POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

PROMET&O: web application for objective and subjective environmental comfort data visualization and assessment

Supervisors

Prof. Antonio SERVETTI

Candidate

Martina SAUGO

December 2022

Abstract

For buildings to be managed effectively, gathering information regarding indoor environmental quality is essential. Sometimes, raw data is not fully representative of the actual well-being of the of a place's users, which is why the PROMET&O portal was developed.
In PROMET&O the chance of filling a survey about Indoor Environmental Quality combines with a rich dashboard displaying various measures about four comfort domains: Thermal Comfort, Visual Comfort, Acoustic Comfort and Indoor Air Quality.
In order to compare the predicted comfort level, defined by the sensors' data, and the perceived comfort level, defined by the users' opinions, statistics regarding the aforementioned domains are gathered through the survey.
The users are also required to do a brief questionnaire regarding their lifestyle. This will make it possible to look into the potential connection between certain behaviours and perception toward the environment in more detail.
Additional knowledge regarding objective comfort indices is presented in order to benefit the users by educating them about the influence that these measurements can have on their lives and what they signify.
Users can also find hints on how to undertake common tasks and improve the quality of their indoor spaces while keeping an eye on both comfort and sustainability.
The additional opportunity for comparing the graphs in different time windows and among them allows for the creation of insightful correlations that benefit all users' quality of life.
The PROMET&O online portal's major goal is to better understand the correlation between environmental conditions, comfort and habits and to offer tools that help the public become more informed, so that buildings can be more efficient and comfortable.
These features are provided through a Single-Page Application developed in JavaScript React supported by a serverless backend, implemented through Amazon Web Services.

Summary

PROMET&O is a web portal designed in order to collect data about the comfort perception of the occupants of a building. Through the use of a survey, users can provide insights on the level of comfort of their environment. The collection of data about the people's habits, lifestyle and interaction with their environment helps define interesting correlations between comfort perception and behaviour.
The subjective information can be compared with objective measurements collected by an infrastructure of multisensors distributed across the involved spaces. The collected data will be visible in a dashboard, together with notions and hints to provide value to the users through knowledge about different factors that have an impact on their environment and how to better interact with them for more efficient and more comfortable spaces.
An example environment to be taken into consideration could be an office of about 10-12 seats where the users can change from day to day. In each of these offices, a few tablets are made available as compilation stations in order to allow the users to answer the questionnaires and visualize the objective data dashboard.
Because of the alternation of people in our target environment, there are two main ways for the compilation of the questionnaire: a logged way, where the received answers are linked with the ones to the personal information survey and an anonymous way, where the user does not identify and is later asked to fill a set of personal questions or to create an account. In order to achieve all attended results we are expected to build: two surveys, a dashboard, a data management system, a users management system and an email system. Moreover, the portal is expected to be available both in Italian and English language.
Our web application is expected to be deployed over some tablets made available in the office taken into exam. They would be connected to a SIM-generated wireless network, together with a set of multisensors, which communicate through MQTT to their own Objective Data database. This very same database is expected to receive elaborated data about Indoor Environmental Comfort indices and to display said data over a Grafana dashboard, which will be queried and referenced through iFrames in PROMET&O's dashboard.
PROMET&O is also expected to have a backend composed of a Database, an API

Gateway connected to a set of functions, a deployment service, an authentication service and its own domain, used both to make the site reachable and for email communication.
The average approach to the development of a web application is to choose either a monolithic architecture or a distributed one. In the first case we would have a simpler but more limiting and less fault-tolerant architecture, while in the second one we have a much more reliable system but far more expensive and of difficult implementation. An alternative to these two approaches is to rely on a serverless solution.
This option was not only taken into consideration because of its reliability, fault tolerance and ease of implementation, but also because of the availability of some AWS credit provided by the Politecnico di Torino and because of the interest in this novel approach to backend deployment.
The serverless backend has been built through the use of AWS.
Amazon provides services to set up an API gateway to invoke functions, implemented through AWS Lambda, in order to access the Amazon DynamoDB used to store the application data. Amazon Cognito manages authentication and sessions and AWS Amplify deploys the frontend application, combining it with the previously described backend structure. Together with this, Route53 is also used in order to customize the application's domain and that very same domain is furthermore used to send email through the usage of Amazon SES.
For the frontend, the React framework has been chosen because of the previous familiarity with it and for its expressive power. This technology allows creating Single-Page Applications that ensure great manageability thanks to its structures, such as states and effects. Moreover, the creation of web pages and code maintainability is ensured by the usage of Components. Finally, the content of the page can be determined through the usage of React Router, which interprets the url and takes action accordingly. React is also compatible with a wide set of libraries, including survey-react-ui, which is used in order to include SurveyJS surveys in a React project.
This service has been chosen in order to automate the creation of surveys, thanks to its intuitive Survey Creator, available through a web interface. It allows creating surveys with several kinds of questions distributed in pages and with tools able to define a logic path through the questionnaire in order to ask the user all and only the applicable questions.
The Indoor Environmental Comfort survey covers four aspects of comfort: thermal, acoustic, visual and indoor air quality.
In order to ensure a quick survey completion, when a user answers positively about the comfort of his or her environment, the only following question is about which aspects were particularly satisfying. Otherwise, the user is asked to choose which aspects brought him or her discomfort and to answer some specific questions about

them in order to evaluate the corresponding comfort indices.
The survey about personal information covers some aspects about the user's lifestyle in order to be able to understand whether there can be a correlation between routine and perception of comfort.
The first few questions collect demographic information, such as the user's gender, age, nationality and educational qualification. Then, the survey inquires about the context, asking what kind of building it is, the person's role in it and how many people are in the same environment. Secondarily, the questionnaire collects data about the person's lifestyle, asking about impairments, habits and what kind of impact the lack of comfort has on the user's well-being and productivity. Later, a set of questions is presented in order to understand which degree of control the user has on the facilities' systems and the importance that the user gives to this control.
The application's dashboard has the double task of informing the logged users about Indoor Environmental Quality and objective measures and providing useful knowledge for living the environments in a more efficient and conscious way.
In the main page of the dashboard, gauges about Indoor Environmental Quality indices are displayed and, by clicking on any of them, an explicative text and relative measures are shown. These punctual measures can also be seen in graphs representing the piece of data in different time windows and comparisons can be made between different measures or time windows. The "Hints" and "More" buttons provide trivia and additional information about a measure of interest in order to teach the users the impact it has on their lives and how to have a healthier environment.
To remain on a serverless approach, a EC2 machine-based solution was also analyzed.
The requirements of the machines, combined with a very irregular expected traffic and a time-based billing system, makes it so that most of our resources would remain unused for large portions of time. On the other hand, our AWS services, with their pay-per-usage billing system, make it so that we are charged only for the actual usage of systems, resulting in cheaper monthly fees. This analysis has been carried out for both the current expected use case, which is an office of about a dozen users, and for a possible future application, represented by the possibility of deploying the tablets in one of the largest rooms of Politecnico. This would make it so that about 4000 students every day would have the possibility of accessing PROMET&O, implying a rise in requirements and billing, but still generating lower costs than relying on a EC2 machine.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In current times, the interest in buildings efficiency has increased notably. Literature and norms have been created in order to be able to define whether an environment is considered comfortable or not.

The PROMET&O web portal has been born with the goal of building a digital interface in order to collect data about Indoor Environmental Comfort and display said data in conjunction with the ones gathered by the sensors and describing Indoor Environmental Quality.

Indoor Environmental Quality indices alone, in fact, may not be actually representative of the comfort of an environment, as comfort is a subjective perception which varies from person to person. In order to ensure more efficient, healthy environments, the proactive collaboration of the occupants is of crucial importance. This is why great attention has been put into granting an engaging and pleasant user experience.

1.1 Context of application

The PROMET&O web portal is thought to be deployed in a context of reduced size, such as an office, although some of its characteristics make it fit also for larger applications.
In particular, we took into consideration offices where the users can change from day to day and where the amount of simultaneous users is not higher than a dozen people. These workers can be focused on different kinds of jobs: they can work on individual tasks, they can be required to listen to videos and conferences and, sometimes, they're also required to join in.

The amount of people in the room can determine not only a change in the temperature, but also on the sound pressure level, on the amount of surfaces that can cause glare and on the air quality. All these aspects can influence the occupants' quality of life, their comfort and their productivity and are object of the survey the users are proposed to answer.
Together with this, another survey is also presented. This second questionnaire is about the lifestyle of the users and includes information such as the users' age, gender, country, the kind of building they're in and their role in it, their habits and impairments and also which degree of control they have on the facility's systems, including heating, cooling, shading and noise reduction solutions.
All these characteristics are instrumental in having a more complete understanding of the users' perception on the environmental conditions and will be analyzed in correlation with the answers they provided.

A sample environment to be taken into consideration as a use case is an office of about 10-12 seats where the users can change from day to day. In each of these offices, a few tablets are made available as compilation stations in order to allow the users to answer the questionnaires whenever they feel like it and visualize the objective data dashboard. This makes it so that scalability is not a current issue, as a limited amount of data will be forwarded at the same time through the tablets.

As previously mentioned, the users in each room may vary even frequently, this is why it has been deemed fit to introduce two main ways for the compilation of the questionnaire:

- A logged way, where a user with an account has logged in and fills the survey. The answers here received will be linked with the answers to the personal information survey in the name of that user.

- An anonymous way, where the user does not identify himself or herself and will later be asked to fill the personal information survey or to create an account.

In order to sign in, the users are asked to provide an email address and a unique token that they will use for identification.
After signing in, the user will receive a verification link in their email. Without verifying their account, the user won't be able to log in. By clicking the verification link sent, the verification process will be completed, allowing the new account to log in.
The target environment for this application is considered to be a relatively safe area accessible by a limited amount of people. Moreover, it was considered a top priority to be able to complete the survey in the quickest possible way. This is why the login phase has been built more like an identification process than an

authentication. In this step, the user is only required to insert his or her token. This kind of operation is considered less complicated than having to insert into the tablet both email and password each time and sufficient for identifying who is the user who filled the Indoor Environmental Quality survey in order to associate his answers with his personal information and habits. It's not excluded the possibility to provide a more traditional authentication system, depending on the requirements provided by the specific context of application.

1.2 Requirements

The application developed had to be able to:

- Host two surveys, one to assess Indoor Environmental Comfort and one to gather information about the occupants' habits and lifestyle;
- Forward the collected data both to an internal database to store the whole received information and to the Objective Data database after being processed;
- Manage the users, allow them to sign in and identify themselves, allowing Indoor Environmental Comfort survey only through the tablet stations;
- Send email in order to get in touch with the users, not only to verify the account but also to stimulate a proactive approach to the portal;
- Display the Objective Data dashboard, where measurements about Indoor Environmental Comfort are displayed and explained. Information about the values' compliance to norms is displayed and the users have also the chance to compare graphs in different time windows or different indices. The dashboard should also provide additional information and hints about the Indoor Environmental Quality indices in order to bring value through teaching.
- Be available in both Italian and English

1.3 Surveys

The implementation of a feature like a survey can take possibly a long developement time.
Among the required features there were closed-ended questions with single or multiple choices, image pickers, open questions, dropdown menus and, overall consistent and rich customization, in order to meet also the aesthetic requirements.
In order to quicken the development phase it was thought to rely on online tools

used in order to generate surveys, but also on form makers, as surveys and forms are conceptually very similar.
Automated tools to build them both have been analyzed in order to decide which one could possibly fit best our needs: the simple implementation, combined with a wide range of customization were imperative aspects.
In particular, among the proposed solutions, our choice fell on SurveyJS. Their service is built specifically for the automation of surveys, provides a wide set of pre-built questions and systems to manage the questionnaire's flow, it is compatible with React (the environment chosen for the frontend) and allows for thorough personalization through css classes.

1.4 Backend

Implementing the backend is another time-consuming and complex task.
In this stage, we found ourselves before two possible choices: a monolithic, server solution or a distributed, serverless one.
The first choice was of simpler implementation, but the nature of our context made it so that most of our resources would be unused for a very large portion of time, making our investment somehow inefficient. Moreover, a possible future modification of context would force an update on the infrastructure in order to sustain a change in the amount of users.
The latter choice was far more interesting. It was first taken into consideration because of the existence of some AWS credit, made available by the Politecnico di Torino. By analyzing this option in more detail, we discovered that it involved large automation of deployment and part of the development, better resources management through a pay-per-use policy, often joined with a free tier in services use. Among the serverless solutions taken into exam, AWS was chosen.

1.5 Users and device management

Inside the system, authentication is split into two different levels:

- device authentication, through Cognito
- user authentication. through our own APIs and Database

Device authentication is necessary in order to make sure that some of PROMET&O's functionalities, such as filling the Indoor Environmental Comfort survey, are only made available through well-known terminals so as to be able to correlate the subjective data collected with an objective equivalent from the nearest multisensor. Authentication in this case is managed through Cognito, where credentials are created through its graphic interface and forwarded through Cognito's APIs in order to authenticate the device. Moreover, Cognito also manages sessions and the refresh of the jwt tokens used to keep track of said session.

User authentication is instead managed through our own APIs and database, in conjunction with states, cookies and localStorage.
The user inserts an email and a token in order to sign in. Then a confirmation email is sent to the user and, until the user proves to be in control of the email address he or she provided, the account is in a locked state, meaning that the user can't log in.
By clicking on the confirmation link the user received by email, the account is marked as verified and the user can now use it to see his or her personal profile, answer personal questions just once and visit the dashboard.
Information about the currently logged person is also made available through the use of cookies, which Grafana requires in order to authorize graphs visualization for the user.

1.6 Emails

At the moment, PROMET&O uses very few but useful emails.

The first email is sent to the user upon registration: this message has the fundamental role of making sure that the user is actually in control of the mail he or she used to create the account. If it is so, the user shall click on the confirmation link received, in order to show that he was able to receive, access and read the email. Doing so, the account is enabled for use.

The second email instead is received upon completion of the user's first survey. In this communication, PROMET&O's team thanks the user for the effort put in answering the questions and assures support and availability in the future.

Although only a few messages are exchanged, it's thought that in the future new communications may be integrated in order to build a stronger bond with the user, trying to stimulate a more and more proactive approach to the questionnaire in order to obtain as much meaningful data as possible.

1.7 Dashboard

The dashboard is one of the core elements of PROMET&O.
It displays the objective data over time through the use of Grafana's iframes and

queries. Moreover, it compares the Indoor Environmental Quality indices with their subjective counterpart.

The graphs are generated through Grafana, a software built in order to produce different formats of data visualization starting from a database. Our dashboard takes its data from the database managed by professor Montrucchio's team, where the multisensor devices store their measures during the day. Their case of application is quite different from ours, as the objective data database collects new entries every few seconds from a multitude of devices through MQTT. Subjective data is instead collected seldom, with an expected amount of compilations that stands between one or two survey answers per user, per day and only during weekdays and working hours.
This makes the two categories of data significantly different because of their amount and frequency, but being able to put them in correlation allows us to obtain meaningful insights on how the users actually feel about their environment.

1.8 Localization

PROMET&O is available in both English and Italian.
Language management in the application happens through a state variable called "ita" that can be toggled in order to choose between the two languages. This state is then forwarded to all the components in order to let them choose which string to render between the Italian and English version of texts.
SurveyJS, on the other hand, has its own localization tool which makes it very easy to translate surveys and their options. Several languages can be made available and all it takes is to fill the table under the "translation" tab in Survey Creator. Afterwards, set the survey's variable "locale" to the desired language at survey loading and all questions, options and commands will be translated in the chosen language.

Chapter 2

Architecture analysis

2.1 Architecture

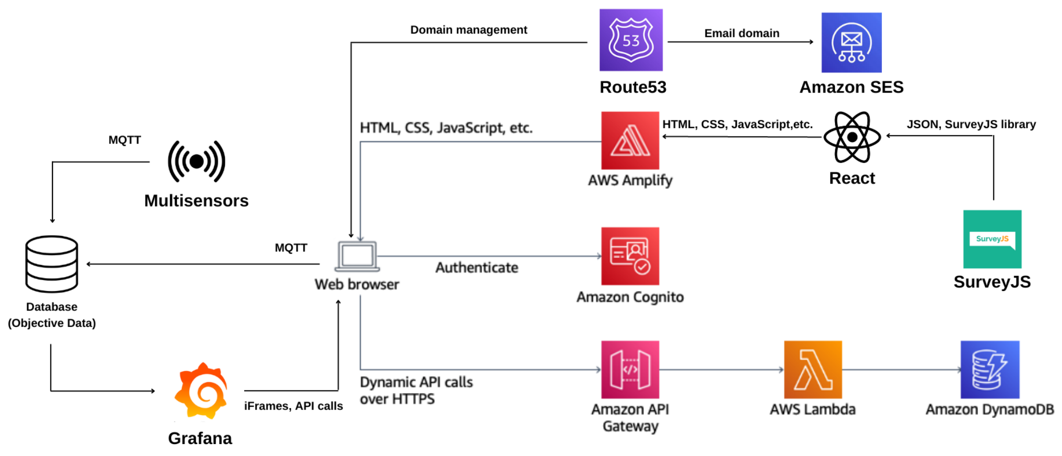


Figure 2.1: Overview of the architecture

The full architecture of the PROMET&O's project includes a wide variety of services and tools.

