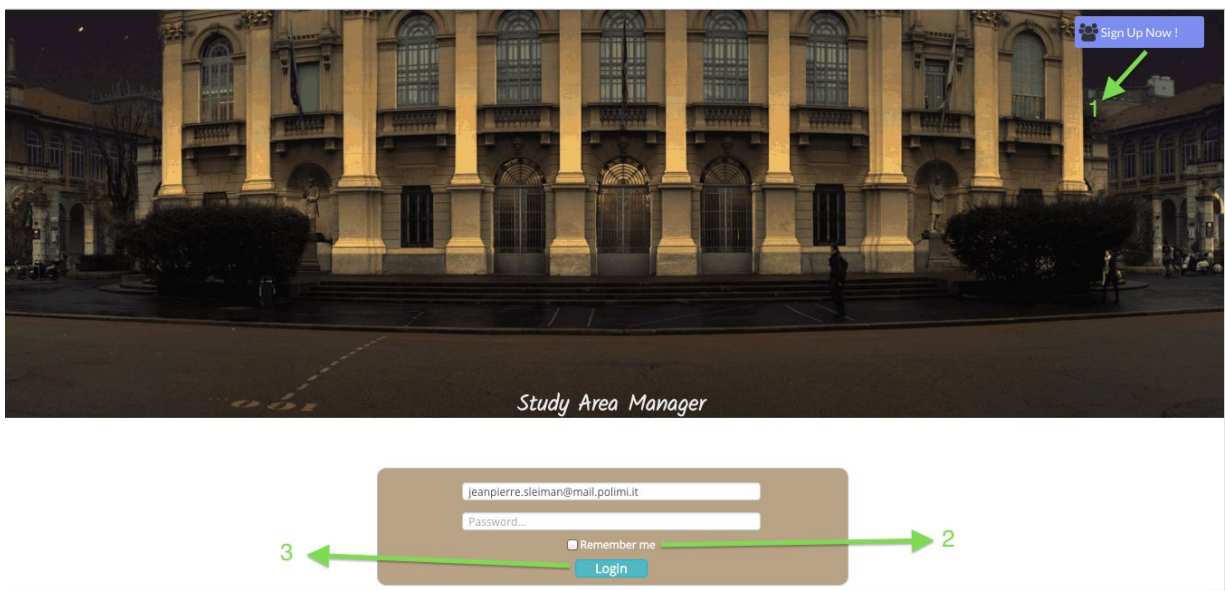
- **Scalability:** Our App must be scalable in terms of being able to support multiple users (Polimi students) as well as support many potential study-areas (especially classrooms which also have an associated schedule defined at the beginning of every semester).
- **Reusability of code:** To minimize the implementation time and the debugging time, every part of the final system was designed and tested separately before integrating them altogether.
- **Usability:** It was important for us to create a user interface that was highly intuitive and easy to work with, without having to go through any sort of user manuals.
- **Wearability:** The users should have access to our App wherever and whenever needed. Which is why our goal was extended to make PoliSAM work on any desired device (especially on mobile phones).

3. General Architecture

In this section we merely give a front-end view of our system, with its different building blocks along with their corresponding functionalities. While the way we defined our logics and events, as well as what was included in our database, will all be further elaborated in the upcoming section.

We start by showing below, PoliSAM's two User GUI's:

User GUI 1



User GUI 2



Now by referring to the numberings in the above two figures, we form the following table, showing the functionalities of the different components:

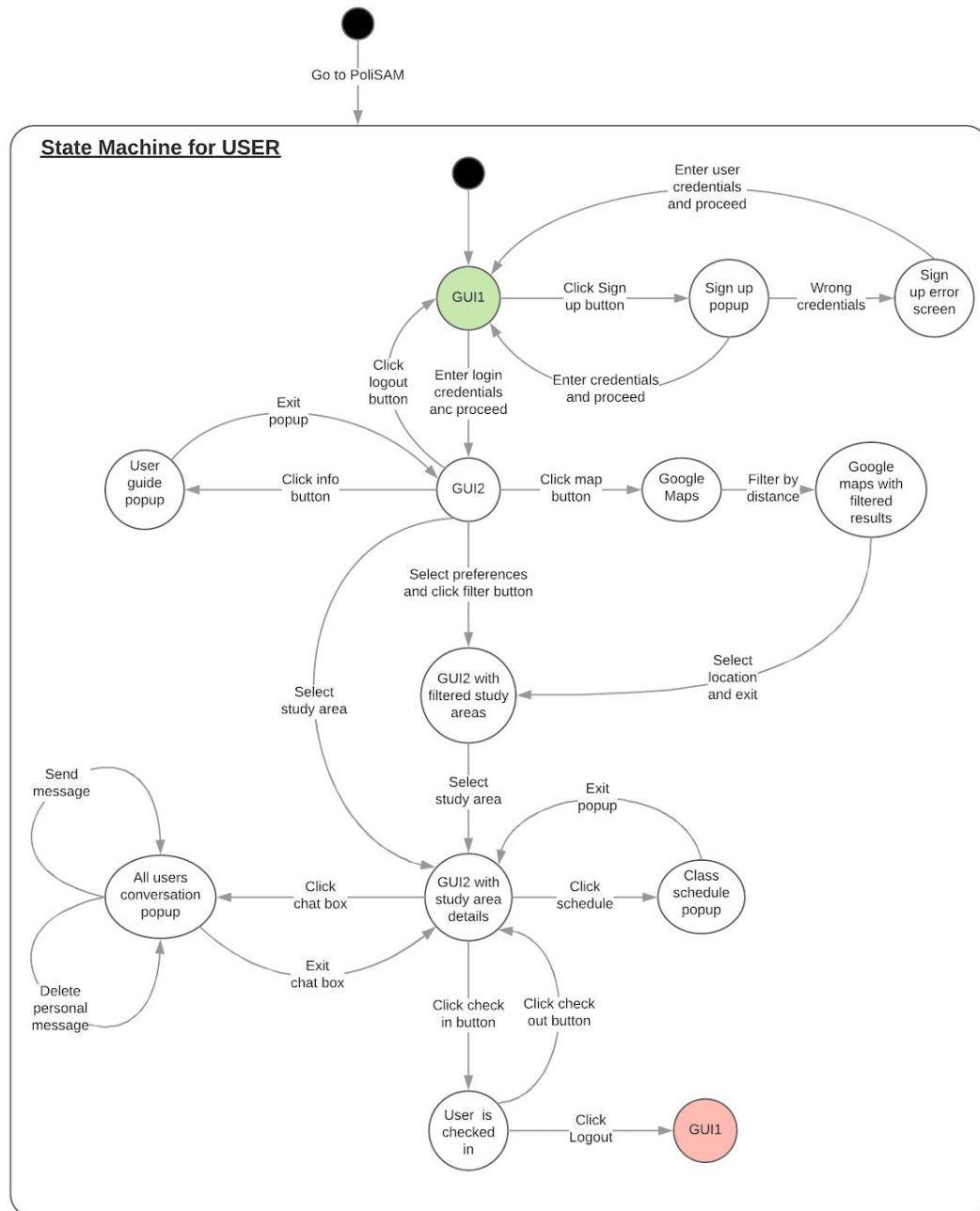
Numberings	Functionalities
1	Opens a popup where the user enters his information for signing-up
2	Saves the user's email on the this page (when enabled)
3	Logs in the user after entering the correct information, and takes him to GUI 2
4	Gives some help on how the user should interpret the information displayed on GUI 2
5	Filters the displayed study-areas according to the desired choices entered above
6	Displays all the study-areas available in our database
7	Selecting one of the displayed study-areas reveals further information about it on the right, and enables the user to perform certain actions connected to it
8	Checks in or Checks out the user from the selected place
9	Displays on the left, the place where the user is checked in, in case he is looking for other options at the same time (it only appears during check in)
10	Allows the user to write messages which can be seen by other users (the messages correspond to a specific selected area from [7])
11	Gives the user his current location on a google maps interface, as well as the possibility to filter study-areas by distance (also the areas' locations are displayed on the map as tags)
12	Logs the user out and takes him back to GUI 1

Below we show a state machine diagram that integrates all of these functionalities together to model our full PoliSAM system, thus showing all of the above mentioned parts in action:

PoliSAM (FSM)

State: all visually different screens are considered as different states

Input/ action: any event that leads to a change screen (visually)



4. Implementation

In this section, we delve deep into what lies behind our system's database as well as our system's workflow (which includes the logic driving the defined events and conditionals). In other words, this provides a back-end point of view of our web application.

Database

For the sake of designing our PoliSAM server, it was quite obvious that we initially required a database that held the different information related to the **Users** as well as the **Study Areas**, and later on the information related to the messaging feature where a specific user enters a certain **Post** corresponding to a desired study area at a given timing. **Bubble** provides the developer the possibility to create a custom database on the same working environment, without having to connect to an external one (although there exists a plugin that allows developers to read/write data from databases such as MySQL databases). The idea behind filling up our database in **Bubble** is based on defining a general **Type**, and then for each type specifying the appropriate **Fields**.

User

This is the first defined **Type**, and it is actually built-in automatically with any **Bubble** application and it contains, by default, the **Field**: email which is of type **text**. It also has an associated password with the email, but it is hidden, even from the developer. We have added some extra fields to the **User** class: the first one is **Username** which is of type **text** (it is used to specify a unique username for every User who signs up), the second one is **Place** which is of type **Study Area** (this will be defined later on) (it is used to show in which Place the student is checked in), and the last one is the field **Checked in** which is of type boolean (yes/no) (it is used to show whether the user is checked-in in some place or not)

Fields for type User (show deleted fields)

Type name	User			
Checked in	yes / no	default	false	 
Place	Study Area			 
Username	text	default		 
email	text	Built-in field		

The proper User data is filled in the database every time a new student signs up.

<input type="checkbox"/>		Email	Checked in	Q	Place	Username
<input type="checkbox"/>	Run as →	marcelo.noriegadelcastillo@ma	no			marcelo
<input type="checkbox"/>	Run as →	jeanpierre.sleiman@mail.polim	no			jp
<input type="checkbox"/>	Run as →	michael.khayyat@mail.polimi.it	no			mk

Study Area

In this part, we defined a **Study Area** general type with the following fields: **Building**, **Campus**, **Name** and **Category** which are of type **text**; **Capacity**, **Occupants** and **Ratio** which are all **numbers**; also a field for **Address** (of type **geographic address**) and a field for **Schedule** (which is of type **image**).

Fields for type Study Area (show deleted fields)

Type name

Study Area



Address

geographic address

Building

text

Campus

text

Capacity

number

Category

text

Name

text

Occupants

number

Ratio

number

Schedule

image

In this case, the data for the different study-areas had to be hardcoded in the database, since we do not have any access to the Polimi database (to which we can connect and thus read its contents). Therefore, we were obviously only able to enter a limited number of examples. Moreover, concerning the **Schedule** field, only the study-areas with category

“Classroom” had an associated image; and it should be clear by now why we had used an image to represent a semester schedule instead of having to rigorously create a proper schedule with actual data related to dates and times and slot availability.

Address	Building	Campus	Capacity	Category	Name
Via Giuseppe Candiani, 72, 20158 Milano MI, Italy	6	Milano Bovisa	220	Classroom	B6.2.1
Via Giovanni Durando, 38A, 20158 Milano MI, Italy	7	Milano Bovisa	98	Classroom	CS.03
Via Giuseppe La Masa, 34, 20156 Milano MI, Italy	15	Milano Bovisa	150	Classroom	LM.4
Via Giuseppe La Masa, 34, 20156 Milano MI, Italy	12	Milano Bovisa	167	Classroom	L.05
Via Raffaele Lambruschini, 4, 20156 Milano MI, Italy	27	Milano Bovisa	154	Classroom	BL.27.05
Via Raffaele Lambruschini, 4, 20156 Milano MI, Italy	27	Milano Bovisa	212	Classroom	BL.27.01
Via Camillo Golgi, 40, 20133 Milano MI, Italy	25	Milano Leonardo	54	Classroom	D.2.3

Address	Building	Campus	Capacity	Category	Name
Via Camillo Golgi, 39, 20133 Milano MI, Italy	21	Milano Leonardo	130	Classroom	E.G.2
Via Andrea Maria Ampère, 2, 20133 Milano MI, Italy	11	Milano Leonardo	248	Classroom	III.B
Edificio 2A, Piazza Leonardo da Vinci, 32, 20133 Milano MI, Italy	3	Milano Leonardo	88	Classroom	S.1.5
Edificio 2A, Piazza Leonardo da Vinci, 32, 20133 Milano MI, Italy	2	Milano Leonardo	380	Classroom	N.0.1
Via Camillo Golgi, 40, 20133 Milano MI, Italy	25	Milano Leonardo	120	Classroom	D.3.3
Via Andrea Maria Ampère, 2, 20133 Milano MI, Italy	11	Milano Leonardo	400	Library	Biblioteca Centrale
Via Raffaele Lambruschini, 4, 20156 Milano MI, Italy	27	Milano Bovisa	60	Common Area	Edificio BL27


Address	Building	Campus	Capacity	Category	Name
Via Giuseppe Candiani, 72, 20158 Milano MI, Italy	6	Milano Bovisa	220	Common Area	Edificio B62
Via Giuseppe La Masa, 37, 20156 Milano MI, Italy	12	Milano Bovisa	130	Common Area	Edificio B12
Via Camillo Golgi, 20, 20133 Milano MI, Italy	26	Milano Leonardo	100	Common Area	Edificio 26
Via Edoardo Bonardi, 9, 20133 Milano MI, Italy	16	Milano Leonardo	150	Library	Biblioteche Dipartimenta
Edificio 2A, Piazza Leonardo da Vinci, 32, 20133 Milano MI, Italy	2A	Milano Leonardo	250	Common Area	Acquario
Via Edoardo Bonardi, 14, 20133 Milano MI, Italy	14	Milano Leonardo	150	Classroom	B.2.4
Via Andrea Maria Ampère, 2, 20133 Milano MI, Italy	11	Milano Leonardo	100	Common Area	Patio
Via Raffaele Lambruschini, 4, 20156 Milano MI, Italy	25	Milano Bovisa	300	Library	Biblioteca delle Ingegneri Bovisa

Post

The last dataset was introduced when we decided to include the Messaging feature in PoliSAM. This class contains the following required fields: a **Message** of type text (which is the message written by the user to be posted), a **Person** of type User (which signifies the student who wrote the message), a **Timing** of type date (which is the time at which the

message was posted), and a **PostPlace** of type Study Area (which is the place to which this message corresponds).

Fields for type Post

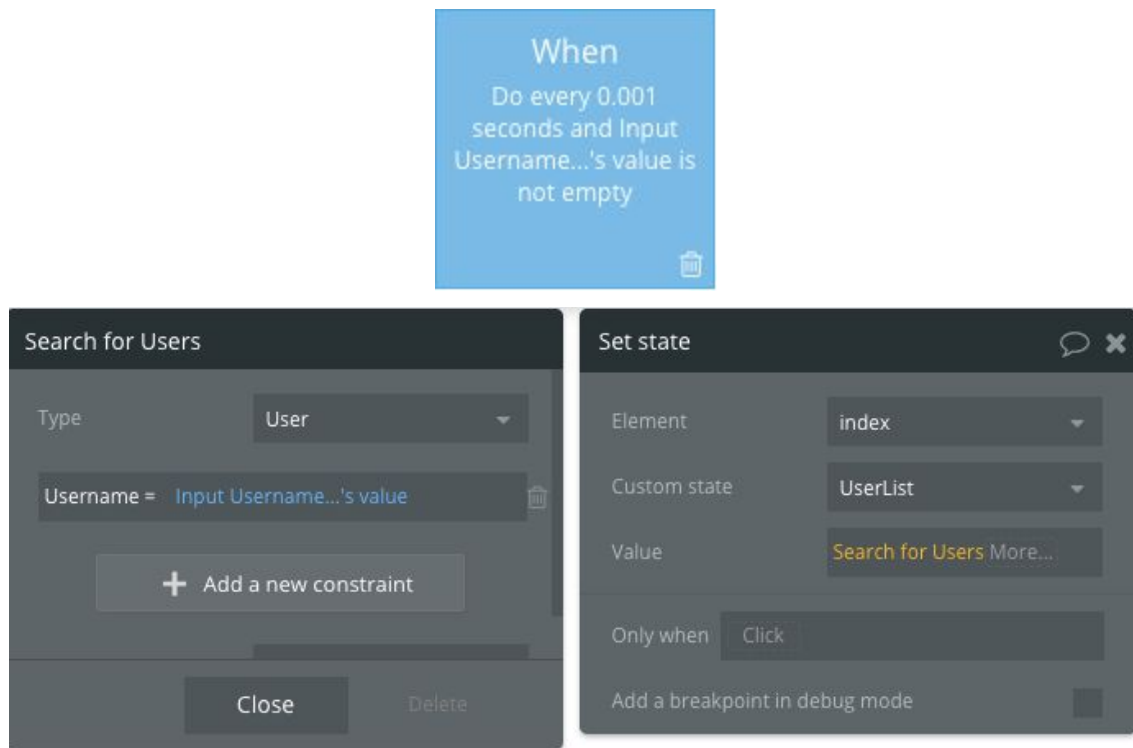
Type name	<input type="text" value="Post"/>	
<input type="text" value="Message"/>	text	
<input type="text" value="Person"/>	User	
<input type="text" value="PostPlace"/>	Study Area	
<input type="text" value="Timing"/>	date	