The objective data acquisition part has been developed by the electronic team in combination with professor Bartolomeo Montrucchio's computer science team. This includes the multisensors infrastructure, which communicates the perceived

data through MQTT protocol to the Objective Data database. Here data is collected and analyzed by Grafana, a software used to produce dashboards containing graphs and other data visualization tools.
Grafana's iframes and query results are embedded in the PROMET&O web portal in order to make this information easily available to the users.
The web application also sends some MQTT data to the Objective Data database. The data sent is about the comfort indices evaluated through the users' answers to the Indoor Environmental Comfort survey and is displayed alongside the expected comfort indices obtained from the corresponding objective data.

The subjective data acquisition part instead is represented by a React application supported by an AWS serverless backend. SurveyJS, our surveys automation tool of choice, has also been chosen because of its compatibility with the React framework.
SurveyJS is able to represent questionnaires through the usage of a JSON file embedding all of the survey's characteristics and its library, where functions can be found in order to start the questionnaire, customize the css classes of some aspects of the questions, choose how the survey is supposed to act upon completion and selecting the language.
This tool has been used twice, the first time to implement the Indoor Environmental Comfort survey and the second time to implement the questionnaire about the users' lifestyle and habits.
The serverless backend, instead, has been built through the use of AWS.
Amazon provides services to set up an API gateway[1] to invoke functions, implemented through AWS Lambda[2], in order to access the Amazon DynamoDB[3] used to store the application data. Amazon Cognito[4] manages the device authentication and AWS Amplify deploys the frontend application, combining it with the previously described backend structure. Together with this, Route53[5] is also used in order to customize the application's domain and that very same domain is furthermore used to send email through the usage of Amazon SES[6].

The chosen solutions were not the only available options. In the next few sections, other possibilities are described, together with the reason of our final choices.

2.2 Backend

The backend architecutre represents an interesting aspect to which we devoted special attention.
Several options are made available, from a simple, monolith architecture to a more elaborated distributed infrastructure. The serverless option was particularly

interesting: it represents an innovative solution, granting also availability, reliability, automation of some development tasks and, possibly, a scalability which would allow the infrastructure to adapt if different use cases were to arise.
Several choices have been analyzed and the results of our considerations can be found in the next subsections.

2.2.1 Comparison between server and serverless solution

One of the most common approaches, when developing your own web application, is implementing a server architecture containing all services to allow your system to run properly.
In our specific case we would have required:

- a web server for the application
- a database to store and retrieve data
- a mail server to communicate with our users
- a hosting service to make our application reachable

This architecture can either be monolithic or distributed.

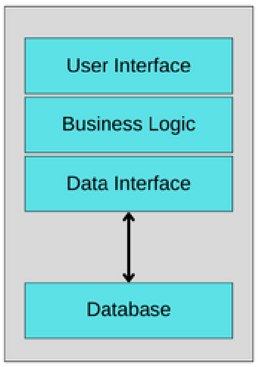


Figure 2.2: Example of a monolithic architecture

The simplest scenario, in terms of implementation, is the monolithic one, which consists of a single machine where all services are deployed.
In this case we have a simpler deployment at the expense of huge limitations related to the availability of our application. Our server, in fact, would become a single point of failure, meaning that if our machine had some issues the entire website

would become completely unavailable. Also, if our requirements were to change in the future, the system would be able to accommodate this growth only up to a certain point, after which the resources of the machine would not satisfy the application requirements.

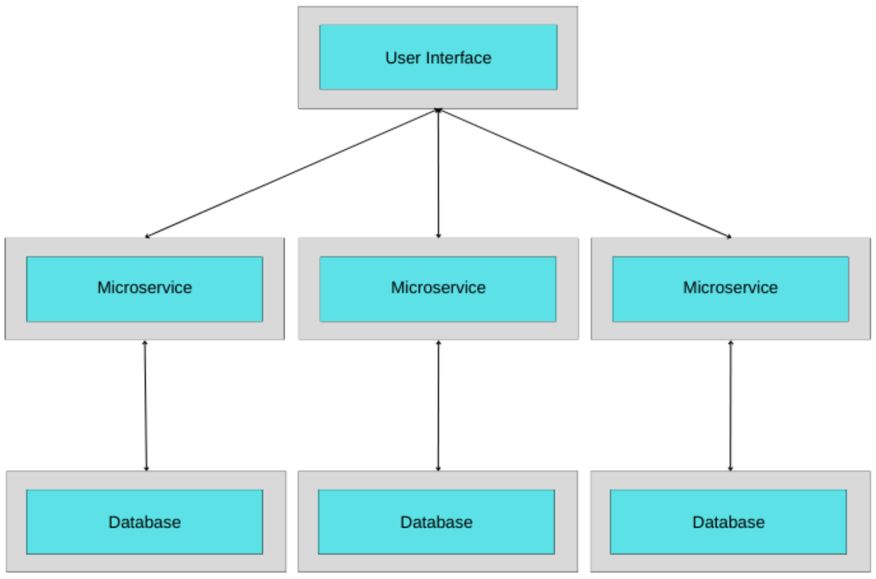


Figure 2.3: Example of a distributed architecture

In a distributed architecture, components are deployed on different machines, which communicate via network in order to achieve their common goal. This makes this architecture much more flexible, allowing resources to be shared, hence allowing higher efficiency through concurrency and a higher fault tolerance.
On a negative note, it is much more complex to design, maintain and secure, and also more expensive since it requires a higher amount of resources and synchronization among all devices.

Both these solutions require a large portion of manual configuration and are not fitting in a scenario with a rapidly changing traffic flow. There could be moments where the traffic is very low, leaving a great part of our resources unused, for example at nighttime, but also moments where the requirements are much higher, for example around lunchtime, with the risk that our infrastructure cannot accommodate them all.
Moreover, this infrastructure requires security, in order to limit as much as possible all cases of attacks to the platform and the data in it. Some hazards can be represented by DoS, SQL injection, Ransomware or Spyware.

The task is therefore non-trivial, but an alternative solution does exist.
Especially in order to satisfy the varying requirements that our application can have, several serverless options have been developed. These solutions use servers made available by companies that manage entire data centers and that, at a price based on a limited time subscription or a per-usage fee, guarantee a solid infrastructure able to satisfy varying traffic requirements and which can host one or more of our services.
The Politecnico di Torino, for example, made available some credit for Amazon AWS, which provides an extensive pool of services to allow us to build a completely serverless application. The unity among AWS' modules makes it so that an additional level of security can be added, granting protection at service and application level alike. Each of these services is built in order to scale automatically based on the received amount of requests, granting availability for every component regardless of traffic and costs related only to how much was actually consumed. In the next paragraph, we will examine some potential serverless environments.

2.2.2 Comparison between potential serverless environments

The diffusion of microservices generated a wide set of options about ready-made serverless backend. This allows the use of an almost standard infrastructure in order to be able to immediately support our frontend. In our application's scenario data collection is particularly interesting. Non-relational databases fit our needs very well, since they can store data whose structure can change over time, still remaining related. A relational option would still be applicable, but would force some restrictions on data structures.

The usage of Google Sheets[7] as a relational database can seem a bit naive, but its ease of implementation makes it particularly noteworthy. Through the use of Google APIs it is possible to transcript data into a worksheet and retrieve them afterwards as if they were rows in a relational database table. The usage of name conventions and a small amount of scripting turns any Google Sheet into a small database. Among its advantages, it's a completely free solution and it doesn't require any cloud management. Its main disadvantage is a much more limited nature, compared to other databases.

CouchDB[8] is a non-relational, JSON based database. Its main feature is the ease of use and its replication system, which keeps data in an eventually consistent state, relying heavily on local data. The lack of locks grants great data accessibility and is the foundation of the concept of eventual consistency previously mentioned.

The introduction of a special table to keep track of users and relative authorizations makes it so that an additional level of security can be added to single requests. The system is based on a paradigm called cURL, which manages queries in a way similar to HTTP requests. Another interesting characteristic of CouchDB is its open source and free nature. A downside of this solution is that deployment has to be handled manually and at least a virtual machine is required for the database to run.

NoSQL[9] is a tool offered by Oracle. It's based on tables and key-value relations and, also in this case, scalability and speed in managing data in a flexible way are its strong points.
In its Always Free version NoSQL offers 133 millions reading and an equal number of writings, with a total space of 25GB per table, up to three tables. This could be limiting in case of complex data structures. Always Free version, moreover is only available for the US West (Phoenix) region. For other regions a payment tier is required, thus the free version is not available in Europe.

IBM Cloudant[10] is an open source implementation of CouchDB offered by IBM. Together with the advantages of CouchDB's structure, the company relies on its long experience in the field, granting reliability, security and consistency, with particular care for innovation. Their Hybrid Cloud strategy grants the possibility of administering different levels of the cloud infrastructure from a single point of contact. The free version of this service provides 1GB of storage and up to 20 readings and 10 writings per second. This could be enough for a context of reduced size, but upscaling the system might require a payment version.

Couchbase[11] promotes especially its learning curve, supported by its extensive documentation and the usage of SQL language, saving data in JSON format, which grants high flexibility. The use of integrated cache, moreover, grants speed in data interaction.
Its architecture is masterless and based on automated data replication. Its load distribution across the infrastructure makes it so that no shutdown is required in case of modifications or updates, making of data availability and response speed its spearhead.
Couchbase is provided in an implementation called Couchbase Capella, a Database As A Service which grants a simple and immediate management of the database, automatically managing its setup, synchronization and replication. These services are provided in a free trial of 30 days with 50 GB of storage, making it a non-viable long term solution.

JSON serverless[12] is a github solution oriented to create a serverless system for JSON files management through REST API, relying on AWS cloud. Through the use of appropriate commands, it's possible to create a folder with an already deployable serverless configuration. It also allows adding personalized middleware and authentication systems. Being available on git, the solution seems to be free, but relying on AWS the costs would stem from the usage of Amazon infrastructure.

Datastrax[13] references the Kubernates environment and provides an open source solution called AstraDB.
The purpose of AstraDB is drastically reducing deployment time, simplifying the interactions with Cassandra through a pay-as-you-go serverless service.
The solution is able to operate natively with JSON and GraphQL and provides API able to query Cassandra's underlying structure in a transparent way, unrelated to its query language.
The free version provides 80GB of storage space and 20 millions operations per second, becoming a very competitive option. The solution can be deployed on AWS, Google Cloud or Azure. AWS is characterized by the previously mentioned pricing limits. Azure provides free trials for 750 hours of usage. Google Cloud provides a 90 days trial.

Restdb.io[14] provides access to data through HTTP requests to a JSON file, creating schema and its relations directly onto the web browser and allowing access through REST API.
Access to data is allowed through a huge variety of devices, allowing to read, create and update data from virtually anywhere. It also provides access control systems in order to limit interventions on data based on roles, random data generators and automated creation of forms. In its free version, Restdb.io provides at most three users, 2.500 records 1 API call per second ad 100MB of storage, resulting in one of the most limiting plans.

MongoDB Atlas[15] is a NoSQL system compatible with most available hosting services, which wants to provide a robust, flexible and scalable service.
It provides a free version which includes 512MB of storage, shared RAM and all services required to access REST APIs. The free version, in particular, provides access to three European regions for the same number of deployment platforms: Frankfurt for AWS, Belgium for GCP, Netherlands for Azure.
Clever Cloud provides 500 MB for free, Object Rocket and Scale Grid have a 30 days trial each while Digital Ocean has a 60 days trial. All of these services support MongoDB.

The Internet thus provides a wide range of services. The additional functionalities abound and are distributed in different bundles among different providers.

About what it takes to manage a simple database, all offers convert towards a NoSQL distributed model, managed through JSON with particular care to data availability and consistency, even eventual.
Surprising, in this scenario is the role of Google Sheets, which provides an extremely simple and free service, although very basic.
These solutions, though, only cover the data management scope, while our interest is towards a fully serverless solution. Although these options are interesting to analyze, we could go back to exploring a solution that is represented by DynamoDB and all AWS services.

2.2.3 Conclusions

Initially, both a server and a serverless approach have been analyzed.
Particular interest has been put in the latter, because of the novelty in the approach to deployment and because of the availability of some AWS credit.

We took into consideration a wide set of options, such as Google Sheets, CouchDB, NoSQL, IBM Cloudant, Couchbase, JSON serverless, Datastrax, Restdb.io and MongoDB Atlas, but ultimately our interest went back to AWS.
AWS, albeit being initially set aside because of the absence of a totally free solution, was in fact not completely excluded, because of the availability of some university-provided credit.
In our analysis we noticed that deployment is a complex task that might require a lot of resources. Free applications often still rely on larger infrastructures, like AWS itself, while the paid versions often provide a free version with huge limitations.
Amazon's scenario became then newly interesting, since it provides a wide range of services related to every aspect we're interested in, granting unmatched integration among them and thorough documentation.

2.3 Surveys

In order to speed up the development phase as much as possible, one of the initial steps of research was to automatize the production of some aspects of our code. A core element which was suitable to be computerized was the surveys structure.
We considered that the most complex and automatable part could be represented by the survey, so different tools were analyzed in order to produce our questionnaires in a quick way, without sacrificing the aspects of customization, a requirement which could not be left out. Another characteristic which would be favourable is compatibility with the React framework, which has been chosen as a frontend

substructure because of the previous familiarity with it, together with the efficiency of the Single-Page Application it produces. Among the possible options for an easy implementation of our survey, two main ways were taken into consideration: survey tools and form tools.

2.3.1 Survey Creation Tools

The Internet offers a wide range of possibilities to create and fill surveys with the most various structures. The greatest part of them allow even the most inexperienced users to operate through the great intuitiveness of their interfaces and high customization.
Among the existing services, we examined LimeSurveys, SurveyJS and Google Forms.

LimeSurveys[16] is the most business-oriented among the proposed solutions. The free version allows users to receive up to 25 answers per month, but provides a wide range of possible question types, tailoring tools, API access and the possibility to export data in various formats, such as csv, excel, word or pdf.
Paid tiers, instead, offer a higher amount of answers per month, widened storage space and the chance to have white-label pages and domains and email support.
Lime allows both the installation of the hosting application on the user's own machine or cloud solutions.
The surveys creation is code-based and requires a higher preliminary knowledge than other solutions, but it also offers a vast variety in questions types (around 30) and an almost complete customization.
Questions can be gathered in groups, validation functions can be introduced, questions can be marked as mandatory or optional and relevance points can be assigned to every question. LimeSurvey also makes it possible, through a scripting language, to adapt questions and relative options based on the previously answered questions.
Although it's possible to introduce JavaScripts fragments inside questions, it doesn't seem possible to export the questionnaire itself. At most, it's possible to embed it inside a page and extract its results.

SurveyJS[17], instead, allows creating surveys with the aim of including them inside other pages, merging with the destination page's look. It also allows to store both the survey and relative answers inside the user's own servers and to extend it through the use of third-party widgets.
SurveyJS also allows the survey's creation in a very intuitive way, thanks to its tool SuvreyCreator, which provides the possibility of creating questionnaires made of up to 20 different kinds of questions and, also in this case, the chance of clustering them in groups (here classified as pages). There is also the possibility of including

testing-like features, such as a maximum number of errors or time duration, thus allowing for the creation of quizzes.
The creator produces the survey's JavaScript and HTML code, also granting the possibility of modifications on code, making it a good solution for experienced and newbie developers alike. It's also provided with a JSON editor which allows it to represent the survey in that format, too.
SurveyJS is free for non-commercial use, in different cases it requires the purchase of one or more licenses. For commercial use, in fact, the free version only allows access to SurveyJS Library, which grants the chance of integrating the survey in any application able to host JavaScript code. To have access to SurveyCreator, Analytics and PDF, the corresponding paid bundles are available, together with one additional cluster which includes them all.

Google Forms[18] is probably the solution with the most intuitive interface. Also in this case, it's possible to create forms through the use of a visual interface, choosing among different kinds of questions, grouping them in sections, defining the validation details and collecting and analyzing the obtained results. This platform moreover allows different users to work on the survey collaboratively, coherently with all other Google services. In this case, the possible question types are only 11 and the customization is more limited. Google, in fact, hides most of the code from the user granting usability even for the most inexperienced users but limiting the possibilities of more expert ones. Through settings it's possible to turn the survey into a quiz with automatic or manual correction, to manage how many answers per user can be provided and decide some visualization options. The only way to integrate said surveys inside a web page is through the HTML tag provided by Google itself.
Despite the limited customization, Google forms is still a much appreciated choice, even by companies, due to the fact that it's available for free even for commercial use and allows to receive and analyze, also for free, up to 5 million answers distributed among all questions. This means that in a 10 questions survey it would be able to collect at most 500 thousands answers. All received responses can be exported in a Google sheets document, be it existing or created ad-hoc, which will also be updated as new answers will be received.