Events Workflow & Conditionals

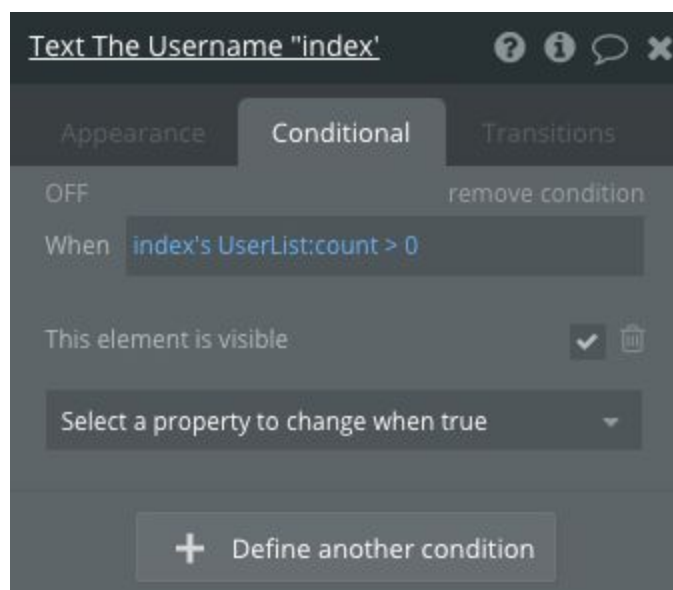
Now we start defining the reasoning behind the functioning of our different “modules” by specifying what we expect from each module to perform along with the corresponding underlying logic (while justifying some indirect or unclear choices). For added coherence between the documentation and the actual implementation, we use images directly from the **Bubble** development environment when defining our events and conditions (it is fairly simple to understand the code as it is written in plain english and no complex syntax is involved):

Signing Up

- When the user clicks on the “**Sign Up Now !**” button, a popup should appear where the user is supposed to enter a username, a Polimi email, and a password and then clicks on a “**Sign Up**” Button.
- For the sign-up procedure to be successful, the user has to enter all the demanded information and the entered username must not be previously used by another registered user, otherwise the current user has to be notified about that immediately after writing his username (which is why we trigger an event every 0.001 seconds as shown below).



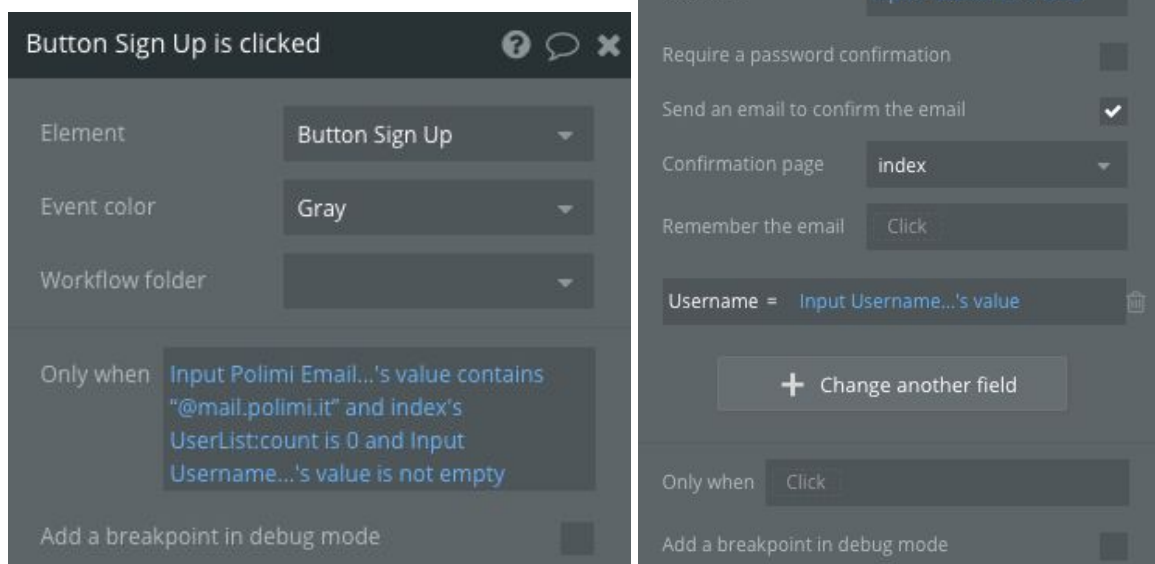
UserList is a custom state that is not stored in the database, and is used to search if there is a user with a similar username as that entered by the current user. Then we were able to decide whether the user has entered an invalid username by checking the size of this state, and therefore making a warning message visible as a result.



- Moreover, only Polimi emails are accepted during registration. If the sign-up is not successful for any of the aforementioned reasons, and the user clicks on the “**Sign Up**” button, a popup should appear indicating so.

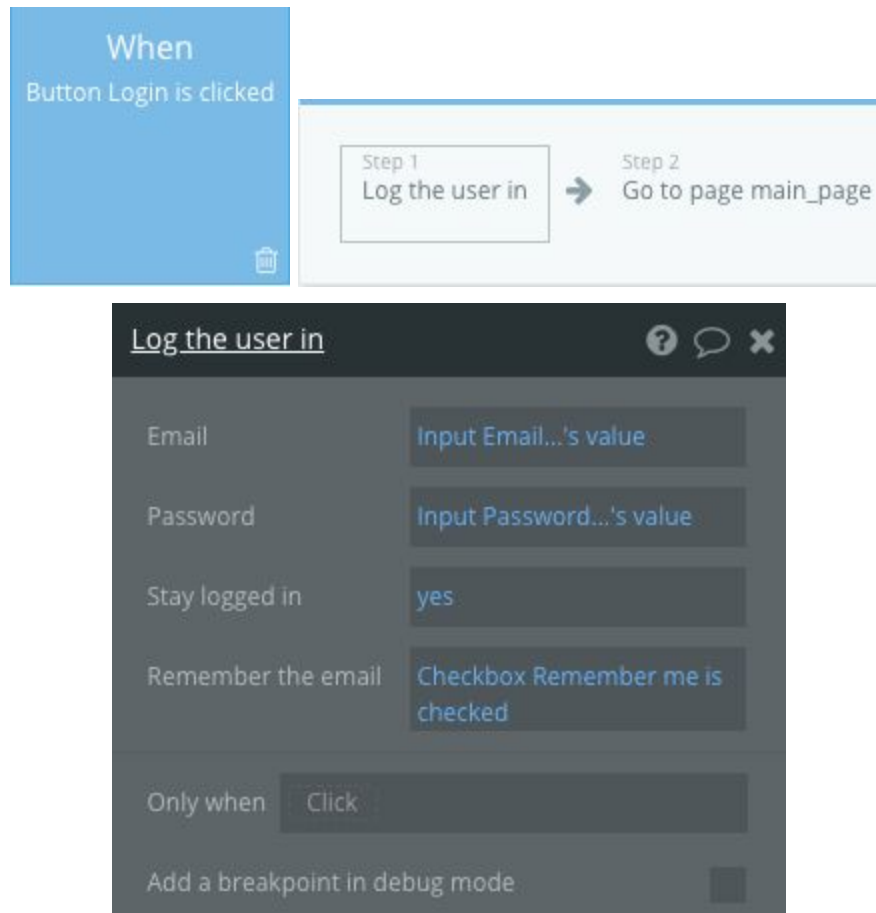


- When all conditions are satisfied, the sign-up procedure is carried out. If the email already exists the user is notified automatically by a built in **Bubble** functionality. Also the user is automatically added to the database with his entered email and password (a confirmation email could be sent upon sign-up, if desired), but the username has to be saved manually.



Logging In

- This step is fairly straightforward, as it only involves entering the correct email and password (a message is automatically generated otherwise), and then taking the user to the second GUI (note that GUI 1 is referred to as “index” and GUI 2 is referred to as “main_page”). The user can also request to have his email remembered whenever he uses the App from the same device.

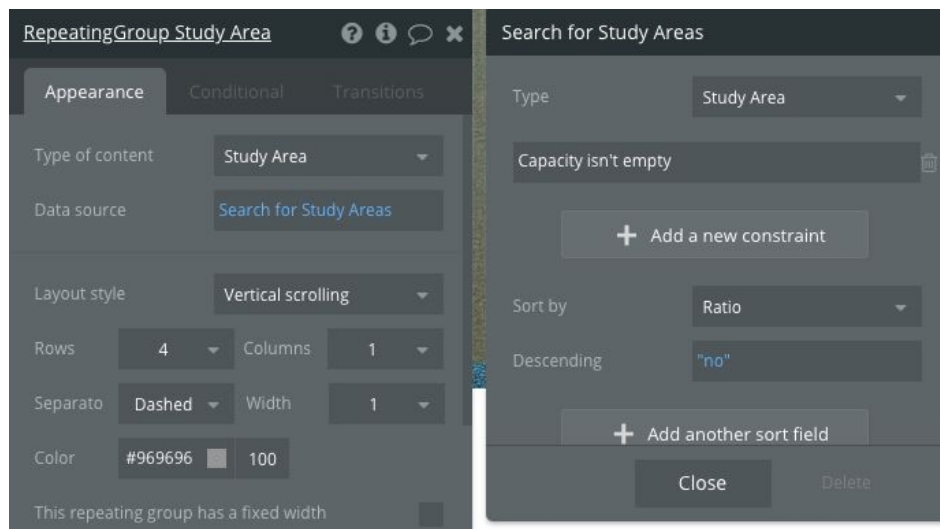


Get Guidance

- This is also very simple since it involves clicking an icon and thus triggering an event which opens a desired popup with some static text.

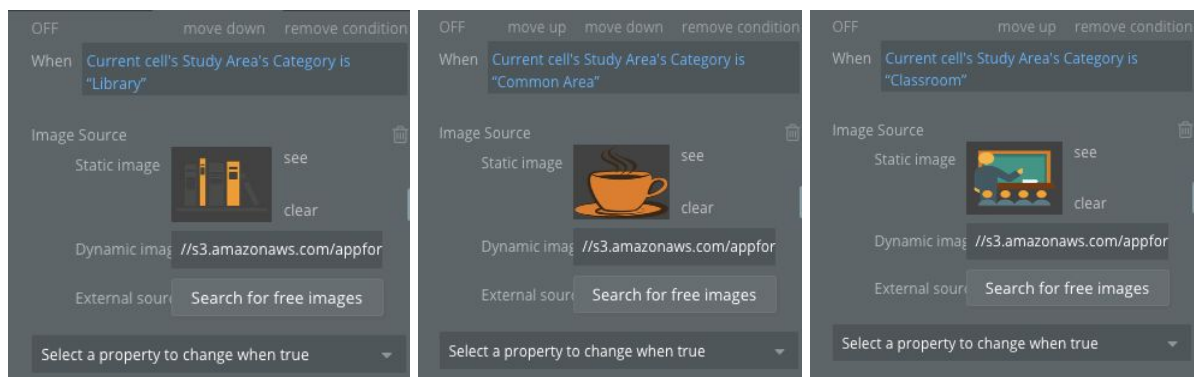
Display of Study-Areas

- A major part of the project was to properly display all the study-areas assigned in the database, in a proper fashion (shown in the left column of GUI 2). This was done with a feature in **Bubble** called a **Repeating Group** where only the first box element is designed and then all other boxes follow accordingly. In our case, we wanted to show the name of the place, the category to which it belonged (using an image icon), and the number of occupants per total capacity. Moreover, the elements were sorted from top to bottom in increasing order of Ratio.



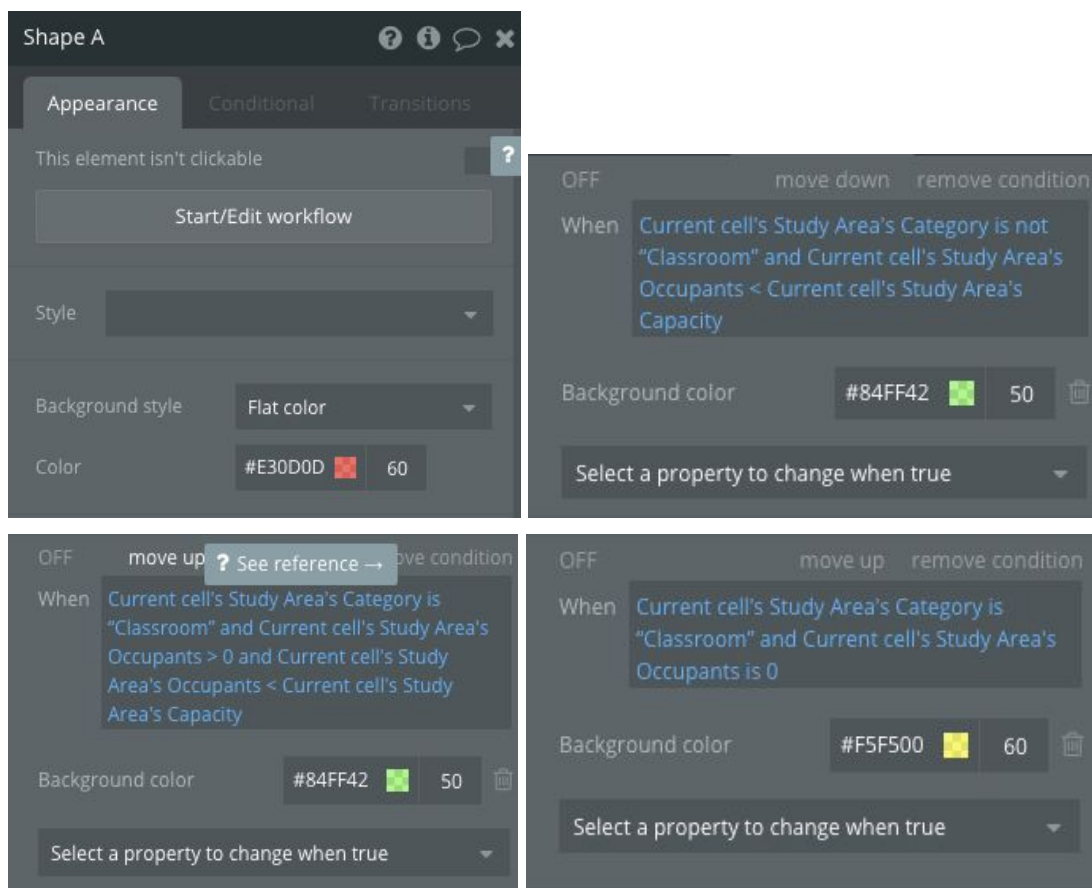
The condition during the search: **Capacity isn't empty**, will be made clear later on when we discuss the use of the Google Maps Interface.

- Moreover, we had to define a set of specific conditions which are responsible for choosing the type of image icon that has to be visible (according to the study-area's category)



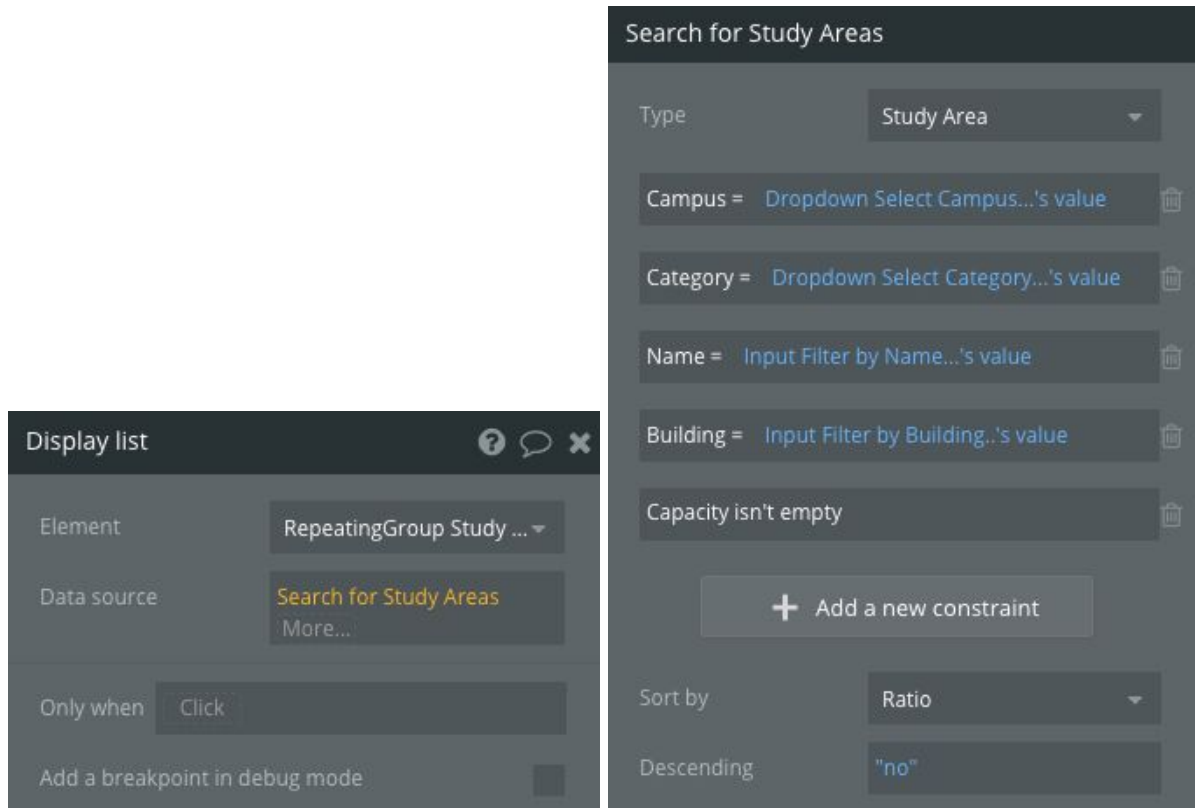
and other conditions for displaying the appropriate color accordingly:

Red when the study-area is fully occupied, Green when the study-area is not a classroom and is not fully occupied and also Green when the study-area is a classroom and is not fully occupied but also does not have 0 occupants, since in cases where classrooms have zero occupants, there could be a lecture going on at the moment and so this should be indicated with a different color (Yellow in our case) (in such a situation, the student ought to check the associated semester schedule that is provided as an image in order to know whether the classroom is available for studying or not).



Filtering

- This button allows the user to filter out undesirable options (from the left column's repeating group discussed above) and keep the ones he/she cares about (in terms of Campus, Category, Building, and Name as well).