Among the analyzed solutions, we could notice a wide range of services. Depending on the application context, it's possible to choose a more professional, customizable but also pricier and complex service, like LimeSurvey, or turn to a more simple but also limiting solution like Google Form. SurveyJS seems to be a good compromise, granting both a simple and intuitive visual interface and the possibility of intervening in the code for customization. All examined systems allow to validate answers and export the obtained data, providing also internal analytics tools to evaluate the responses' distribution and obtain an at-a-glance idea on the participants' opinions.

2.3.2 Forms Creation Tools

Among the analyzed tools for low-code forms, instead, we analyzed FormFlows.ai, Form.io, Zoho Forms.

FormFlows.ai[19] is an open source, low-code platform for business-oriented modules creation. The form is built through an intuitive drag-and-drop interface. It's possible to introduce validation conditions for each field and, moreover, there is a chance to build a BPMN-like graph to drive the user through a customizable path. This allows creating routes which manage all and only the sections of actual interest. FormFlows also provides tools for data collection and analysis through the use of dashboards, in order to transmit the meaning of received information also visually. Developers promise a perfect level of integration with process-automation tools and the use of specific frameworks for higher security. It's possible to implement said service on cloud servers, but it doesn't seem possible to do so on own servers. This solution also introduces aspects of Sentiment Analysis and natural-language recognition, together with tools for simple and fast development. Being business-oriented, it also provides a set of additional tools such as Quick Starter Kit for rapid prototyping, counseling services, formation and technical assistance.

Form.io[20] also has a typically business-oriented imprinting and makes a spearhead of its serverless nature. It allows creating multi language forms, both starting from scratch and from a set of templates. Also in this case, modules creation is assisted by an intuitive drag-and-drop system to locate and define fields and relative meaning. It grants customization also in terms of module views and the chance to introduce a potentially unlimited amount of trigger actions for automation. The created form can then be included in any javascript page thanks to the generated tag. This also grants the possibility of manipulating it through CSS or accessing the REST API provided with the module. Compatibility with Angular and React is also assured, together with the possibility of using webhooks and websockets to interact with the most common set of online services like Office365 and Google docs. Data can also be managed through the provided set of API and forms can also be distributed offline.
The nature of this solution is based on two cornerstones: the chance of using Docker containers to implement the service also on a private cloud and the concept of Deploy Anywhere through Cordova, Electron and other services.

Zoho Forms[21] shares several characteristics with the previously analyzed solutions. It also provides a drag-and-drop system to build multi page forms with a wide range of possible input fields, validation tools and the possibility of defining paths through the use of skip logic, which makes only the desired fields visible to the user based on the previously provided answers, together with the possibility of saving and resuming the form filling in a later moment.
Tools to communicate the positive outcome to the user are also provided, through redirection, mail, SMS or even Microsoft teams.
Forms can be shared through links, email or embedded in a web page. Possibility of aesthetic configuration is also provided, from domains to branding, and responsiveness is also granted.
Analytics tools allow the tabular visualization of collected data and the creation of reports and documents, together with the chance of producing meta-data analysis. Also in this case, automation is possible, for example to contact users based on certain answers or further validate the received answers.
A peculiar feature is the chance of integrating payment systems, which allow transferring money through the form itself in a safe way, producing receipts and giving the chance of communicating the positive outcome of the operation to the user through email.
Forms may also be filled offline. Special fields, such as geolocalization, code scans, signatures are provided, together with the chance of putting the device in kiosk mode, particularly useful in contexts where the form's device is accessible to the public. Also in this case, security features are provided, for example Captcha, SSL and GDPR compliance.

Other than the previously analyzed solution, on the Internet a wide range of similar services are available, almost identical except for minor differences due to the application context.
We can find, for example, extremely data-oriented solutions like Budibase[22], which make its spearhead off its connectivity to the most common databases, or Jotform[23], more decisively oriented to payment systems.
Some systems require some coding abilities, such as Alpaca[24], or are new projects which still require some time to reach its competitors' level, such as OhMyForm[25].
Alternatives are therefore very diverse also in this sector and each are based on some cornerstones which are extremely precious in the current context, such as the chance of developing said modules with little to no code, modes to define the sequence of questions based on previous answers and tools to analyze the collected data.
In particular, FormFlows and Form.io stand out in these aspects.
The first one has the advantage of providing a visual dashboard for data distribution but also has the disadvantage of allowing the deployment to occur only on public,

on-premise cloud providers. The second one provides a less intuitive, tabular data visualization, but allows data interaction through an automatically generated collection of APIs and the possibility of deploying the service in a combination of servers and containers.
Finally, ZohoForms embeds the ease of use of the first two solutions, tabular data visualization and a more commercial imprinting, providing tools for payments management and GDPR compliance.

2.3.3 Conclusions

The usage of forms, such as FormFlows.ai, Form.io and Zoho Forms, was taken into account as an alternative to survey tools, but ultimately it might as well been considered as a secondary choice because of the more fitting characteristics of the first option.
In terms of surveys, we also analyzed LimeSurveys and Google Forms, but the most appropriate solution was deemed to be SurveyJS, for the specificity of its application, the large possibilities of customization and the ease of use
In particular, it was taken in high consideration the management of the survey's creation through visual interface and customization possibilities, the wide variety of possible questions to be added and the tools to split the survey in different pages and browse through them without the need for additional code.
Moreover, the presence of APIs to store the received data in our own database and libraries to inject additional personalization into our questions was positively noted.
Finally, the chance to slightly alter the questionnaire's questions and answers through its JSON makes it so that SurveyJS represented the ideal candidate for our context of application.

Our analysis on Forms, nevertheless, was not completely ignored.
The products analyzed were taken into consideration in order to be used for the implementation of the simple forms which are required throughout the application in order to allow users to sign in and log in.
The complexity of these new solutions, though, was considered as too much of an overhead for the simple application cases which were being analyzed.
We chose to use Formik[26], an outsider in this context, but well known for previously delivered projects. This tool is not as refined as its competitors: it doesn't allow for a serverless data collection and requires code, although little, to be manually written by the user.
The ease of implementation of this library, together with the previously mentioned familiarity and the lack of costs, were decisive aspects in the choice of this solution.

Chapter 3

AWS Services

AWS (Amazon Web Services) is a wide backend serverless environment equipped with a rich pool of services. The perfect integration among them and the wide range of covered topics is one of the reasons why we chose this environment. Many of these services are characterized by free usage thresholds that limit the expense both for the usage of serverless backend and for the frontend's deployment.
The usage of these technologies is perfectly integrated with JavaScript and particularly with React, chosen for the frontend because it grants the possibility of creating fast, dynamic pages and for the familiarity with it.
Amplify CLI (Command Line Interface) allows to set up all backend functionalities by command line, from table creations into the database to management of user's authentication, going through API's creation.
Together with command line commands, there is also a web-based console that allows to perform initialization and maintenance operations, together with monitoring costs and resource usage.
All AWS Services are built to dimension the assigned amount of resources based exclusively on actual usage, granting automatic scalability with increase or decrease of active user base.

3.1 Application's architecture

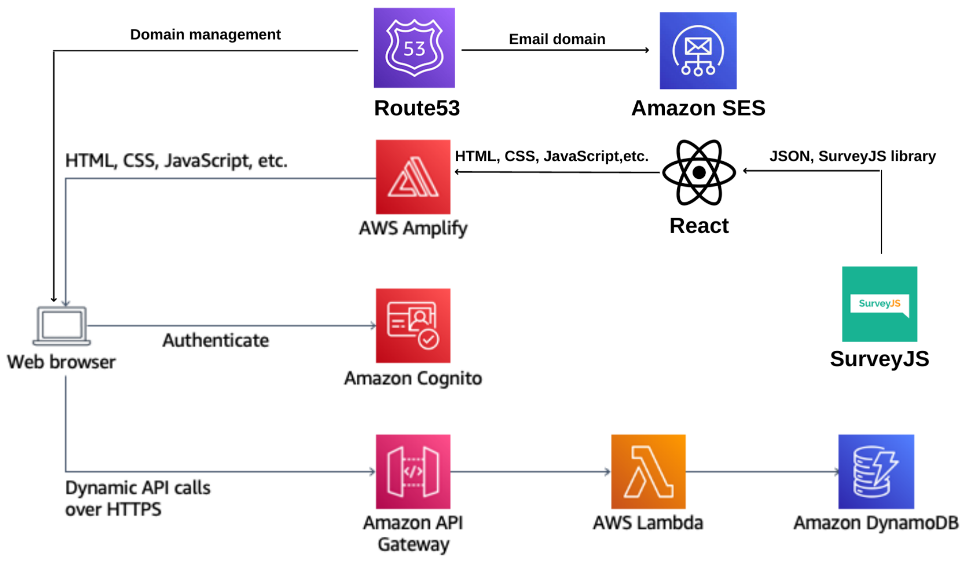


Figure 3.1: Overview of the AWS architecture

The image is a high level representation of the architecture in use.
Starting from the bottom, we notice the trio composed of Amazon API Gateway, AWS Lambda and Amazon DynamoDB.
These three services allow managing HTTP requests in a completely serverless and mostly automated way. Through the Amplify CLI it's possible to create a new API, specifying the path where we want it to be available. Then we can choose which function has to be invoked at a request reception on that path, selecting it from the available functions in AWS Lambda or creating a new one.
Amplify CLI also allows to create Lambda functions with a predefined structure, for example in order to execute CRUD operations (Create, Read, Update, Delete) on a DynamoDB table. DynamoDB is a non-relational database composed of key-value pairs grouped in items, which stores and retrieves our application's data.

Although single users' sessions get managed through ad hoc API because of the peculiar mode of user recognition, devices' sessions are managed through Amazon Cognito. Cognito provides a set of APIs that manage users, grouped in User Pools, allowing their creation, the storage of their credentials and management of sessions. This allows to authenticate devices sending requests to Amazon API Gateway, granting only to authorized users the possibility of sending requests to specific

paths.
API authorization can also be achieved through Lambda functions implementing the authentication logic.

The Frontend is managed by AWS Amplify, which loads it and makes it visible through an amazon-provided address. The address can also be substituted, just like we did, with a dedicated domain. Said domains are created with the use of Route53 and are also used in order to send email through Amazon Simple Email Service.
Amplify also builds the backend, making it accessible by frontend calls.

3.1.1 Amplify CLI

The Amplify CLI[27] is a tool of utmost importance to manage the backend environment in a simple, intuitive way from the user's command line.

The installation process starts from the Amazon Website, where it's required that the user creates his or her account in the first place. When this step is completed, we can move to the command line.
Amplify CLI is made available through npm, so in order to download it I typed the command:

```
npm install -g @aws-amplify/cli
```

Then I was required to complete the Amplify CLI configuration, which can be prompted by the command

```
amplify configure
```

Here the command line opens an AWS web page in order to allow the user to log into the previously created account.
Once done so, I returned to the terminal and I was asked to choose my region of reference through a menu. I initially chose us-east-1 as it was a default region and had lower usage costs, but later in development the backend was moved to a European region for reasons of GDPR compliance. This is why afterwards it was chosen eu-west-3 (Paris).
Once chosen the region, it is asked to choose a username for a new IAM user. This kind of user is a so-called service account and is stored in a service called IAM. It represents a user or application enabled to make AWS requests. In order to be allowed to do so, the IAM user has associated credentials and permissions.
This prompts the opening of another AWS web page in order to complete the creation of the IAM credentials. I confirmed the username and left the AWS access type as the default "Programmatic access" value. Moving on to permissions, I left

the default policy for Administrator Access and confirmed the previous settings. The creation of a user brings to the generation of two credentials strings, Access Key ID and Secret Access Key, which is wise to annotate, as they can be required in the future and won't be displayed again.
Right afterwards, in fact, the command line requires said credentials in order to bind the terminal to an IAM user and be able to execute commands on AWS.
After inserting Access Key ID and Secret Access Key, I kept following amplify configure's instructions and chose a Profile name for the AWS Profile on my local machine. The user set up is now complete.

Once installed the Amplify CLI and completed its configuration, it's possible to create resources deployable in the backend straight from command line, through a multiple choice path driven by aws.
After accessing my React app's folder, I start the initialization of the amplify environment through the command:

```
amplify init
```

Here it's possible to define the project's characteristics. The command line asks to define:

- the project's name: Paris, as it was my first project in the Paris region
- the environment's name: sampledev
- the default editor: none, as I use webStorm and it wasn't among the options
- the type of app I'm building: JavaScript
- the framework of choice: React

Then I was asked to choose source directory path, distribution directory path, build command and start command and for all these settings I kept their default value.
Then the command line asks what kind of authentication to use between Access keys formerly generated and AWS Profile. I chose the first option and inserted the previously created credentials.
This initializes the backend cloud part and the command line suggests to create a new api through the use of the command "amplify add api".
Once completed these operations, it's required to associate an existing user pool to the application or create a new one through the command

```
amplify add auth
```

As I had already created a UserPool before through Cognito's web based interface, I simply bound it using the command

```
amplify import auth
```

and selecting my User Pool, SurveyAdmin, from the provided list.

At the end of this step, the command

```
amplify push
```

allows us to upload all local modifications to the cloud. After displaying the state of the cloud before the modification, a confirmation is asked and, upon selecting "yes", the local data is uploaded to the cloud.

Once embedded Cognito to the frontend to manage users' authentications, it's possible to move on to the creation of APIs, which will later be invoked to interact with the backend.
APIs can be created through the command

```
amplify add api
```

This is followed by a set of choices which the user has to take about every aspect of the resource he's creating.
For example, in order to create the set of resources used to save the data about the app's users, I operated as follows:

- I was asked to select what kind of API I desired, so I chose REST APIs
- then I entered the API name: userTokenAPI
- later I chose a name for the path: /token
- afterwards I was asked if I wanted to use an existing Lambda function or if I wanted to create a new one. I chose the latter.
- I was asked to provide a friendly name for the function in local: userToken-Lambda
- I kept the same name for the Lambda as well: userTokenLambda
- I chose NodeJS as the function runtime
- When asked what kind of template I wanted to use for the Lambda, I picked: CRUD function for DynamoDB (Integration with API Gateway)
- As the previous setting regarded a DynamoDB table, I was also asked whether to use an existing DynamoDB table or to create a new one. I chose the second option.
- Once again, I was asked to pick a name for the local resource. I chose userToken

- I used the same name, userToken, also for the newly created table
  - In the new table, I was asked to name the first column: email
  - I chose "string" as its data type
  - I chose to add another column and I repeated the two previous steps for the column "token", which is also a string
  - I didn't add any more columns as these were the only required ones at the time. It's still possible to create new columns by specifying them as attributes of an object stored into the database. These new columns won't be mandatory for other objects, while the ones created previously are.
  - I was then asked to choose a partition key for the table, I chose the email
  - I was asked if I wanted to have a sort key, I refused
  - I also refused the creation of secondary indices and triggers, as they were not required
- I was asked if the resource was required to access other resources in the same project and I answered no
- I was also asked if I wanted to invoke the new Lambda on a recurring schedule and, being this not the case, I answered no as well
- I was offered the possibility of editing the function in local on the spot and I answered yes, as I required to slightly edit the code in order to filter the data properly upon specific requests (for example, receiving only the information about a specific user chosen by email instead of receiving all user data available)
- I was then asked if I wanted to limit the access to the newly created resource. I refused as I preferred managing this step through the API Gateway web interface

Later, the user is asked if he wants to create another path and the entire process repeats in order to create all of the required resources. When all our modifications are complete, amplify push sends everything to the cloud to make it effective.

3.1.2 API Gateway

Amazon API Gateway is an endpoint that allows any application (React, in our case) to execute on demand functions present on AWS Lambda.
API Gateway allows to create the desired paths and, for each of them, to deploy resources corresponding to the most common HTTP operations, such as GET, POST,

PUT and DELETE. This makes it so that, in response to an HTTP request of a certain kind, the corresponding Lambda function is invoked. The function will take the content of the request and elaborate a response that can be returned to the user.