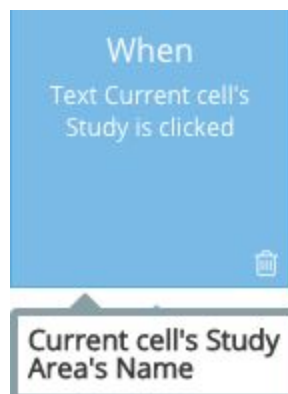


Displaying All Options

- In case the user filters out some of the options and wants to display them all again without having to refresh the page, he is able to do so by pressing the **“Display All”** button.

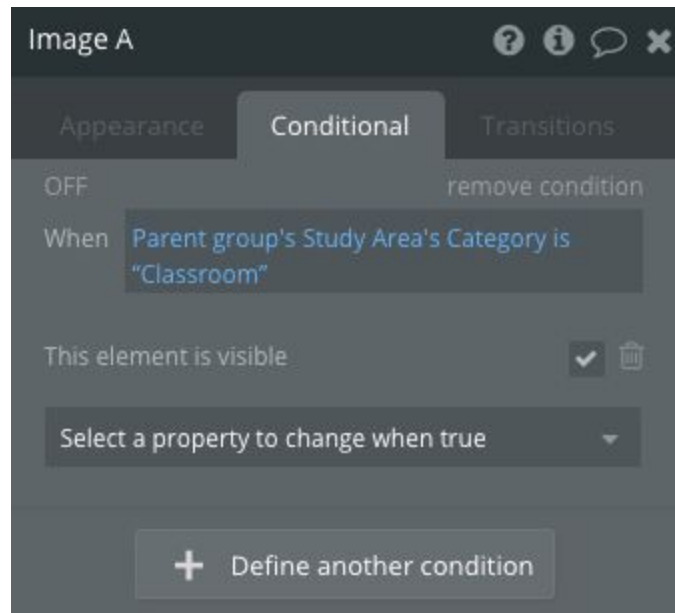
Checking In and Out

- This is probably the most essential aspect of the whole project, and it involves several steps and conditions before obtaining a satisfactory behavior from the App: The first step is to be able to select the desired study-area from the left column, and thus triggering an event which displays all the details associated with it, as well as showing a Check-in button and the proper semester schedule in case the user selects a classroom. All these elements would be displayed together in the form of a group (called **Group Study Area**) shown in the center of the GUI.



The screenshot shows the 'Display data' configuration panel in an application. It has a title bar with a question mark, a speech bubble, and a close button. The panel contains the following settings:

- Element:** A dropdown menu showing 'Group Study Area'.
- Data to display:** A text field containing 'Current cell's Study Area'.
- Only when:** A dropdown menu showing 'Click'.
- Add a breakpoint in debug mode:** A checkbox that is currently unchecked.



When the user clicks the **“Check in”** button, he is allowed to do so only if he is not already checked in any study-area, and if the number of occupants in the selected place is less than the full capacity. Ofcourse, everytime someone checks in to a given place, the number of occupants as well as the ratio are altered accordingly. Moreover, the status of this user’s **Place** and **Checked in** fields should be updated as well, and after that, the **“Check in”** button is replaced with a **“Check out”** button (we did not want to add two buttons that appear at the same time for check-in and check-out in order to have better chances in making the App accessible from a mobile phone which has a relatively narrow width).

Button Check in is clicked

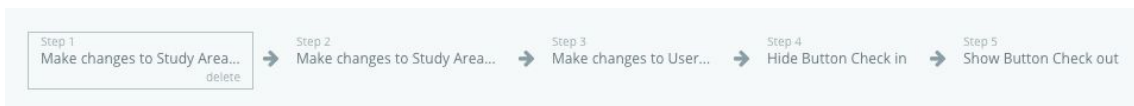
Element: Button Check in

Event color: Gray

Workflow folder:

Only when: Current User's Checked in is "no" and Parent group's Study Area's Occupants < Parent group's Study Area's Capacity

Add a breakpoint in debug mode



Make changes to Study Area...

Thing to change: Parent group's Study Area

Occupants = This Study Area's Occupants + 1

+ Change another field

Only when: Click

Add a breakpoint in debug mode

Make changes to Study Area...

Thing to change: Parent group's Study Area

Ratio = This Study Area's Occupants / This Study Area's Capacity

+ Change another field

Only when: Click

Add a breakpoint in debug mode

Make changes to User...

Thing to change: Current User

Checked in = "yes"

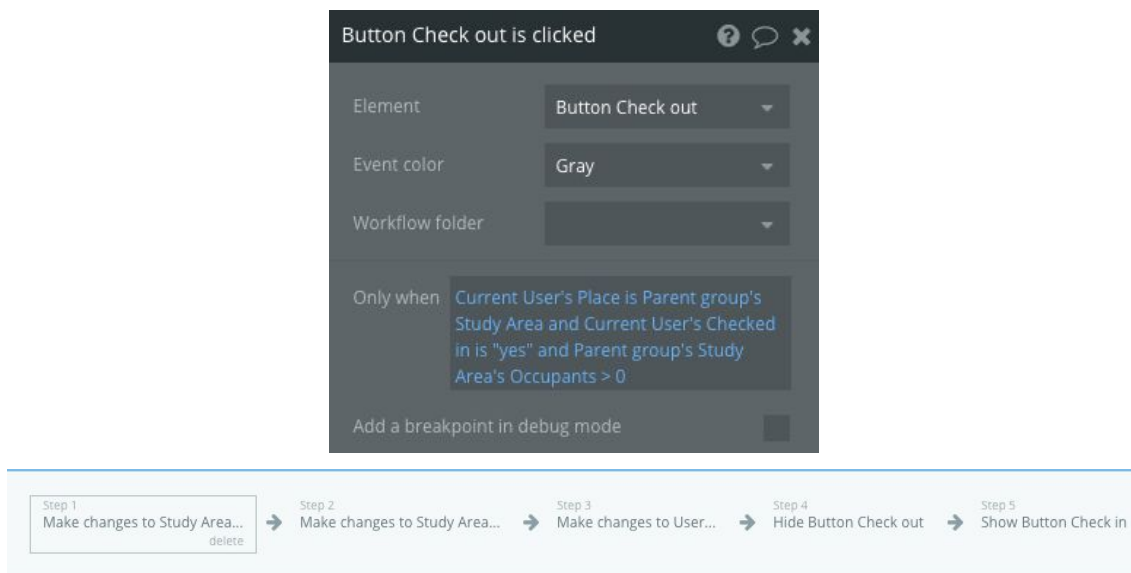
Place = Parent group's Study Area

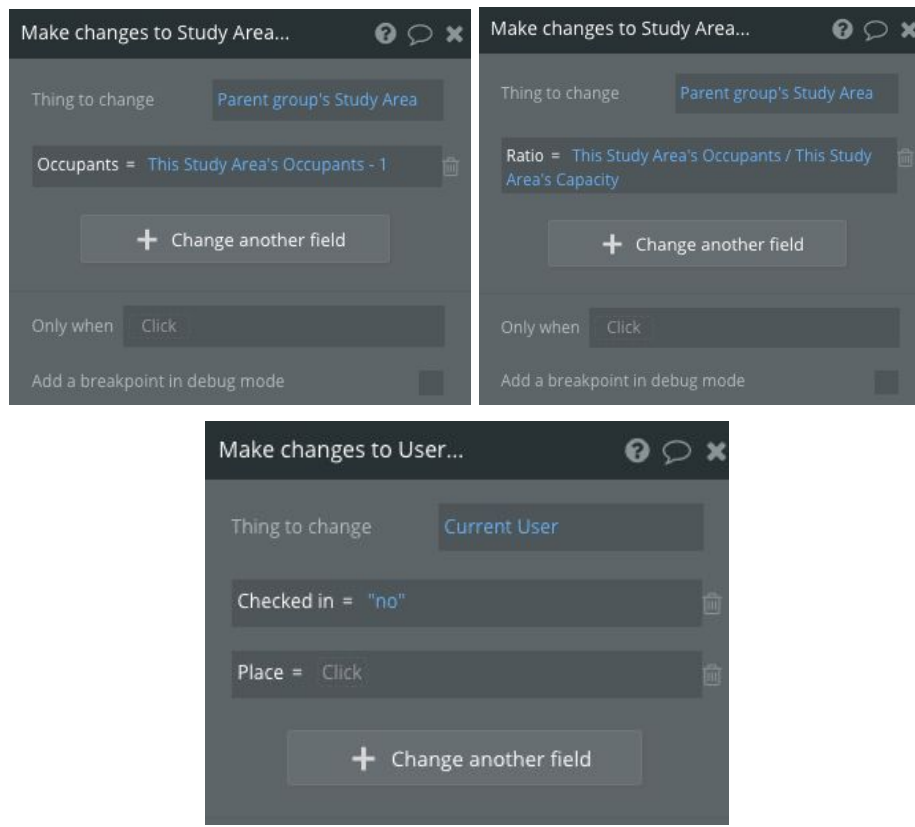
+ Change another field

Only when: Click

Add a breakpoint in debug mode

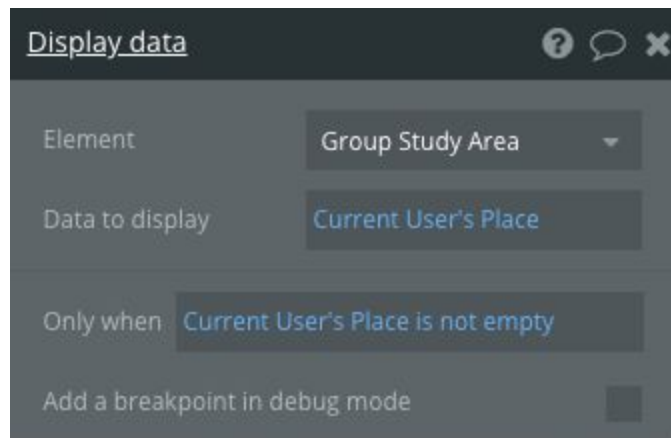
Whereas when the user clicks the “**Check out**” button, he is allowed to do so only if he is checking out from a place in which he is currently checked in, and if the number of occupants in the selected place is greater than zero (this might be redundant since if the student has to be checked-in then for sure this quantity would always be greater than zero, but there is no harm in adding this condition to be on the safe side). Ofcourse, everytime someone checks out from a given place, the number of occupants as well as the ratio are altered accordingly. Moreover, the status of this user’s **Place** and **Checked in** fields should be updated as well, and after that, the “**Check out**” button is replaced with a “**Check in**” button.