Each of these resources can be protected by individual policies, granting high granularity in the definition of security requirements for each request kind.
In particular, Amazon API Gateway relies on a tool called Authenticators in order to enact these security policies.
At the moment we have two Authenticators in use, a Cognito and a Lambda one. The Cognito one is used on a POST request performed on the path /survey, as we don't want that the users are allowed to post answers from devices which have not been authenticated by Cognito. To enforce this, we select our API, userTokenAPI and, under the Authorizer tab we selected "Create a new Authorizer". We named it "Authorization" and we created it as a Cognito Authorizer. We selected "SurveyAdmin" as our reference User Pool and we used "Authorization" as our Token source, as our token will be stored in the Authorization field of the request.
The Lambda authorizer is used on GETs, especially on /survey on /personal and on /token, in order to make sure that their content can only be retrieved by authenticated users. In the login phase, in fact, every user receives a JWT token which gets forwarded during said requests in the Authorization header. The Lambda Authorizer is a lambda function, in our case "jwt_verify", which performs a check on the jwt in order to see if the signature is valid and if the issuer is "prometeo.click". In case of failure it returns an "Unauthorized" error and the request is not inspected any further. In case of success, a policy document allowing the user to execute the function is generated, returning a principalId which will later be used to authorize the execution.
After creating this function, we select the API Gateway resources that we want to secure. Also in this case, the authorizer is set up under the Authorizer tab of API Gateway. We selected "Create a new Authorizer", named it "LambdaAuthorization" and we created it as a Lambda Authorizer, referencing the aforementioned lambda function "jwt_verify". We used "Authorization" as our Token source, as our token will be stored in the Authorization field of the request and saved.
After creating the authorizers, we go back to the Resources and select the ones we want to protect, for example /survey. In our case, ANY method uses the Cognito authorizer, with the exception of the GET, which uses the LambdaAuthorizer. In /survey, we click on ANY and from there onto Method Request. Under Authorization, we pick the Authorization Cognito user pool authorizer we created previously and we select the tick mark to confirm our choice. We repeat the same operation under the GET method, selecting instead the Lambda Authorizer.
Then, under the Actions tab, we select "Deploy API" to make our changes effective.

The resources made available in the project are the following:
userTokenAPI:
Invoke URL: https://bp5j5v8p87.execute-api.eu-west-3.amazonaws.com/sampledev

- **/personal**: the management of the answers provided to the personal questions' survey. Refers an automatically created CRUD Lambda function "personal-Lambda" which operates on the "Personal" DynamodDB table
  - **GET**: /personal/personalID?user=[email]
    description: obtain all answers provided to the personal question's survey by the user with the provided email
    Note: this can only be done through the use of the user jwt stored in the state variable userJwt and generated at login
    usage: Personal.js
    response header:

    ```
    {
        "Authorization":[userJwt]
    }
    ```

    response body:

    ```
    [
        {"personalID":"response30026644",
        "Q1":"Female",
        "Q2":"18-25",
        "Q3":"Italy",
        ...
        "timestamp":1666879284540,
        "user":"[email]",
        }, ...
    ]
    ```

  - **POST**: /personal
    description: store the provided set of answers into the database
    usage: Personal.js
    request body:

    ```
    {"Q1":"Male",
    "user":"anon12345678",
    "Q2":"65+",
    "personalID":"response12345678",
    "timestamp":1666019227587}
    ```

    response body:

```
{"success": "post call succeed!",
"url": "/personal",
"data": {}}
```

- **/survey**: the management of the answers provided to the Indoor Environmental Comfort survey. Refers an automatically created CRUD Lambda function "surveyLambda" which operates on the "SurveyResult" DynamodDB table
  - **GET**: /survey/user?user=[email]
    description: obtain all answers provided to the Indoor Environmental Comfort survey by the user with the selected email
    Note: this can only be done through the use of the user jwt stored in the state variable userJwt and generated at login
    usage: Personal.js
    response header:

    ```
    {
        "Authorization":[userJwt]
    }
    ```

    response body:

    ```
    [
        {"Q1":"1",
        "resultID":"30e1e2c1-a73f-421f-8c2c-1f4e8c9069d3",
        "user":"[email]",
        "multisensor":"test",
        "timestamp":1666269817279},...
    ]
    ```

  - **POST**: /survey
    description: save the answer provided to the Indoor Environmental Comfort survey
    Note: this can only be done through the use of the device jwt stored in the state variable deviceJwt in authenticated devices
    usage: FurtherQuestions.js, Thanks.js
    request body:

    ```
    {"Q1":"1",
    "resultID":"30e1e2c1-a73f-421f-8c2c-1f4e8c9069d3",
    "user":"[email]",
    "multisensor":"test",
    "timestamp":1666269817279}
    ```

    request headers:

```
{
    "Authorization":[device jwt]
}
```

response body:

```
{"data": {},
"success": "post call succeed!",
"url": "/survey"}
```

- **/token**: the management of users' data. Refers an automatically created CRUD Lambda function "userTokenLambda" which operates on the "userToken" DynamodDB table
  - **GET**: /token/email?token=[token]
    description: obtain all data about the user with said token
    Note: this can only be done through the use of the user jwt stored in the state variable userJwt and generated at login
    usage:
    request header:

    ```
    {
        "Authorization":[userJwt]
    }
    ```

    response body:

    ```
    [
        {"active": true,
        "code": "ADF6F8",
        "email":"reloser232@sopulit.com",
        "subscribed": true,
        "token": "[token]"}
    ]
    ```
  - **GET**: /token/object
    description: obtain all data about all users
    Note: this can only be done through the use of the user jwt stored in the state variable userJwt and generated at login
    usage:
    request header:

    ```
    {
        "Authorization":[userJwt]
    }
    ```

response body:

```
[
    {"active": true,
    "code": "ADF6F8",
    "email":"reloser232@sopulit.com",
    "subscribed": true,
    "token": "[token]"},
    ...
]
```

- **POST**: /token
  description: save data about a new user
  usage: CreateAccount.js
  request body:

  ```
  {"email":"goham41933@lance7.com",
  "token":"goham41933",
  "code":"RZ05U6",
  "active":false,
  "subscribed":true}
  ```

  response body:

  ```
  {"success":"post call succeed!",
  "url":"/token",
  "data":{}}
  ```

- **POST**: /token
  description: if the code information is not present, it's used to perform the login
  usage: Login.js, Verification.js
  request body:

  ```
  {"token":"goham41933"}
  ```

  response body:

  ```
  {"email":[user email],
  "token":"[user token]",
  "jwt":[user jwt]}
  ```

- **PUT**: /token
  description:update a user's information, for example upon email verification
  usage: Verification.js
  request body:

```
{"email":"goham41933@lance7.com",
"token":"goham41933",
"code":"RZ05U6",
"active":true,
"subscribed":true}
```

response body:

```
{"success":"put call succeed!",
"url":"/token",
"data":{}}
```

jwt:
Invoke URL: https://9p2jaausn8.execute-api.eu-west-3.amazonaws.com/sampledev

- **/pubkey**: manages the public key to verify the signed jwt

  - **GET**: /pubkey
    description: obtain the public key to verify the signed jwt
    response body:

    ```
    {"key": "[public key]"}
    ```

emailSenderAPI:
Invoke URL: https://822240w7r0.execute-api.eu-west-3.amazonaws.com/sampledev

- **/sendEmail**: manages the sending of an email message. References the manually created function "sendEmail"

  - **POST**: /sendEmail
    description: sends an email with the provided details
    usage: Thanks.js, CreateAccount.js
    request body:

    ```
    {"email":"[email address]",
    "object":"[email object]",
    "message":"[message]"}
    ```

API Gateway allows for several parallel requests to be managed in a completely transparent way. This service, too, can resize the allocated resources based on their actual usage. The cost of this service, in fact, is calculated per-usage, based on how many requests are forwarded and on the amount of inward and outward traffic.
Other than RESTful APIs, API Gateway also allows setting up WebSockets. This service grants a combination of customization and ease of use, which allows to manage requests in detail, while resources get managed automatically.

3.1.3 Lambda

AWS Lambda allows users to execute code in a completely serverless way. Functions can be written directly on Lambda's GUI, uploaded as a .zip file or inside a container and get invoked as an answer to events which trigger them, such as a request to AWS API Gateway.
These functions can also be automatically generated, for example if one desires to have a code to execute CRUD operations on a specific DynamoDB table. The automatically generated functions are:

- **surveyLambda**: manages the CRUD operations about Indoor Environmental Comfort survey's answers onto the SurveyResult table
- **personalLambda**: manages the CRUD operations about personal questions' survey answers onto the Personal table
- **userTokenLambda**: manages the CRUD operations about users and their data onto the userToken table

Other Lambda functions have instead been created manually:

- **jwt_verify**: verifies the signature on the provided jwt and checks if the issuer matches "prometeo.click". It has been created to be used as an Authorizer.
- **sendEmail**: sends an email to the email address specified in the body. The email will have as object and message body the ones specified in the body of the request.
- **getPubKey**: returns the public key necessary to verify the generated jwt. This will be used by Grafana to perform checks on the jwt stored in the cookies.

Lambda can be applied on large amounts of data and fits several frontend contexts, from web to mobile, granting good performance also for machine learning.
Just like all other services, Lambda also scales its resources automatically based on actual usage, making the user pay only for what he uses.
Lambda's serverless nature absolves the user from the responsibility of infrastructure management, granting reliability and high availability promised by Amazon in a service which handles every aspect of the code, from deploy to updates, covering logs and monitoring, leaving to developers only the logic management aspect.

3.1.4 DynamoDB

DynamoDB is a NoSQL database which makes of scalability its spearhead. It's based on key-value pairs, grouped in records furthermore grouped into tables.

Data inside of it is organized based on partition keys, which have about the same role as primary keys inside relational databases.
Partition keys can also be paired with sorting keys, used to optimize order-based queries (such as begins with, intervals, greater than, lesser than...)
The tables used in this project include:

- **surveyResult**: stores data about the answers provided to the Indoor Environmental Comfort survey
  - **resultID**: string, partition key, represents a unique identificator for a set of answers
  - **multisensor**: string, the multisensor identificator corresponding to the closest measurement station
  - **Q1**: string, answer to the question "Are you satisfied with the thermal, acoustic, visual, and air quality conditions in your environment?". Possible values are:
    - **1**: very satisfied
    - **2**: slightly satisfied
    - **3**: slightly unsatisfied
    - **4**: very unsatisfied
  - **Q2**: array of strings, answer to the question "Your evaluation is negative, can you tell us which environmental quality aspects are you dissatisfied with?". Possible values are:
    - **THERMAL COMFORT**
    - **ACOUSTIC COMFORT**
    - **VISUAL COMFORT**
    - **INDOOR AIR QUALITY**
  - **Q2.5**: array of strings, answer to the question "Your evaluation is positive, can you tell us which environmental quality aspects you consider particularly satisfying?". Possible values are:
    - **THERMAL COMFORT**
    - **ACOUSTIC COMFORT**
    - **VISUAL COMFORT**
    - **INDOOR AIR QUALITY**
  - **Q3**: string, answer to the question "Please indicate on the following scale how YOU feel NOW.". Possible values are:
    - **3**:Hot
    - **2**:Warm

    * **1**:Slightly warm
    * **0**:Neutral
    * **-1**:Slightly cool
    * **-2**:Cool
    * **-3**:Cold
  - **Q4**: string, answer to the question "Please indicate on the following scale how YOU find the AIR VELOCITY in your environment NOW.". Possible values are:
    * **4**:Very draughty
    * **3**:Draughty
    * **2**:Slightly draughty
    * **1**:Not draughty
  - **Q5**: string, answer to the question "Please indicate on the following scale how YOU find the NOISE in your environment NOW.". Possible values are:
    * **4**:Very annoying
    * **3**:Annoying
    * **2**:Slightly annoying
    * **1**:Not annoying
  - **Q6**: array of strings, answer to the question "Please indicate any sources of noise YOU can hear in your environment NOW.". Possible values are:
    * **Building systems**
    * **Computer, printer, other office equipments**
    * **People chatting**
    * **Road traffic**
    * **Other noises from the outside**
    * **Other**
    * **None**
  - **Q7**: string, answer to the question "Please indicate on the following scale how YOU find your VISUAL environment NOW.". Possible values are:
    * **4**:Very uncomfortable
    * **3**:Uncomfortable
    * **2**:Slightly uncomfortable
    * **1**:Not uncomfortable

- **Q8**: array of strings, answer to the question "Please indicate any sources of glare YOU can see in your VISUAL environment NOW.". Possible values are:
  - **Windows**
  - **Lamps**
  - **Glass surfaces**
  - **Computer screens**
  - **Reflective surfaces**
  - **Other**
  - **None**
- **Q9**: string, answer to the question "Please rate on the following scale how YOU would like your visual environment to be NOW.". Possible values are:
  - **3**:Much lighter
  - **2**:Lighter
  - **1**:Slightly lighter
  - **0**:No change
  - **-1**:Slightly darker
  - **-2**:Darker
  - **-3**:Much darker
- **Q10**: string, answer to the question "Please indicate on the following scale how YOU find the AIR QUALITY in your environment NOW.". Possible values are:
  - **4**:Very smelly
  - **3**:Smelly
  - **2**:Slightly smelly
  - **1**:Not smelly
- **Q11**: array of strings, answer to the question "Please indicate any sources of pollution that contribute to the AIR QUALITY in your environment NOW.". Possible values are:
  - **Tobacco smoke**
  - **Human odours**
  - **Chemical odours**
  - **Other**
  - **None**

  - **Q12**: string, answer to the question "If you want, you can leave other comments".
  - **timestamp**: number, milliseconds since the Unix Epoch when the survey was completed
  - **user**: string, email of the logged user who answered the survey or string in the format of "anon[number]" for anonymous users

- **Personal**
  - **personalID**: string, partition key, represents a unique identificator for a set of personal answers
  - **Q1**: string, answer to the question "Gender". Possible answers:
    * **Male**
    * **Female**
  - **Q2**: string, answer to the question "Age". Possible answers:
    * **18-25**
    * **26-35**
    * **36-50**
    * **51-65**
    * **65+**
  - **Q3**: string, answer to the question "Country of Birth".
  - **Q4**: string, answer to the question "Educational qualification". Possible answers:
    * **Ph.D**
    * **Master's Degree**
    * **Bachelor's degree**
    * **High School**
    * **None**
  - **Q5**: string, answer to the question "Intended use of the building". Possible answers:
    * **Office**
    * **School**
    * **Museum**
    * **Hotel**
    * **Hospital**
    * **House**

- **Q6**: string, answer to the question "Ambit/role". Possible answers:
  * **Engineering**
  * **Management**
  * **Administration**
  * **Creative, design and architecture**
  * **Sales and public affairs**
  * **Teaching and research**
  * **Services**
  * **Head Teacher**
  * **Teacher**
  * **Administrative staff**
  * **Technical staff**
  * **Auxiliary staff**
  * **Student**
  * **Manager**
  * **Research, care and management of collections staff**
  * **Services and relations with public staff**
  * **Administrative, financial, management and public relations staff**
  * **Facilities and safety staff**
  * **Tourist Guide**
  * **Tourist**
  * **Receptionist**
  * **Chambermaid**
  * **Waiter**
  * **Chef**
  * **Barman**
  * **Customer**
  * **Medical director**
  * **Hospital secretary**
  * **Doctor**
  * **Nurse**
  * **Social health operator**
  * **Patient**
  * **Inhabitant**
  * **Guest**

    - **Other**
- **Q7**: string, answer to the question "Number of people in the environment". Possible answers:
    - **1**
    - **2 to 5**
    - **6 to 10**
    - **10+**
- **Q8**: string, answer to the question "Visual impariments". Possible answers:
    - **Yes**
    - **No**
- **Q9**: string, answer to the question "Hearing impariments". Possible answers:
    - **Yes**
    - **No**
- **Q10**: string, answer to the question "Do you smoke?". Possible answers:
    - **Yes**
    - **No**
- **Q11**: string, answer to the question "Do you conduct a healthy lifestyle?". Possible answers:
    - **Yes**
    - **No**
- **Q12**: string, answer to the question "Does an unsatisfactory Indoor Environmental Quality significantly reduce your work productivity?". Possible answers:
    - **Yes**
    - **No**
- **question2**: string, answer to the question "Does an unsatisfactory Indoor Environmental Quality significantly reduce your well-being?". Possible answers:
    - **Yes**
    - **No**
- **Q13**: string, answer to the question "Do you have control on widows opening and closing?". Possible answers:
    - **Yes**

  - **No**
- **Q14**: string, answer to the question "Do you have control on solar shading?". Possible answers:
  - **Yes**
  - **No**
- **Q15**: string, answer to the question "Do you have control on electric lightings?". Possible answers:
  - **Yes**
  - **No**
- **Q16**: string, answer to the question "Do you have control on heating system?". Possible answers:
  - **Yes**
  - **No**
- **Q17**: string, answer to the question "Do you have control on cooling system?". Possible answers:
  - **Yes**
  - **No**
- **Q18**: string, answer to the question "Do you have control on reducing annoyance from noise?". Possible answers:
  - **Yes**
  - **No**
- **Q19**: string, answer to the question "Do you think it's important to have control on widows opening and closing?". Possible answers:
  - **Yes**
  - **No**
- **Q20**: string, answer to the question "Do you think it's important to have control on solar shading?". Possible answers:
  - **Yes**
  - **No**
- **Q21**: string, answer to the question "Do you think it's important to have control on electric lightings?". Possible answers:
  - **Yes**
  - **No**
- **Q22**: string, answer to the question "Do you think it's important to have control on heating system?". Possible answers:

    * **Yes**
    * **No**
  - **Q23**: string, answer to the question "Do you think it's important to have control on cooling system?". Possible answers:
    * **Yes**
    * **No**
  - **Q24**: string, answer to the question "Do you think it's important to have control on reducing annoyance from noise?". Possible answers:
    * **Yes**
    * **No**
  - **timestamp**: number, milliseconds since the Unix Epoch when the survey was completed
  - **user**: string, email of the logged user who answered the survey or string in the format of "anon[number]" for anonymous users

- **userToken**: stores data about the users' accounts
  - **email**: string, partition key, unique email of the user
  - **active**: boolean, whether the account has been verified (and is therefore active) or not
  - **code**: string, 6 letters code used to activate the account
  - **subscribed**: boolean, not in use yet, whether the user wants to receive emails or not
  - **token**: string, token the user uses to log in

DynamoDB grants encryption for data at rest and its scalability is based on usage, so that a user doesn't have to intervene in order to change database settings to use all and only the required resources. The great automation in database management operation makes it a fitting solution even for beginner deployers. It is, moreover, perfectly integrated with other AWS services, not only for data manipulation, but also for metadata. Through the monitoring tools provided by AWS it's possible to analyze the current functioning status and make forecasts about future developments.
The first 25GB are stored in the database for free and 200 million reads and writes are granted for free each month.
Writing data into DynamoDB is managed through AWS APIs, which allow to access AWS Lambda functions through AWS API Gateway.
At table creation, an option is provided to generate code able to perform CRUD operations. Data is provided and retrieved in JSON format, which makes them

easy to write, interpret and manipulate.
One of the limits imposed by this solution is that performance is not as favorable in case of time series. AWS documentation suggests taking special care in settings if one desires to work with time series, in order not to affect the system's efficiency.