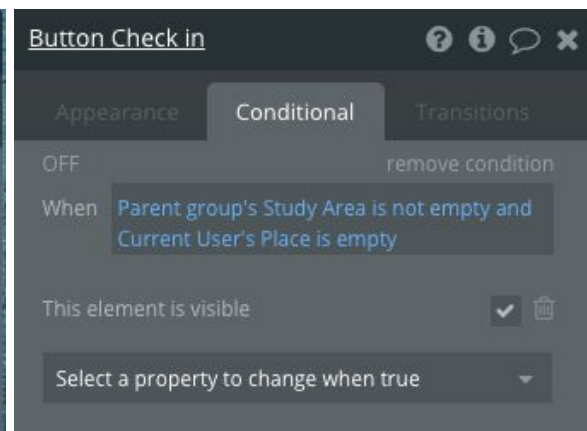
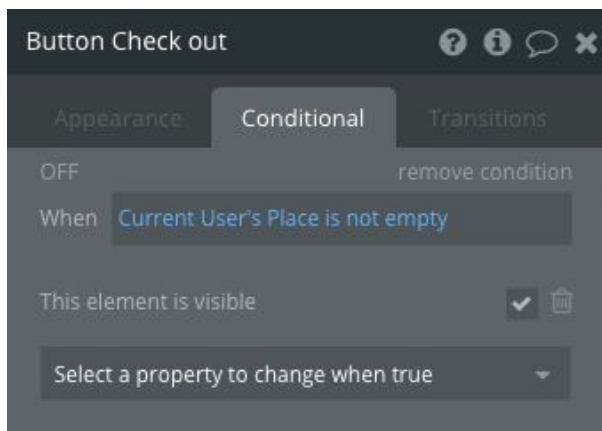


Furthermore, when the user checks in, a message indicating the place with a “thumbs-up” icon appears, and disappears when he/she checks out. As mentioned in the previous section, clicking on this icon enables the user to go back to the place in which he is checked-in, just in case he decides to search for other options in the meantime.

Another issue is related to the possibility that the student might be checked-in and suddenly decides to logout, or loses connection, thus having to refresh the GUI 2 page again (this leads to the loss of the front-end’s dynamic information, but whatever is in the database is still there). Therefore, an event was added that solves this problem by displaying the saved checked-in place, associated with the user, in the middle of the page again.



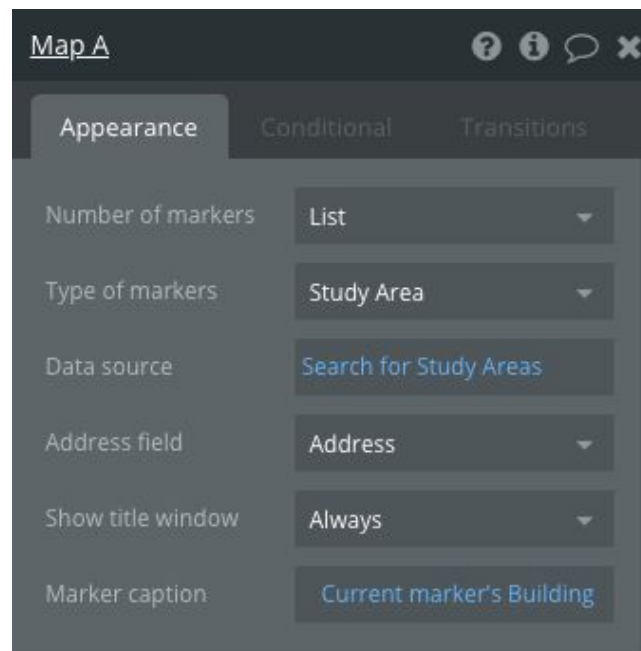
Finally, in order to decide which button should be visible and which one should not, as soon as the page is loaded, we applied the following logic: The check-out button appears whenever the current user's **Place** field is not empty, while the check-in button appears when it is empty and the user has selected an option to be displayed in the central Group.

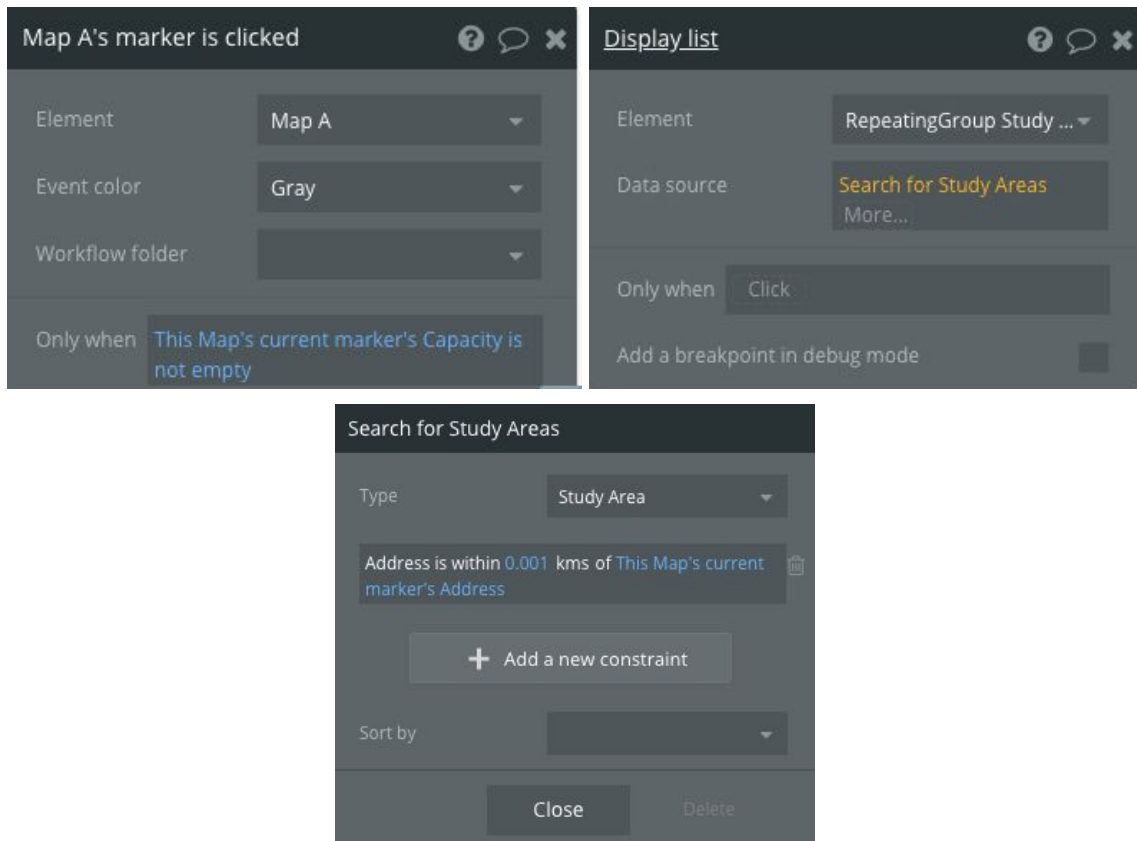


Google Maps Feature

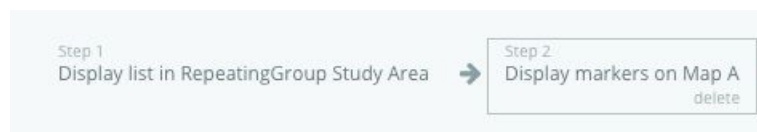
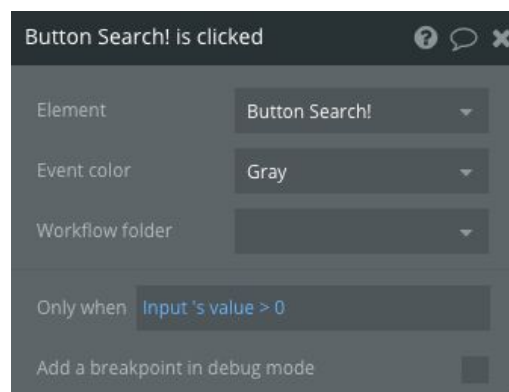
- **Bubble** comes with a pre-installed plugin that allows the developer to benefit from the functionalities provided by the Google Maps API (such as finding locations on a map, giving the device's current geographic location, calculating geographic distances...). Therefore, we found it quite interesting to use these additions and apply them to our PoliSAM App.

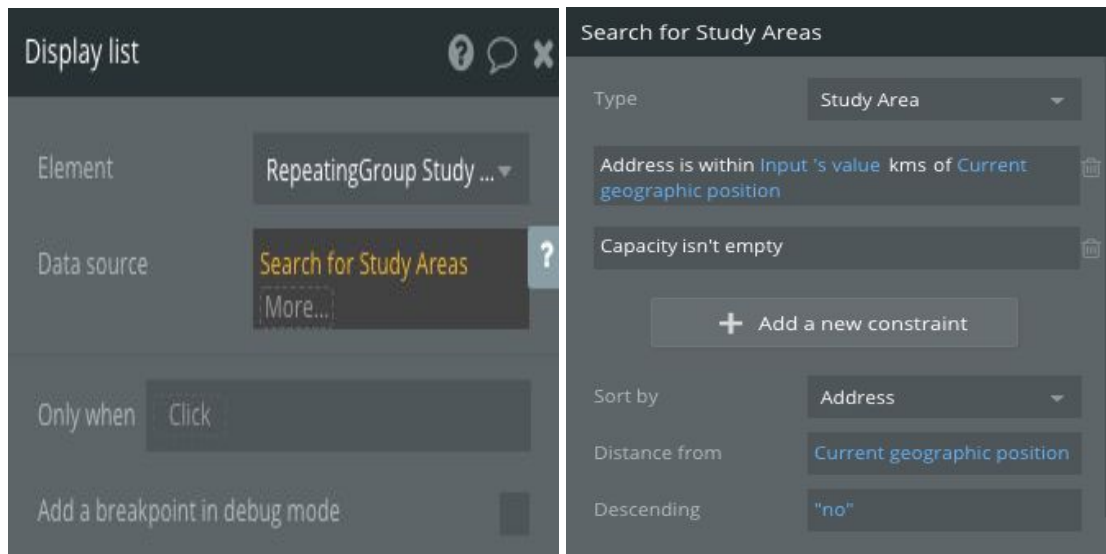
First of all, when the Maps image icon on the right is clicked, a popup appears. The latter contains a map that initially shows the user's current location, and it also contains a search button that filters the study-area options by distance; so if the student requests all the places that are at most 2 km away from him, the map shows him all the feasible options along with their corresponding buildings, and the same options are displayed in the left-hand corner of the screen as well (but this time they are sorted in increasing order of relative distance instead of ratio). Moreover, the user can select his desired building to appear in the left column, starting from the map by clicking on the corresponding marker (sometimes each marker could be associated with several buildings in case they have the same Address).



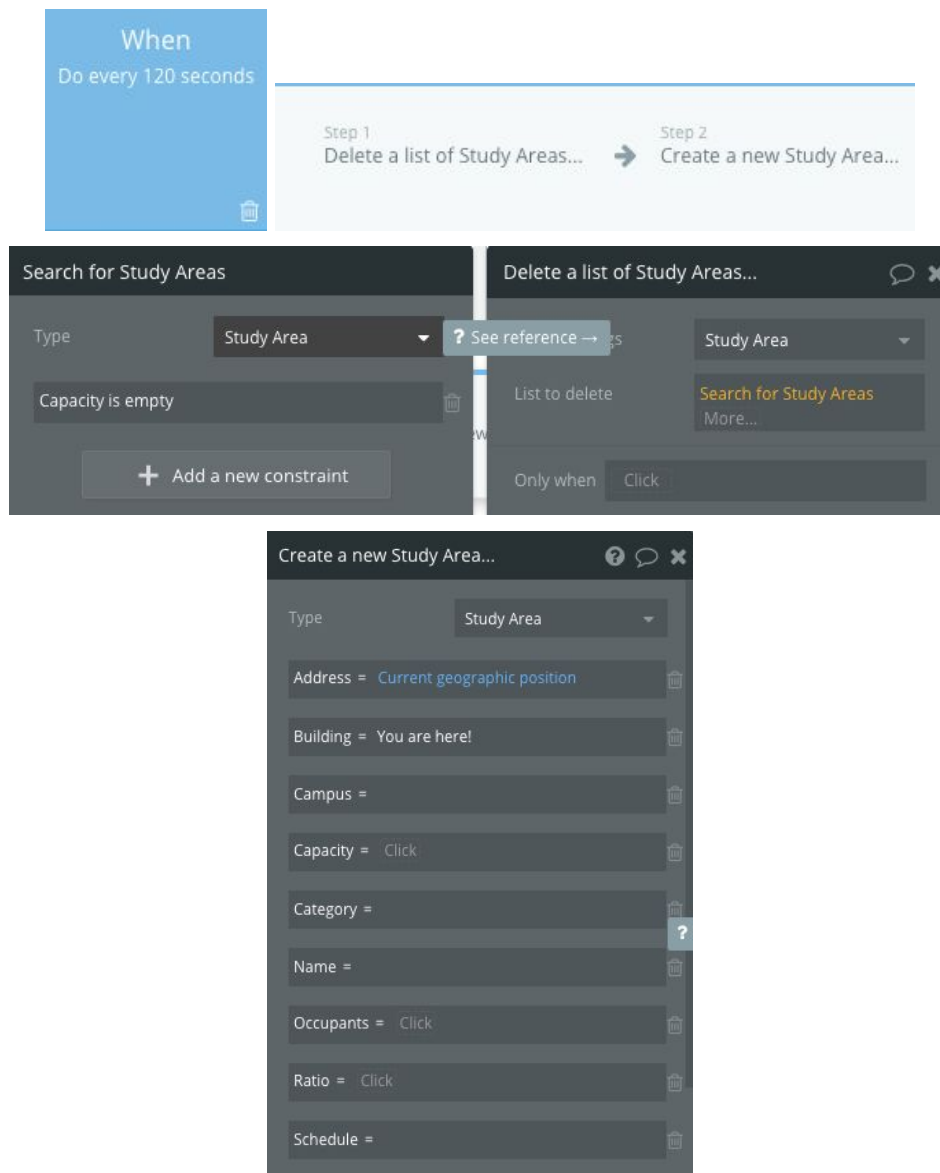


One can notice from the figures below that a condition was added on the inputted desired radius of search, indicating that the user cannot enter a radius that is less than or equal to zero.





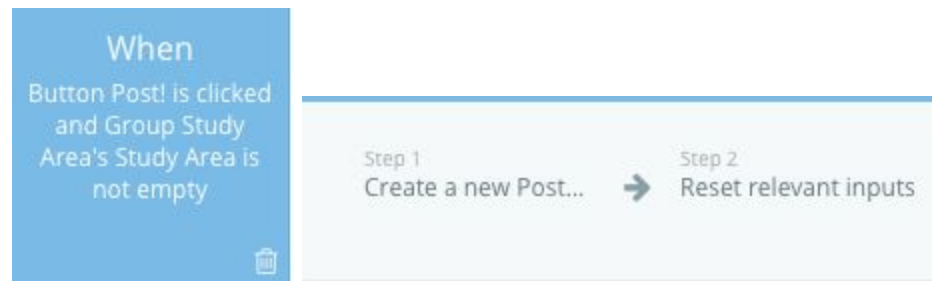
The last thing that was not mentioned in this part is how we measure the current user's position in real-time. The thing is that it is simple to display a user's geographic location (with **Current geographic position**) but that can happen only once, when the page is loaded. As workaround for that, we trigger an event that returns the current position every 2 minutes. However, what we did not explain yet, is the fact that it is not possible to display on the same map a single geographic location as well as the locations given after doing a search in the database (in our case we search for study areas). This is only possible if they are placed on two different maps; and so to avoid that, we had to embed the user's current position within a Study-Area data-type structure and save it in the database along with the other areas. To do that, we fill in the **Address** field (current geographic position) as well as the **Name** field (Your location is here!). And all other fields are empty (this is why we imposed a condition before, stating that we would not want to display in the left column any study-areas with empty **Capacity**). Furthermore, to avoid overloading our database with multiple saved junk locations, we deleted all such locations before saving a new one every 2 minutes! (the same is also done as soon as the page is loaded)



Messaging Feature

- The last interesting feature added to PoliSAM enables students to add posts associated with the selected study-areas (for instance asking questions concerning availability, or questions concerning a lost item, or even just to let others be aware of any unexpected upcomings...). The user should be able to select the messaging image icon thus opening a popup, and in this popup he/she can type messages and post them, see other people's messages, or even delete any of their own previous messages. This was also implemented by using a **Repeating Group** where the first element contains: The messenger's username, the time at which the message was posted (using **Current date/time** within an event triggered at every second, similar