3.1.5 Amazon Cognito

Amazon Cognito is an essential tool to manage in an almost automatic way the authentication and authorization part, together with sessions. It grants data encryption both in transit and at rest and allows authentication to access Amazon resources through Authorizers, which can look directly into their reference User Pool or can be based on Lambda Functions for users' authentication.
Through the creation of a User Pool, one can keep track of the users registered in an application.

To create a User Pool, a user could either rely on Amplify CLI, through the command "amplify add auth" or through the use of Cognito's web interface.
To create the SurveyAdmin User Pool we chose the second option. First I opened Cognito's User Pool list and I selected "Create user pool".
Between the Authentication Providers, I chose "Cognito user pool", as the context didn't call for Federated Identities providers.
AWS APIs allow to memorize the registration of a user based on customizable authentication criteria, which can include email, username, address or phone number. These info can then be used to verify users and, once the account has been activated, for login. As these accounts had the only goal to authenticate our devices, I simply selected "User name" as the only sign-in option and I ticked the option to make the username not case sensitive.
In the password policy settings, I set that our passwords had to be at least 8 characters long and with the only requirement of having 1 lowercase letter.
I didn't enable MFA nor password recovery, as it's thought to be managed directly by the administration side.
I disabled Cognito's self registration and automatic delivery of messages for verification, as they are not required in this context. No attribute is required for creation, as well as no custom attributes.
Emails, although it's not expected to send them, would be managed by Amazon's SES service. All other default settings for the Email have been accepted.
We're then asked to choose a name for our user pool, which in this case would be "SurveyAdmin" and we're asked to define our app clients. We need one for each version of our application we have deployed. In this case, we have two, with the same settings: surveyWebApp and NS01_webapp.
Ours, in particular, are public clients. We're asked to choose a name for the

app client of reference, we choose not to generate a client secret in order to directly access the server-side of our app. Our authentication flow includes ALLOW_CUSTOM_AUTH, ALLOW_REFRESH_TOKEN_AUTH and ALLOW_USER_SRP_AUTH. The authentication flow duration is kept to the default value of 3 minutes, the refresh token expiration is set to 3650 days while Access token and ID token expirations are set to 1 day. Advanced authentication settings are kept to their default values. Once the review of these settings is complete, our Client Web App and our User Pool has been configured.

Cognito provides prebuilt, customizable templates for registration and login, based on the previous settings. These forms already include references to the previously cited APIs.
In our case, we chose to manually embed these APIs in our code.
Application integration happens through the creation of a configuration file containing the User Pool's details. This configuration file is used for the so-called Account Context, which contains not only info about active users, but also functions required for their management, such as authentication or getting data about the current session.
We created a file called UserPool.js where the user pool identificator and web app client identificator have been stored and became parameters for a CognitoUserPool object. This file has then been imported in another file called Account.js. In that file, through the use of the amazon-cognito-identity-js library and CreateContext by React we can create a context and functions to get the current session and authenticate the user.
In particular, the session is obtained through user.getSession, where user is the current active user according to the User Pool, and authentication can be achieved through user.authenticateUser, where two callback functions are taken as parameters, one with the actions in case of success and one in case of failure.
These functions are then used in App.js in order to keep track of the AccountContext, through the use of react's useContext, which takes Cognito's AccountContext as its parameter. Then authentication is performed taking username and password from the website's query string and inserting them into the "authenticate" function provided by Account.js.
A check on the current active user through useContext and getSession is performed every time the URL changes.

As aforementioned, in this User pool accounts are created by the administration through Cognito's user interface.
To do so, it's also required a terminal where AWS is installed. From Cognito's dashboard, select the desired User Pool, in our case SurveyAdmin. Under the Users tab, click on "Create user". Tick "Don't send an invitation", select a username and

set a password, then click on "Create user".
This will create an account whose state is "Force change password". In order to exit this state and obtain a confirmed account, access the terminal and install aws cli. Once the aws cli is installed, configure it through the command:

```
aws configure
```

This will ask the user to insert his or her credentials and information about his region and default output format. In our case:

- **AWS Access Key ID**: the previously created AWS Access Key ID
- **AWS Secret Access Key**: the previously creates AWS Secret Access Key
- **Default region name**: eu-west-3, as we created our Cognito User Pool in this region
- **Default output format**: json

Then, from here we can insert the following command, which will prompt a change of the password and, consequently, turn the newly created user into a Confirmed state.

```
aws cognito-idp admin-set-user-password
--user-pool-id [User Pool ID] --username [username]
--password [password] --permanent
```

Where [User Pool ID] is the Identificator of our User Pool, [username] is the username of the account we created previously and [password] is the password we want to have for that user.
Now the account will be in a Confirmed state and, therefore, it will be possible to use said credentials to log in.

Together with credentials, Cognito also provides a Federated Authentication mode, which allows access through other public user pools such as google, facebook, OpenID Connect and SAML.
Cognito applies the authentication standard OAuth2.0, granting the users' identities through the use of an access token in the form of a JWT, which gets periodically renewed through a Refresh Token, as long as the session is active. These tokens are generated by Cognito based on an asymmetric key.
Cognito also provides a free usage tier which allows up to 50.000 active users simultaneously, largely satisfying the expected requirements.

3.1.6 AWS Amplify

After setting up out backend, it's possible to upload our frontend to make it publicly accessible. This result can be obtained through the use of Amplify, which promotes its ease of use, granting the possibility of deploying a web or mobile application in minutes, with all its Amazon services perfectly integrated.
Managing both backend and frontend publication, it generates and manages a CI/CD pipeline which allows to release updated code frequently and easily.

Our deployment, particularly, has already been done through the use of Amplify CLI, which deploys our frontend and the backend we created through the use of "amplify add api".
One missing step is to save Cognito's references into Amplify's Environment Variables, in order to grant a seamless deployment.
To do so, select the created app from the Amplify's console, in our case it would be ECS_v0. Under "App Settngs" choose the tab "Environment Variables" and the following variables should be added to the list:

- AMPLIFY_WEBCLIENT_ID: the ID of the App Client to be used by web application, available from Cognito's console, inside the User Pool's settings, under the "App integration" tab
- AMPLIFY_NATIVECLIENT_ID: the ID of the App client to be used by native applications, which in our case matches the AMPLIFY_WEBCLIENT_ID
- AMPLIFY_USERPOOL_ID: the ID of our User Pool, available from Cognito's console, next to the reference User Pool

Amplify accepts code from different sources, which can be Github, Gitlab, AWS CodeCommit (Amazon's Version Control System) or without using a Git provider.
In our case, we chose to use GitHub.
As previously stated, our first version of the application was deployed in us-east-1.
The local files include both the frontend and information able to reconstruct the backend inside a folder named "amplify".
We split our git repository into another branch called "Paris" and cloned it.
Inside the newly cloned folder, we opened a terminal and executed the commands previously described in Amplify CLI, except for "amplify add api", as information about our API was already available.
This created a new environment for our application, meaning that although it worked exactly like the American version, their data was completely separated and the two could evolve independently.
Some functions, such as getPubKey and sendEmail, had not been saved in the application's local files, therefore had to be manually copied from the N.Virginia

environment to the Paris one, by creating the relative Lambda functions and API gateway resources through the use of the web interface.
For example, to do so, I went into the N.Virginia page of the Lambda console and selected "getPubKey". In its page, under the "Actions" menu, I selected "Export function" and later "Download deployment package". This resulted in the download of a zip file.
Moving to the Paris version of the Lambda console, I selected "Create function", chose "getPubKey" as its name and chose "Create function", leaving the other settings at their default value. From the new function page, under Code source, I selected "Upload from" and chose ".zip file". I then uploaded the code from the zip file I had previously downloaded.
To make this function accessible, I also had to create a API resource, therefore I opened the API Gateway console, still in Paris region.
I chose "Create API" and clicked "Build" on "REST API". In its settings, I chose the REST protocol, to create a "New API", I named it "jwt" and chose a Regional Endpoint type.
Being brought to the new API page, I selected "Create Resource" under the actions menu. I named the path and the resource as "pubkey" and I ticked "Enable API Gateway CORS". I then selected the newly created resource and, still under the "Action" menu I chose "Create Method" and chose the "ANY" method. I chose "Lambda function" as integration type, I ticked "Use Lambda Proxy integration" and entered the name of the previously created Lambda function as "getPubKey" and saved.
Afterwards, under the "Actions" menu I selected "Deploy API". I was asked to choose a deployment stage name and I chose "sampledev". Then I clicked on "deploy" to make the API available.

Amplify is connected to the branch and it automatically deploys our app upon every new commit on its reference branch, always ensuring the most recent version without interruptions during updates.
Together with the frontend, in this phase we also deploy the backend previously developed, making our website usable in minutes.
On Amplify there's also the chance to define one or more URLs where to make the application reachable, based upon a Route53 domain or a third party one. In our case, we chose Route53 for the perfect integration with the environment and for its ease of configuration.

After creating our Route53 domain, in our application's page of Amplify console, under App Settings, I choose "Domain management". From here, I selected "Add domain" and entered the domain we had just bought, "prometeo.click", then I clicked "save". From the newly created domain, I clicked "manage subdomains"

and clicked on "add", inserting a new subdomain called "paris", meaning that our website will be reachable from "paris.prometeo.click".
I also disabled the master "prometeo.click" domain, so that the application is only reachable from "paris.prometeo.click". After clicking update, the website will be available in a matter of minutes.
This process has also been repeated for the second version of our application, ECS_NS01, in order to make it available on "ns01.prometeo.click". This makes it so that different versions of our application are available on different addresses, still under the same domain.

3.1.7 Route53

Route53 is a DNS service which allows for the creation of domains and rules to determine the routing towards the user's own resources, both inside and outside AWS' ecosystem.
The system is moreover integratable with a variety of services such as Load Balancer and Traffic Flow's routing rules, which makes it so that traffic is automatically redirected towards the correct resources to grant efficient operation for the network.
In our case, the service has been used for the creation of a domain to deploy the application for collection of subjective data about environmental comfort and to assign an alias for the application about objective data visualization, placing them under the same domain.

To register a new domain, we accessed Route53's dashboard, selected "Register domains" and typed the domain to register, checking for its availability. In our case, it was chosen "prometeo.click". At this point, the domain was added to the cart and purchased, making it available for the customer for the amount of time defined when buying it. It's also possible to activate the automatic domain renewal at expiration time. After the purchase phase, an email is sent to the buyer to certify the correct domain acquisition.
Other than for domain acquisition it's also possible to use Route53 to create routing specifications. This was also required in order to prove we were in control of the domain we proposed when registering our Amazon SES mail domain.
In particular, some entries have been automatically added:

- CNAME records are automatically inserted as we register a new subdomain in Amplify, as previously described
- a NS record to correlate the newly created domain, in our case "prometeo.click", to their corresponding nameservers. This has also been done for a previously created subdomain, "dev.prometeo.click"
- a SOA entry to return authoritative information about the DNS zone

Other entries, which have been manually added, are:

- a CNAME record, "caimano.ns1.prometeo.click", has been introduced in order to route the requests to that address to "caimano.polito.it", where our Grafana dashboard was hosted. This has been done in order to put Grafana and our dashboard under the same domain
- more CNAME entries whose characteristics have been described by Amazon SES in order to prove we were in control of the domain we were trying to verify for our mail server
- an MX record to redirect the incoming email traffic to a secondary address in order to be able to read and receive email

To insert a new record, we opened Route53's dashboard, visualized the Hosted Zones and selected the desired Hosted Zone, in our case "prometeo.click".
In the main page, a table displays all current records. In that page select "Create Record". This opens a form which allows to insert the new record's characteristics, which vary from type to type
For example, in order to add a CNAME record for AmazonSES we select "Create record". In "record type" we choose "CNAME". We insert the record name and value as defined by Amazon SES and keep the other settings at their default value. Then, we click "create records" and the new entry will be added to the table.

As previously mentioned, the domain defined in Route53 has also been used to define an email address with which we can contact the app's users to send them messages in case of sign up or survey's completion.

3.1.8 Amazon SES

Amazon Simple Email Service is an Amazon service which allows users to send automatic email in a fast and simple way. Through the use of its APIs it makes it possible to send and receive emails. A particular aspect of this service is its use of metadata, which creates reports and statistics about the communication's efficiency in order to keep track of the user's own reputation to make sure that every message is received efficiently.
Amazon SES is recommended for both transactional and commercial mail.

To start sending emails, we created an identity from Amazon SES' console by clicking on the "Create identity" button.
We chose to create an identity based on the domain we previously created on Route53, "prometeo.click", so we chose Domain as our Identity type and we inserted our domain under the "domain" field. To verify our domain, Amazon SES advices

us that the verification will happen as a DKIM-based domain verification. This means that the DKIM values will have to be inserted in the DNS records table to demonstrate its ownership.
Once completed the domain registration a verification procedure will start and afterwards our address will be inserted in a sandbox. To exit the sandbox and make the email domain completely operative, we need to advance a request to Amazon. This can be done through the Amazon SES dashboard. By selecting the identity we want to take out of the sandbox (in our case, prometeo.click), we click "Account dashboard". On the top part of the page, a message will appear if the identity is still in the sandbox. Together with it, a button to require the exit will be displayed. By clicking on it, a set of information about the intended use of the email will be asked. This form will then be analyzed by Amazon which will then decide whether to actually enable email transmission outside the sandbox or not. The decisional process can take an amount of time that can range from some hours to a few days. In the meantime, we created a Lambda function to send emails.
In order to do so, we created a Lamdbda function called "sendEmail", which requires "aws-sdk" library in order to interface with Amazon SES. We also setup our SES reference by setting its region to eu-west-3.
After taking the email parameters from the request body, we use ses.sendEmail(params) function to send the email according to the parameters defined in the "params" object. An example of params is:

```
{Destination: {ToAddresses: [/*destination email*/]},
Message: {
    Body: {Text: { Data: /*message content*/ },},
    Subject: { Data: /*message subject*/ },
},
Source: "no-reply@prometeo.click"}
```

This function must then be deployed through the use of the "Deploy" button on top of the text editor.
An API Gateway resource called "EmailSenderAPI" is also created in order be able to invoke the previously created function. The creation process works as earlier described.
Moreover, in order to allow the function to send email, a special policy has to be created. To create a policy we access Amazon IAM web interface and, under the "policy" tab, we select "Create Policy". In order to do so, we rely on the JSON definition, clicking on the "JSON" tab of the creator and pasting the following piece of code:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "VisualEditor0",
            "Effect": "Allow",
            "Action": [
                "ses:SendEmail",
                "ses:SendRawEmail"
            ],
            "Resource": "*"
        }
    ]
}
```

We choose to add no tags and, after selecting an evocative name for this policy, in our case "SendEmailPolicy", we can finalize its creation. At creation of the lambda, an execution role is automatically generated. In our case, it was named "sendEmail-role-4rzc240v" and we could find it in the Roles tab of the IAM Console. By opening its details, we choose "Add permissions", "Attach policies" and select the SendEmailPolicy we created earlier. Finally, we click "Attach policies"
Since this function has no particular correlation to DynamoDB tables or different versions of data it references, we chose to use the very same resource for both our environments, sampledev (which is Paris' backend environment) and glori (which is ns01's environment).

3.1.9 Considerations about AWS Services

The richness of AWS' environment makes it particularly interesting on the development side. Despite its costs being higher than some of its main competitors, there is a payback in terms of a rich and exhaustive documentation, an active community of users, a continuous support by its developers and a wide offer of services.
The perfect integration among resources that represent virtually any need a user might have is undoubtedly a strong point of this solution.
Albeit, high automation of some of its aspects might make it difficult to introduce a more precise customization.
An example of said limitation has been noted in the attempt of setting up the communication between the app for the collection of subjective comfort data (on Amplify) with its objective counterpart (on Grafana). In order to achieve a higher level of optimization, it would have been fit to be allowed to set up ad hoc caching policies, but these were systematically overwritten by a service called CloudFront. CloudFront is a CDN Service (Content Delivery Network) which mostly operates at the network edge to make it so that our application is distributed, protected and available through systems that span from load balancing and routing, encryption

and access control. Just like many other previously analyzed aspects, CloudFront is also automatically provided and managed autonomously by Amplify in a standard way.
Together with the previously mentioned services, CloudFront also manages the caching policies we would have liked to customize, however, being CloudFront integrated, it's not possible to access its dashboard, with the result that there is no way of personalizing those policies.
Afterwards, optimization has been achieved by modifying the rendering order and groups of some React components, other than substituting some iframe elements with a textual field updated through a single query repeated every 5 seconds to update the values.

3.2 Reusability of the solution

The extreme modularity of AWS' architecture makes the proposed solution particularly fit also for a context where modifications are expected. Adding new services does not imply substantial modifications to what is already present.
Being AWS extremely business-oriented, Amazon developed its infrastructure in a way that it can adapt to the necessity of having distinguished environments. This can be used both to segregate development and production environments, but also for different releases, just like we did for our two versions, "paris" and "ns01".
The management of the different environments works in a way much similar to git's branches, allowing for the creation of new environments, pushing to save local modifications to the cloud and swap from one area to another through checkouts.
This versatility also applies to the frontend. If one desired, for example, to create a new application specifically developed for mobile, the backend would still be entirely usable as it's independent from the frontend's paradigms and technologies.

3.3 Cost analysis

Our system is expected to be deployed in Europe, more specifically in the region eu-west-3 (Paris) as it provides all of the services we are to use.
The solutions to be compared are:

- A machine-based solution, where a server is rented to host our services
- A serverless solution, based on the cumulative costs of services per usage

In this analysis we can make the following assumptions on costs:

- One month (30 days) of operativity without considering Saturdays and Sundays, which are around 8 per month

- Around 10 logged users and 10 anonymous users
- An average of one answer per day
- All logged users have answered once their personal questions
- All anonymous users reply to the personal questions
- One visit per day per user on the profile page
- One deploy per week, each of about 5 minutes of deploy time
- Average answer size of surveys: 250 bytes
- Average answer size of personal questions: 125 bytes
- Average account size: 100 bytes

The average answers' sizes are calculated as the average between the size of an answer where all fields have been filled and the size of an answer where the minimum number of fields have been answered, both rounded up to the higher 50 multiple in order to get a conservative estimate.
In detail the largest survey answer is around 350 bytes and the smallest is around 100 bytes, while the largest personal answer is around 250 bytes, while the smallest is 0 bytes, as all the questions are optional.

3.3.1 Machine-based solution

The machine-based solution can be implemented by the use of Amazon EC2.
This service has an hour-based pricing and an estimate can be performed through the use of AWS Pricing calculator[28].
A quick estimate was performed, inserting the following assumptions:

- AWS[29] suggests using 2 t2.micro EC2 instances.
- As our project will be used for 8 hours a day, the utilization percentage will be around 33
- The operating system taken into consideration was Linux
- The pricing strategy chosen will be On-Demand Instances
- The expected storage during the first month of operativity will include:
  - 10 users for a space of 1000 bytes

- 230 personal questionnaire's answers for a space of 28750 bytes
- 440 indoor environmental survey's answers for a space of 110000 bytes
- Thus, the total expected space for the first month of operativity will be around: 139750 bytes or 0,13975 MB. Every following month under the same conditions will bring an increase of 0.13875 MB.

To keep a conservative approach and due to the fact that there would be no price overhead, we can set the amount of storage to 22MB, which is expected to largely satisfy our requirements.

The total cost of this solution would be 13.43$ per month, assuming to find free versions for all the needed services.
Alternative solutions have also been considered:

Model	vCPU	Mem [GiB]	Storage [GB]	Cost taxFree [$/month]
AWS t2.micro	1	1	variable	10.20 + 0.12 per GB/stor
AWS t2.small	1	2	variable	13.43 + 0.12 per GB/stor
AWS t2.medium	2	4	variable	26.94 + 0.12 per GB/stor
Aruba VPS V2I4	2	4	80	13.25

Table 3.1: Machine-based cost comparison

3.3.2 Service-based solution

The service-based solution allows us to rely on a free tier for a few of the required services.
The expected monthly costs, evaluated according to AWS Pricing Calculator, are the following:
In more detail:

- **Amplify**:Amplify costs around 0,01$ per month for each minute of build. Moreover, additional costs should be registered for the storage of the different versions of the application. Since our website is about 1.5GB and four builds of it are expected to be delivered every month, we must accommodate 6GB of storage. Finally, also the data exchanged monthly has an influence on these costs, and it is estimated to be less than 5MB of data.
- **Route53**: Route53's costs stem from the fact that, in the eu-west-3 region, each hosted zone costs 0,50$ every month.

Amplify	0,55$
Route53	0,50$
DynamoDB	0,00$
Cognito	0,00$
API Gateway	0,00$
SES	0.00$
Lambda	0.00$
TOTAL	1,05$

Table 3.2: Expected AWS services costs in an office context

- **DynamoDB**: DynamoDB costs depend on storage and on the amount of read and write operations performed. In our system the expected storage required is a few MB and we expect to perform an amount of read and write operations in the order of magnitude of thousands. Since the size of our data is in the order of Bytes, DynamoDB does not represent a significant cost.
- **Cognito**: Cognito can still be configured within the free tier, as we expect to have an amount of monthly active users much lower than the 50.000 provided and we did not set up any advanced security features.
- **API Gateway**: API gateway's cost is almost null due to the fact that 0,01$ would allow us to perform up to 5000 requests, which is much more than expected
- **Amazon SES**: Amazon SES is also free, as we expect to send a few dozens email messages and up to 100 emails can be sent for less than 0,01$
- **Lambda**: Lambda allows us to answer up to one million requests for free each month, satisfying our requirements with a large margin.

To these monthly costs should also be added the expected development costs.
The need to deploy the changes often, the amount of attempts required in order to write and test features, the purchase of the domain and the space required to store the data can make the costs rise, but operatively the monthly expenses are limited.

The serverless solution results particularly advantageous in our specific use case, due to the fact that the current context of application is of a reduced size. This is because AWS services scale based exactly on the actual usage and the billing works in exactly the same way, resulting in a significant saving compared to the machine-based solution.
The EC2 solution, in fact, is based on the operativity time and, even if the machine we need has small requirements, we can still assume that a great part of the

resources we have allocated and we paid for will not be fully exploited.

A different speculation can be performed based on another potential use case, which would be represented by deploying our tablets inside one of the largest classrooms of the Polytechnic University of Turin.
The assumption, in this case, would be to have around 5000 students every day filling the survey twice.
In this situation, different assumptions should be made on the different services' usage.

- **Amplify**: The expected amount of build time would not change, just like there would be no change in the amount of stored space to host the application. The amount of data served, albeit rising, would determine a minimal rise in costs.
- **Route53**: Route53's costs should not change, as we would still require just one hosted zone.
- **DynamoDB**: The increased requirements in storage, together with the much higher amount of read and write operations, determined a significant rise in costs.
- **Cognito**: Cognito would still fall under the free tier requirements even if it were to be expanded to the management of users, other than devices authentication, therefore there would be no increase in costs.
- **API Gateway**: The increase in the expected requests made us go beyond the free tier's consumption, resulting in an increase in costs. In particular, we expect to answer around 5000 http calls, represented by the email sending and slightly more than a million REST API calls.
- **Amazon SES**: We expect to send about 5000 emails, which would result in an increase in costs of about 0,50$ per month
- **Lambda**: The amount of expected Lambda requests would exceed the million of requests per month represented by the free tier. An amount of 1.1 million requests per month should be enough to satisfy our needs, resulting in minimal costs.

This solution is therefore to be considered interesting even in different contexts of application, with a larger pool of active users, thanks to the automatic scaling system and the pay-per-use pricing system enforced by Amazon.

API Gateway	3,50$
Amplify	0,60$
Route53	0,50$
SES	0,50$
DynamoDB	0,20$
Lambda	0,02$
Cognito	0,00$
TOTAL	5,32$

Table 3.3: Expected AWS services costs in a university context

Chapter 4

Frontend

Our application's frontend is developed in Javascript React, which has been chosen for its familiarity, diffusion and expressive power.
To quicken some implementation aspects, we looked for a solution which could also include customizable, pre-built components, especially for the building of surveys and forms.
We chose to use SurveyJS and Formik, respectively.
Because of the huge importance given to the aesthetic aspect of the application, the use of pre-built components was not sufficient: a large amount of CSS allowed us to create the pages in order to be exactly matching with the descriptions proposed by the Architecture department.
On the other hand, the punctual personalization and the mostly horizontal layout given to the application made it very little responsive to vertical layouts.

4.1 Auxiliary Tools

4.1.1 SurveyJS

SurveyJS is a free, open source solution for the implementation of surveys. It provides a wide set of possible questions, a large chance of customization and useful functions which allow to collect data and partially intervene on the questions' characteristics.
Our SurveyJS journey begins in the SurveyJS Creator, a precious tool in the form of a graphic interface to allow us to build the structure of our code.
There, we are free to:

- add new questions
- group the questions in pages

- choose the visibility of said pages depending on a wide range of possible conditions
- selecting questionnaire-wide options like:
  - the survey's title and description and whether to display it
  - the default language
  - if the questionnaire is editable or read-only
  - if it's responsive or fixed-width or automatically sized
  - which logo to display and what size
  - showing or hiding navigation buttons
  - the text that we want the buttons to display
  - the possibility of moving automatically to the next question right after answering
  - how all questions are displayed (title and description position, indicators for mandatory questions, display errors)
  - the structures of pages
  - logic, data collection and validation
  - what to do on survey completion
- settings to make the survey a quiz or add a timer
- testing the questionnaire flow
- adding logic conditions on the questionnaire flow
- preview the generated JSON code
- translate the questionnaire in multiple languages for better localization

Moreover, for each question we can:

- change the kind of question we're working on
- choose the name and description
- choose whether it's visible, required or read-only
- choose the options, for closed-ended questions, by either adding them manually or getting them from an API
- introduce conditions on the question's visualization

- introduce conditions on the questions' options' visibility and on how many answers can be chosen
- define a secondary location where to store data
- choose a correct answer, in case of quizzes
- introduce validation

All these actions can easily be performed by the convenient visual interface and afterwards a JSON file will be generated containing all the information we defined earlier. That document holds by itself all of our survey's characteristics. For example, the files which hold our surveys' configurations are respectively "survey.json" for the Indoor Environmental Comfort survey and "personal.json" for the personal questions' survey.
In order to introduce it in our code, we must place the file in a convenient location in relation to our project's structure. From there, we can import the JSON file into the JavaScript file where the survey will be rendered. Together with it, we must also import the libraries needed to correctly display our survey and the relative React components.
Then we introduced some code to:

- Set the language
- Set some special CSS rules in correspondence to some questions where we wanted to set a particular style
- Display the survey
- Set the survey's behavior upon its end.

A bittersweet aspect of SurveyJS is the complete autonomy in the management of surveys.
Each survey can be represented with a finite state machine where every question is a new state and this feature is managed entirely by SurveyJS itself. This makes it impossible to intervene on the page using React states or effects, because every state change would cause the page to refresh and the survey's finite state machine to restart.
Although this can be limiting, the survey's management is still seamless and a huge load of work off the developers' shoulders.

4.1.2 Formik

Formik is a JavaScript library used to create forms in a declarative way.
Through the use of the appropriate tag and relative properties, it's possible to create

a form with all the required characteristics in terms of validation and behavior in a few lines of code.
All that is required is to define the initial values, the validation function, the callback function on submit and set up the form with regular html tags.
Formik will emit a group of elements to reference our variables' value, if there is any error associated with them, if it has been modified and of course it will provide handles to manage value changes and submit.
Inside the functions, we can reference those very same objects to act consequently, for example to display an error or to allow data submission.
Everything is therefore networked in an almost transparent way through the use of Formik's variables, which set the user free of the burden of managing several states and errors.
This makes the production of forms easier and more straightforward, granting also that no aspect has been left behind.
These characteristics, together with the fact that it's completely free and easy to introduce in our React code represented very strong points in favor of this solution.

4.2 React

React[30] is a JavaScript library used to realize Single-Page Applications, which are dynamic pages whose content depends on the interaction that the user has with them.

To do so, React relies on structures called Components.
Each application has a core which manages the rendering of all needed components, all the time.
A component is a JavaScript class which returns fragments of html code which will then be used inside the page. React's particular characteristic is that the code that precedes the return statement can be used to affect its content, granting therefore higher control on what will be visualized.
A peculiar feature of these elements is that, by including other Components in html, they can generate a hierarchical structure.
Just like a Chinese box, each component can be made of other components itself. This grants a higher and safer code reuse, because each duplicated code fragment can be substituted by a component invoked multiple times and maintainable from a single file. As previously mentioned, these components get selected and rendered each time individually. Choice can be based on multiple criteria: an element can be rendered always, only in specific paths or based on state variables' conditions.

SurveyJS has also been chosen for its compatibility with React.
By importing the libraries "survey-react-ui" and "survey-core" it's possible to have access to SurveyJS-developed components like Survey, Model and StylesManager. These components are fundamental for the management of the survey itself.
The following code will illustrate how SurveyJS' classes are used inside SurveyJS.js in order to implement the Indoor Environmental Comfort survey.

Survey is a React Object which takes its model and css as properties, together with the function to be called upon questionnaire completion:

```
<Survey id = 'surveyjs' css={css} model = {survey}
onComplete={sendDataToServer} />
```

"css" is a normal CSS file containing all the style directives to be applied onto the survey.
"survey" is the object generated by the Model constructor from the JSON file containing all information about the survey.

```
import * as surveyJSON from './resources/survey.json';
[...]
let survey = new Model(surveyJSON);
```

SendDataToServer is a special function which decides the behaviour of the survey upon completion. Since we can't tell at this stage if the user is done or still has to fill personal questions or create an account, at this stage we only save the answers in a state variable in order for them to be shipped only when the transaction is complete.
In this stage, user's data is also completed with additional information, such as timestamp, a unique id and multisensor reference.

```
function sendDataToServer(sur) {
let data = sur.data
data.resultID= uuid()
data.timestamp= new Date().getTime()
data.multisensor = localStorage.getItem("multi")
if (props.logged===""||props.logged===null) {
  data.user = generateAnonId()
  props.setAnon(data.user)
  props.setAnswers(data)
  navigate("/furtherQuestions")}
else {
  data.user = props.logged
  props.setAnswers(data)
  navigate("/thanks")}
}
```

The survey itself is represented by the Model object, which gets built, as previously described, by receiving the survey's JSON as its parameter.
This object is later used in order to define:

- the survey language:

  ```
  if(props.ita)
      survey.locale='it'
  ```

- ad-hoc css classes for specific questions:

  ```
  survey.onUpdateQuestionCssClasses.add((sur, options) => {
  let classes = options.cssClasses
  if(options.question.name==="Q4"||
  options.question.name==="Q3") {
    classes.title += " thermal noBorder"
    classes.titleOnAnswer = "";}
  else if(options.question.name==="Q5"||
  options.question.name==="Q6") {
    classes.title += " acoustic noBorder"
    classes.titleOnAnswer = "";}
  else if(options.question.name==="Q7"||
  options.question.name==="Q8"||
  options.question.name==="Q9") {
    classes.title += " visual noBorder"
    classes.titleOnAnswer = "";}
  else if(options.question.name==="Q10"||
  options.question.name==="Q11") {
    classes.title += " air noBorder"
    classes.titleOnAnswer = "";}
  })
  ```

StylesManager is used to apply the "modern" style onto the survey. These very same characteristics get supplemented by the previously described CSS classes.
We operate in the very same way to implement also the Personal Questions' survey in Personal.js.

As previously stated, these answers don't get immediately sent to the server. Depending on the path chosen by the user, different options are available:

- If the user is logged, the answers are sent up to the server in the Thanks.js page available right after completing the survey in the name of the logged user.
- If the user is unlogged and chooses to answer the personal questions, the answers are marked as answered by an "anon" user and they are sent up to the server in the Thanks.js page, right after sending the personal questionnaire's answers to server, still from the same anonymous user.
- If the user is unlogged and chooses to sign up, the answers are sent up to the server in CreateAccount.js in the name of the newly created user.

In order to store the answers into the DynamoDB database, we rely on the "aws-amplify" library. This library provides us with an API component which, through methods such as .get(), .post() and .put() allows us to send REST API requests to the resources we created on Amplify CLI.
Particularly, in order to save the survey's answers, this is the operation to perform:

```
let init = {
body: props.answers,
headers: {"Authorization": props.deviceJwt}
}
API.post("userTokenAPI", "/survey", init)
.then(/*action to perform in case of success*/)
.catch(/*action to perform in case of failure*/)
```

where "props.answers" is the JSON of answers generated by the survey upon completion complete with additional information, such as the user, timestamp and multisensor of reference as previously built in sendDataToServer function.
This command will trigger the userTokenAPI in its /survey path in order to store the received data into the surveyResult table.

To manage paths inside our Single-Page Application in React, we rely on a group of React-provided components, defined altogether as React-Router.
These components allow for the creation of Routes, each of which has its own path and element. This means that, when accessing that specific path, that very own element will be rendered.
Each route, itself, can contain others within, making it possible to create multilevel paths.
More in detail, we used the component ProtectedRoute for those pages that could only be accessed by authenticated devices. This special route works so that, when trying to connect to its path, the current logged device is checked. If a value is present, the Outlet component is returned, meaning that the page is displayed. Otherwise, the user is redirected to Page401, meaning "unauthorized"

```
//in ProtectedRoute.js
return props.logged ? <Outlet/> : <Page401/>;

//in App.js
//closed route
<Route exact path='/survey' element={<ProtectedRoute
logged={adminLogged}/>}>
    <Route path='/survey' element={<SurveyJS setAnon={setAnon}
    setAnswers={setAnswers} ita={ita} logged={logged}
    doLogout={doLogout}/>} />
</Route>
//open route
<Route path='/login' element={<Login deviceJwt={deviceJwt}
doLogin={doLogin} ita={ita}/>} />
```

The navigate construct can be used to switch paths and, therefore, visualize different components. It has been used in several pages. For example, from the home page in order to move to the Indoor Environmental Comfort survey page.