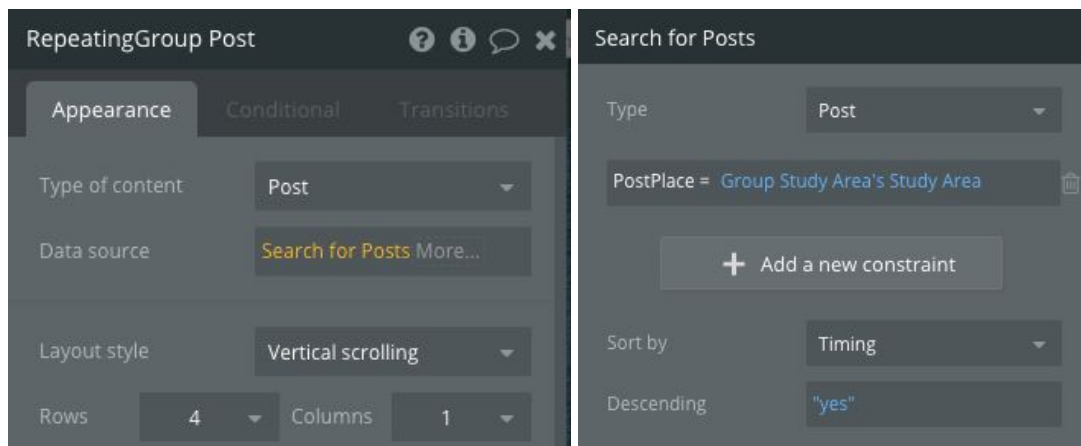
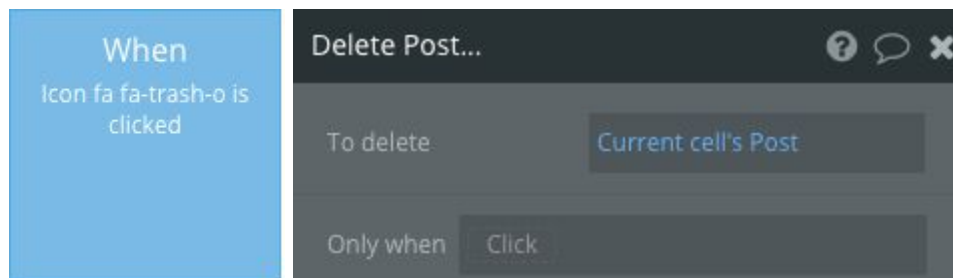
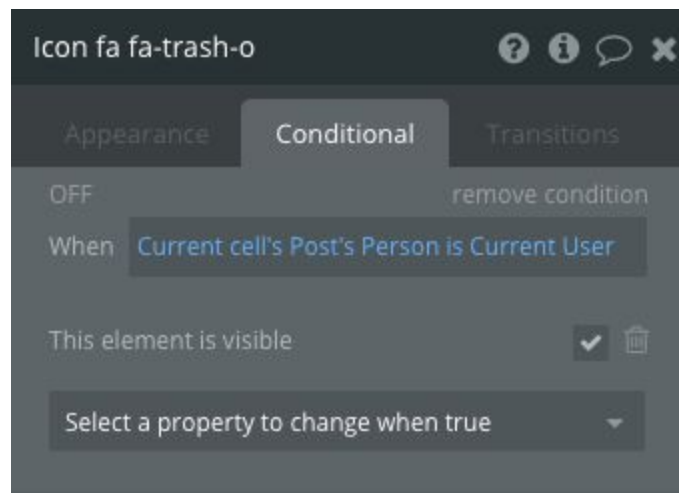
to what was done when attaining the current position every 2 minutes, but instead of saving the value in the database, a custom state **CurrentTime** was used), and the message itself (along with a trash icon that could be used for deleting the message if it is his/her own). The posts are sorted from top to bottom in descending order of timing.

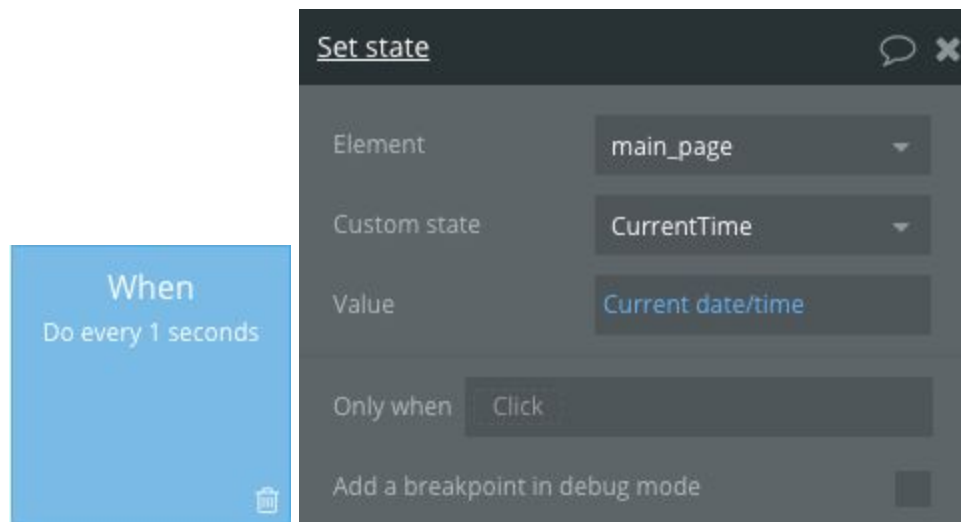


The screenshot shows a dark-themed dialog box titled "Create a new Post...". It features a dropdown menu for "Type" set to "Post". Below this, there are four input fields, each with a trash icon on the right for deletion:

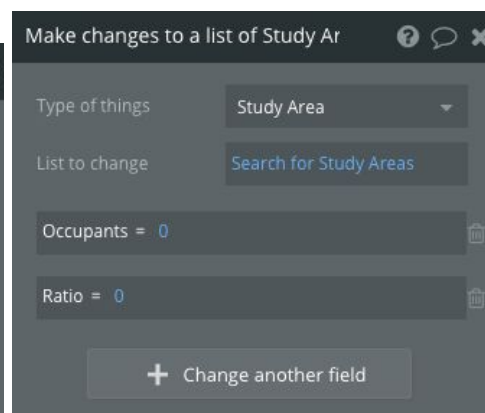
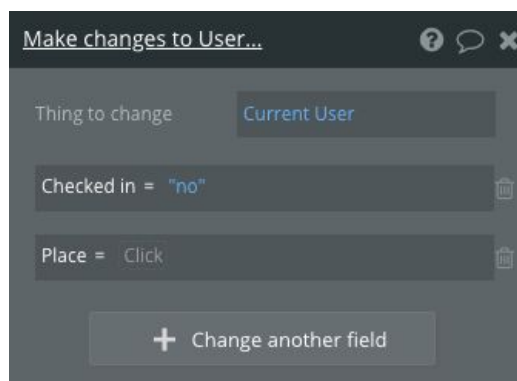
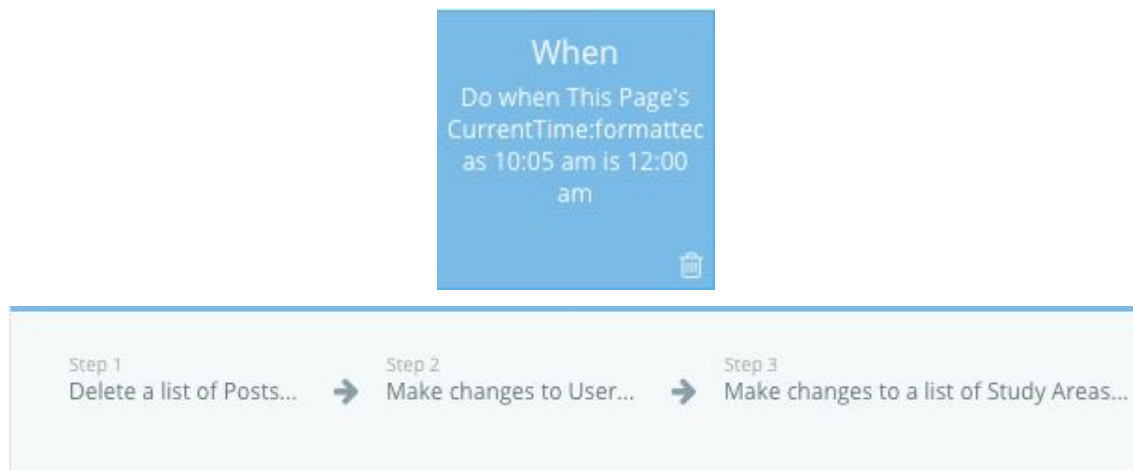
- Message =
- Timing =
- Person =
- PostPlace =

At the bottom of the dialog is a button labeled "+ Set another field".





Moreover, all messages written by all users are cleared every 24 hours at 12:00 am, also anyone who forgot to check-out after leaving is forcibly checked-out and so all study-areas are cleared as well, as shown in the figures below:



5. Responsive feature of Bubble.is