```
let navigate = props.useNavigate();
const routeSurvey = () => {navigate("/survey");}
[...]
<button  [...] onClick={routeSurvey}>{props.ita ? "Inizia il sondaggio"
"Start the questionnaire"}</button>
```

Within a path, components can be modified also based on states.
The useState construct allows creating a variable used as a state and a function to safely modify it. Updating a state variable makes it so that the component gets reloaded with the new state and consequently updated.
States can also be used inside the construct useEffect. An effect is a function which, according to its dependencies, can be executed only at a components' mounting, at the update of any of the elements in its dependency array or repeatedly as long as the component is mounted. This can be used to update states and the page's content.
For example, useEffect has been used in Personal.js in order to retrieve the user's previous answer to the personal questions' survey in order to allow the user to simply update them.

```
useEffect(()=>{
    if((user===null || user === undefined)&&props.deviceJwt===null)
        navigate("/login")
    if(user!==null) {
        let init={
            headers:{
            Authorization : localStorage.getItem("userJwt")
            }
        }
        API.get("userTokenAPI",
        "/personal/personalID?user=" + user, init)
        .then(resp => {
            resp.sort((a,b) => (a.timestamp < b.timestamp) ? 1 :
            ((b.timestamp < a.timestamp) ? -1 : 0))
            oldValues=resp[0]
            if(oldValues!==null)
                fillOldValues()
        }).catch([...])
    }
}, [])

function fillOldValues()
{
    for(let ov in oldValues)
    {
        if(ov!=="personalID" && ov!=="user" && ov!=="timestamp") {
            survey.setValue(ov, oldValues[ov]);
        }
    }
}
```

4.3 CSS

The precise aesthetic requirements of the application make it so that the customization level has been substantial. This has been achieved both through the use of classic CSS and through the use of Bootstrap.

CSS stands for Cascading Style Sheets and is made of a set of indications about how each element should be presented. Any aspect can be customized, starting from its margin or text color and size, moving on to its border and rounding.
CSS is made of classes, which group together rules related to an object, which can be identified based on class, kind of html component or identification. This makes it so that these rules are repeatable to keep consistent semantically related object's appearance.
SurveyJS uses consistent class names for every component of his surveys, making

it so that defining a set of css rules related to a specific class propagates the same modification to all equivalent objects. Moreover, additional class names can be added to some well-known elements, for example question titles, through the use of a function called "add" which can be used upon the property "onUpdateQuestionCssClasses" of the object Model.
This allows us to perform very punctual modifications to questions. For example, in that function we can notice the lines

```
let classes = options.cssClasses
if(options.question.name==="Q4"||
options.question.name==="Q3") {
  classes.title += " thermal noBorder"}
```

This means that, if the visualized question has name "Q3" or "Q4", the classes "thermal" and "noBorder" should be added to their list of classes. This makes it so that, together with the style properties they already have, some additional indications will be applied. Particularly:

```
.thermal {
background-color: rgba(167,101,102,0.6) !important;
border-top-left-radius: 50px 50px !important;
border-bottom-left-radius: 50px 50px !important;
border-top-right-radius: 50px 50px !important;
border-bottom-right-radius: 50px 50px !important;
}
.noBorder
{
    border-bottom: 0 !important;
}
```

This will result on questions Q3 and Q4 to have a reddish background color with slightly rounded corners and no orange border underneath them.
This very same principle has also been applied to a variety of other components by integrating their already existing css classes with additional rules to customize their appearance. For example, the navigation buttons generated by SurveyJS to move from one question to another, which are characterized by classes such as "sv-footer___prev-btn", "sv-footer___next-btn", "sv-footer___complete-btn" have been customized with the following indications:

```
#surveyjs .sv-footer__prev-btn,
#surveyjs .sv-footer__next-btn,
#surveyjs .sv-footer__complete-btn {
background-color: #FF9724!important;
font-size: 175%;}
```

Meaning that the buttons will have an orange color and a large text, matching the aesthetic of analogous orange-coloured buttons across the website.

Other than through the use of stylesheets, which group aesthetic indications in a single file, css rules can also be applied to a single object, making the modification more precise.
This can be noticed in the Hello.js page, where some specific indications have been added to the title.

```
<h1 className="display-3 text-center" style={{"color":"white"}}>
[...]Welcome to the questionnaire of Indoor Environmental Quality!
</h1>
```

These characteristics in the h1 tag are divided into two different kind of rules: "display-3 text-center" are class names used by Bootstrap in order to enforce rules about text styling and layout. The "style" tag instead enforces a rule created by me in order to make the text within the tag appear white.

The choice about where to place said modification should not be casual. A priority order exists in order to make sure that, when two rules are in conflict, the more specific one will be applied over the general one. This order can be non-trivial, because it depends not only on single rules, but also on their combination or on the presence of keywords such as !important.

Bootstrap is a CSS framework largely used for the building of webpages.
Bootstrap makes it so that, by assigning one of the many provided classes to an object, the associated style will be applied. To make an example, if I associate a button with the class btn-danger, it will turn red.
Although Bootstrap is often used because of its responsiveness, together with the pleasant aesthetic of its components, in the present case it has been chosen mostly because of the intuitive management of the pages' spaces. Bootstrap's CSS classes, in fact, also allow organizing the page dividing the space in rows and columns. Columns, in particular, are 12 and their width depends on the available space. This makes it possible to partition the page in proportional blocks. This is particularly notable in pages such as ThanksEmail.js:

```
<div className="row h-25" />
<div className="row h-50 align-items-center">
<div className="col-12">
/*thank you message*/
</div>
<div className="row">
```

```
<div className="col-lg-3 col-1"/>
<div className="d-grid col-lg-6 col-10">
/*home button*/
</div>
<div className="col-lg-3 col-1"/>
[...]
```

This divides the page in:

- an empty container as tall as the 25% of the available space
- a container as tall as the 50% of the available space containing:
  - A title as wide as the entire page containing a Thank You message
  - A row divided as follows:
    - 1/4 of the page empty, which becomes 1/12 of the page on medium and smaller screens
    - a button as large as 1/2 of the available space, which becomes 5/6 of the page on medium and smaller screens
    - the remaining 1/4 of the page empty, which also becomes 1/12 of the page on medium and smaller screens

This allows to organize the space proportionally among the objects and also to choose these proportions according to the width of the visualization screens.
This aspect of Bootstrap makes it suitable for responsive designs, but the tight requirements on a mostly horizontal layout made it so that the application can be optimally visualized only on screens with a horizontal layout.

Chapter 5

User interface

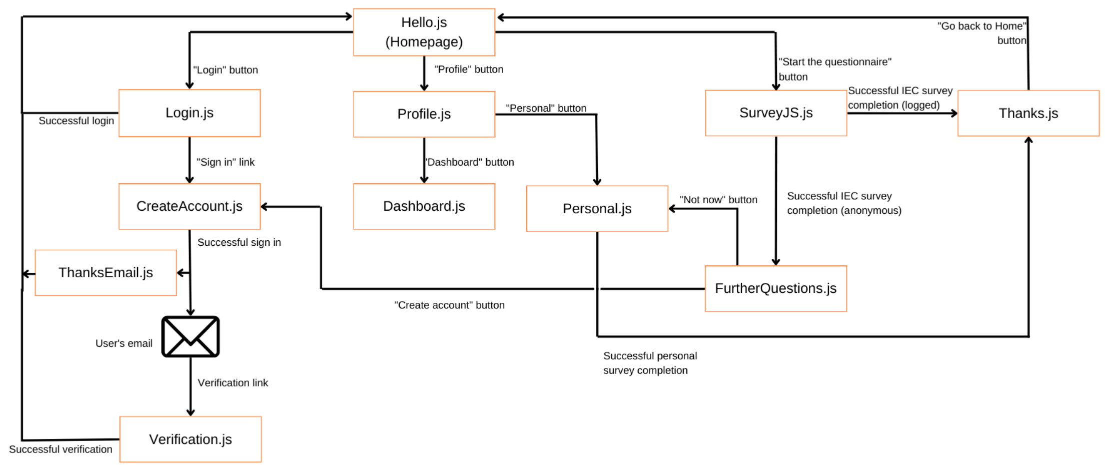


Figure 5.1: Overview of the routes in the application

The User Interface presents itself differently depending on the kind of device that the application is accessed from.
The application looks for device credentials at the first access to the website. If these are available, a request is forwarded to Amazon Cognito in order to authenticate the device and store information such as a reference number to identify the closest multisensor station.
This generates a jwt which will then be used in those requests that require the device to be authenticated, such as the POST on /survey. Whether the device is authenticated or not is also an important indicator to tell which pages should be displayed or not: a non authenticated device, in fact, won't allow to display the Indoor Environmental Comfort survey as it wouldn't be possible to correlate said

data to any multisensor.

5.1 Unregistered device

If the device is not authenticated and no jwt is available for it, the homepage will look like this:

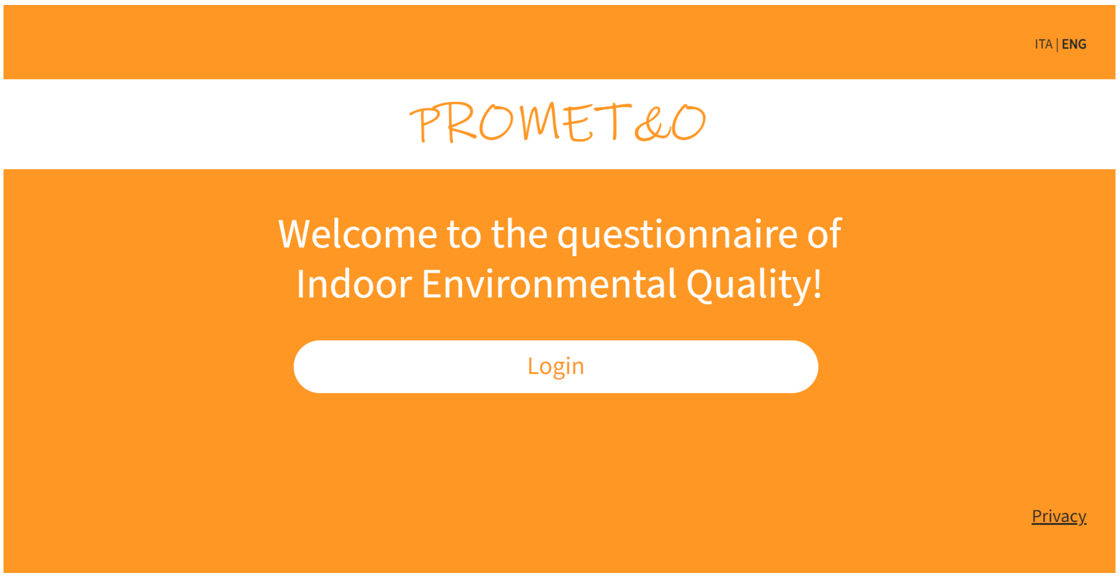
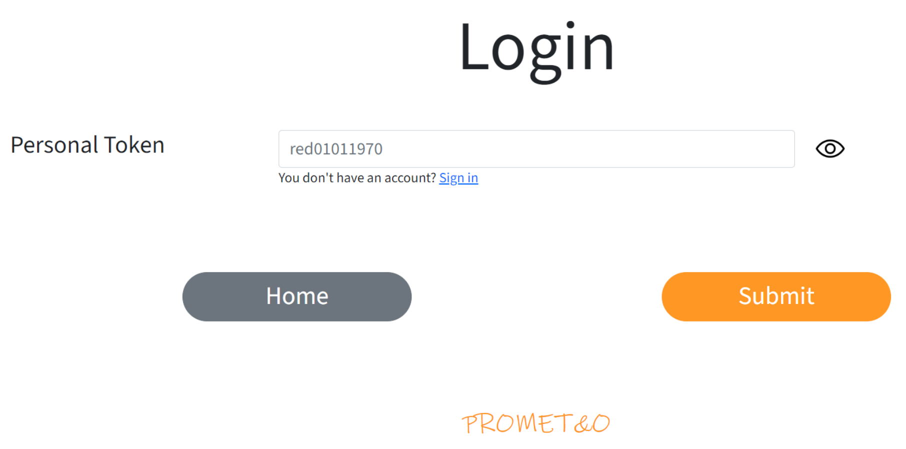

ITA | ENG
PROMET&O
Welcome to the questionnaire of Indoor Environmental Quality!
Login
Privacy

Figure 5.2: Homepage on unregistered device

The user can either create an account by filling the first survey or through the link in the Login page.


Login

Personal Token

red01011970

You don't have an account? Sign in

Figure 5.3: Login page

In this page, an API POST call is performed on the resource "/token" of "user-TokenAPI" sending the token we desire to check in the body.
In the API some checks are performed. For example, if there is less than a user or more than one associated to that token or the account hasn't been verified yet, a body with null email and token will be returned, otherwise a body containing email, token and a newly generated jwt will be sent.
By inserting an existing, valid token and pressing “Submit”, the user will be logged and brought back to the homepage.
The jwt token obtained will be used to authorize some API access and to prove the user's identity to Grafana by storing it into the cookies.

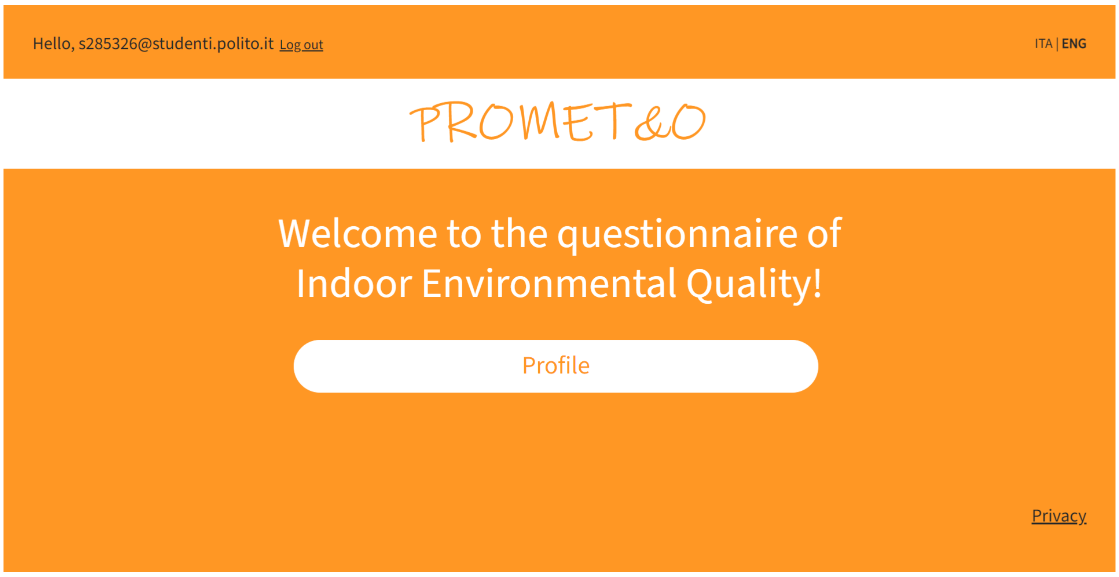
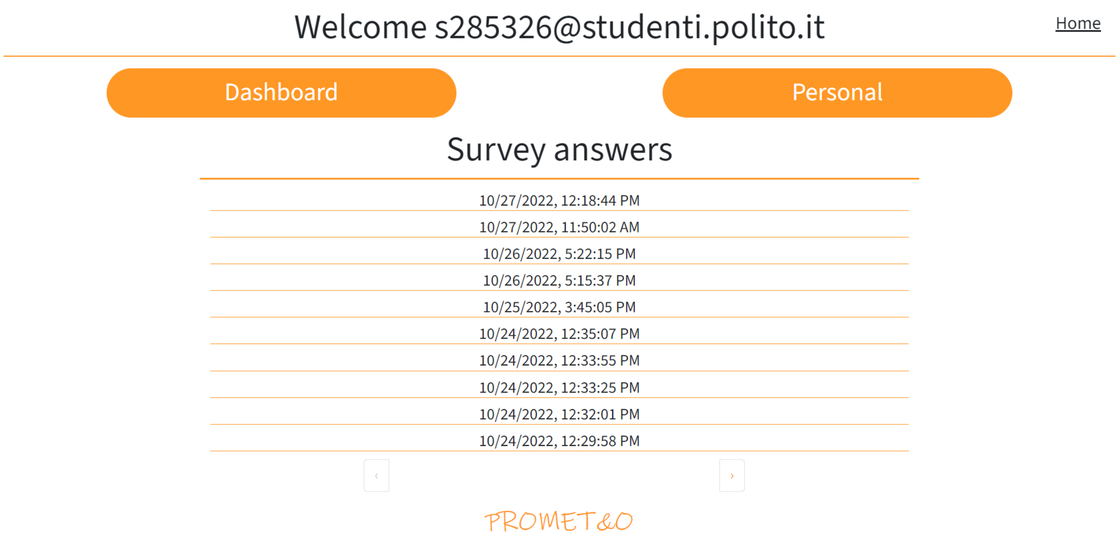

Hello, s285326@studenti.polito.it Log out
ITA | ENG
PROMET&O
Welcome to the questionnaire of Indoor Environmental Quality!
Profile
Privacy

Figure 5.4: Homepage of a logged user on an unregistered device

In this version, the user is only allowed to visit his or her profile by clicking the "Profile" button.

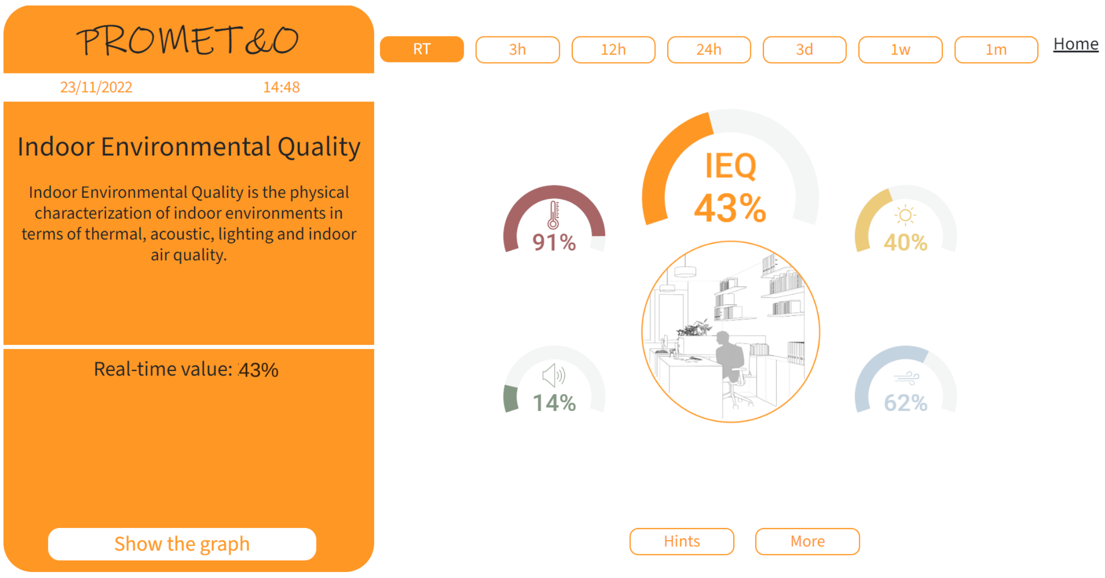


10/27/2022, 12:18:44 PM
10/27/2022, 11:50:02 AM
10/26/2022, 5:22:15 PM
10/26/2022, 5:15:37 PM
10/25/2022, 3:45:05 PM
10/24/2022, 12:35:07 PM
10/24/2022, 12:33:55 PM
10/24/2022, 12:33:25 PM
10/24/2022, 12:32:01 PM
10/24/2022, 12:29:58 PM

Figure 5.5: User's profile

Here the timestamps of past surveys are displayed and two paths are provided: "Dashboard" and "Personal"

In order to display the latest survey completion timestamps, a GET request is forwarded to "/survey/user?user=[email]" of "userTokenAPI" where [email] is the one of the logged user. Moreover, in the header, the user's jwt must be added in the "Authorization" field.
The resulting answers are subsequently sorted by timestamp and grouped in pages of up to ten elements.
By clicking on "Dashboard", the user is lead to the objective data dashboard.



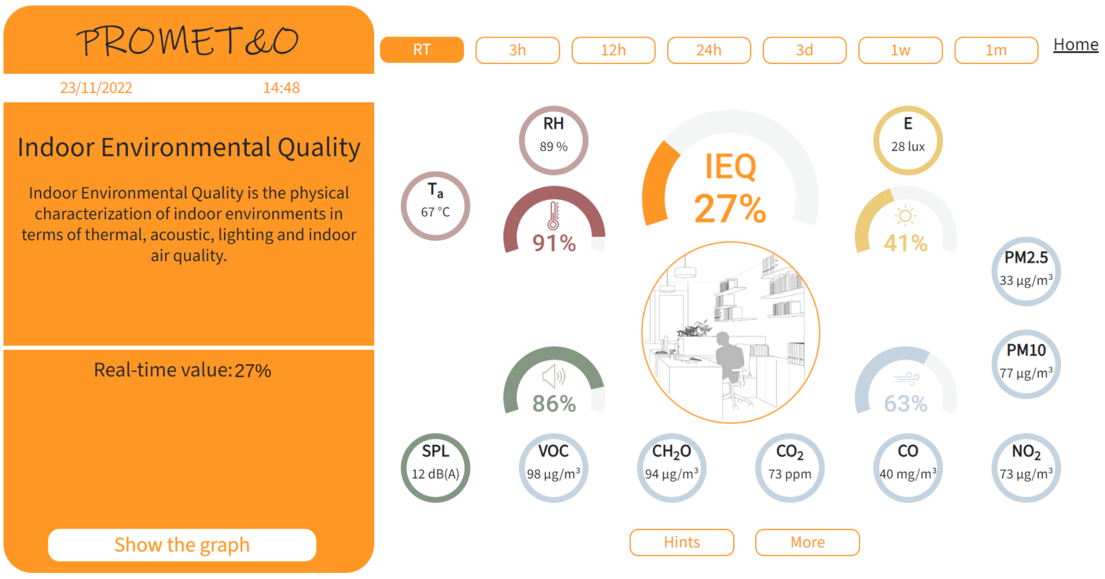
Indoor Environmental Quality

Indoor Environmental Quality is the physical characterization of indoor environments in terms of thermal, acoustic, lighting and indoor air quality.

Real-time value: 43%

Figure 5.6: Dashboard

The Dashboard is used to display the objective data collected by the sensors placed around the building. By clicking on any comfort aspect, related objective measures appear. Clicking on Indoor Environmental Quality's gauge makes them all appear.



Indoor Environmental Quality is the physical characterization of indoor environments in terms of thermal, acoustic, lighting and indoor air quality.

Figure 5.7: Dashboard fully activated

The analyzed time window can change depending on the selected button at the top. By clicking on "Hints" and "More", some additional information about the selected topic is displayed in a pop-up. Hints are randomly selected from a pool of suggestions, More instead is a fixed, more detailed explanation of the desired aspect.

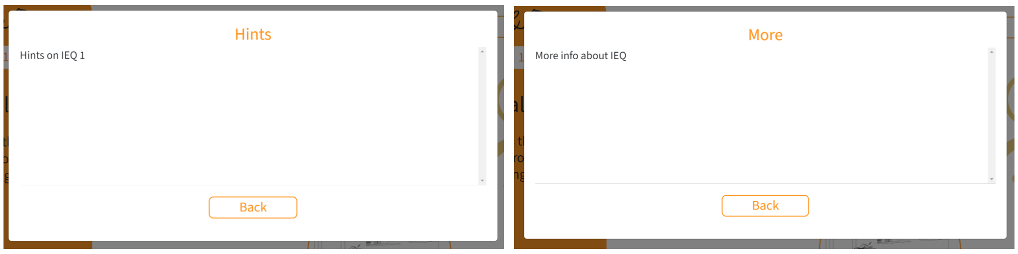
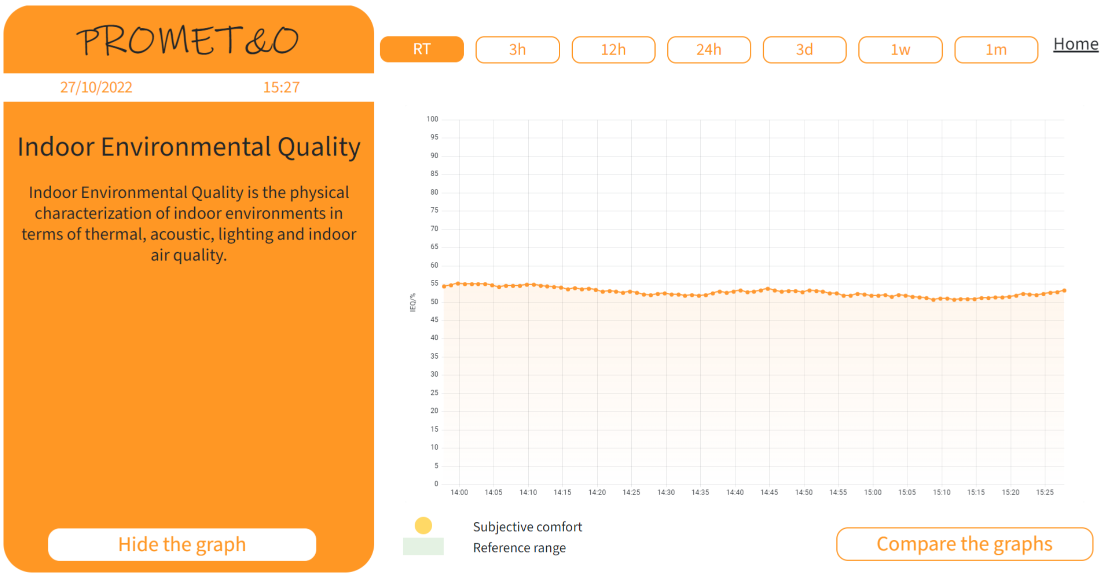


Figure 5.8: Hints and more

By clicking on "Show the Graph", a graph related to the desired topic and time window will be displayed



Indoor Environmental Quality

Indoor Environmental Quality is the physical characterization of indoor environments in terms of thermal, acoustic, lighting and indoor air quality.

Figure 5.9: Graphs visualization

And by clicking “Compare the graphs” the chance will be given to compare the graph to up to three other ones. The other graphs can be related to different measures in the same time window or to different time windows of the same measure.

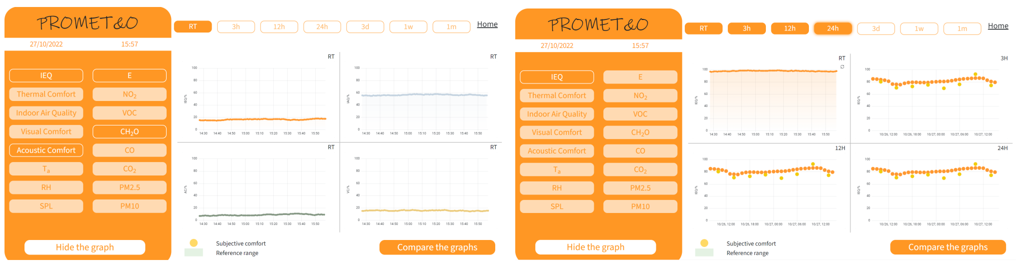
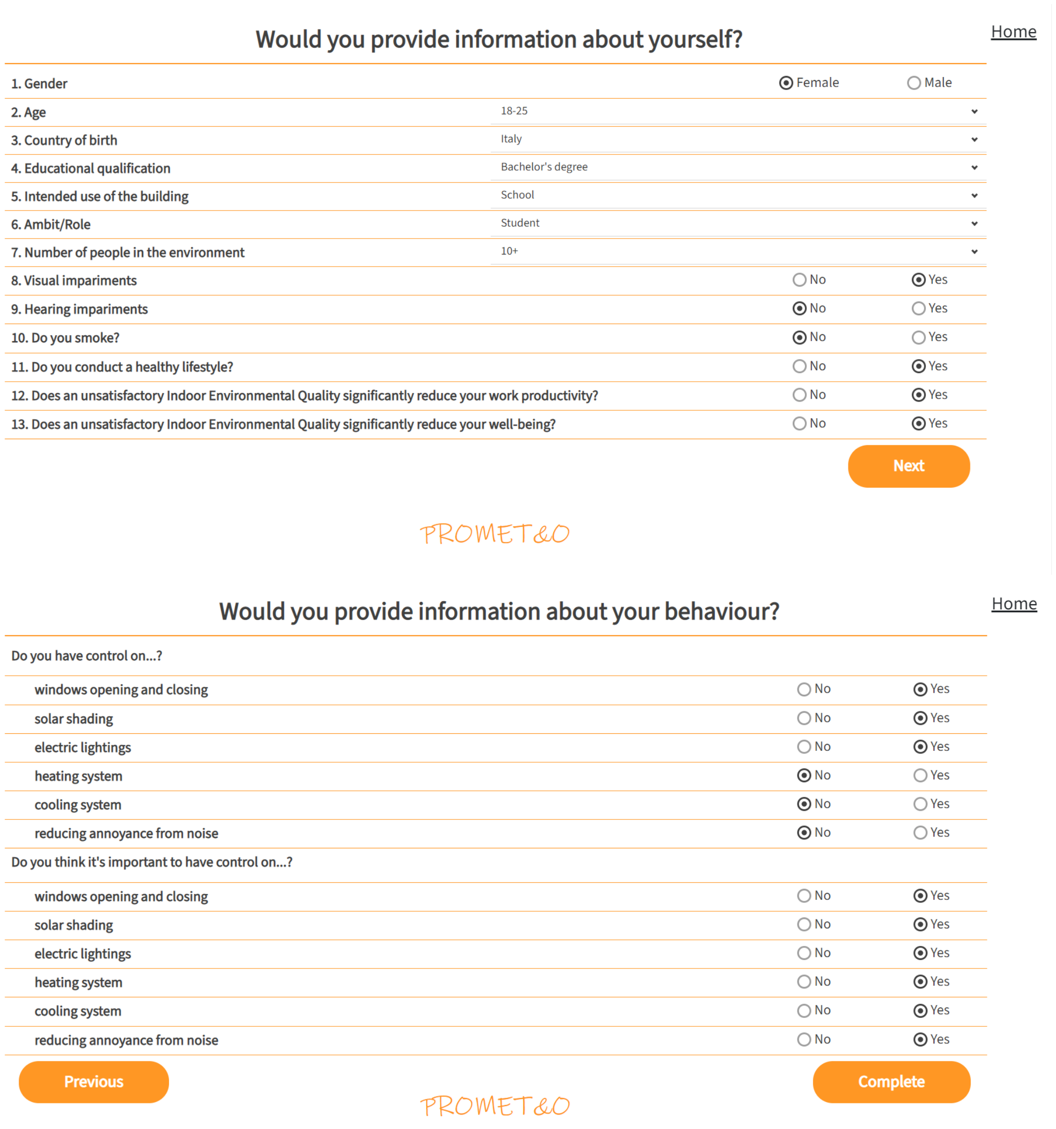


Figure 5.10: Graphs comparison

The “Personal” path of the profile will display the possibility of filling a quick two-pages survey about the users’ habits and environment. If available, the most recent set of answers is displayed as default values, in order to allow the user to change only the specific aspects he wants to update.
In order to do so, a GET request is forwarded to "/personal/personalID?user=[user]" onto the API resource "userTokenAPI", where [user] is the email address of the user who wants to answer the survey. It also requires to send the user’s jwt in the Authorization header for authentication. This returns all previous responses, sorts them by date and selects the newest, if at least one is available.

These old values are then set as the default answer of the survey, allowing the user to update only selected questions, keeping the previous answers for the ones that haven't changed without having to renew them.
To do so, for all the questions present into the survey we perform this operation:
survey.setValue(ov, oldValues[ov]);
where "ov" is the question index and oldValues[ov] is the relative answer.



Would you provide information about yourself?

1. Gender	[✓ Female]	Male
2. Age	18-25	
3. Country of birth	Italy	
4. Educational qualification	Bachelor's degree	
5. Intended use of the building	School	
6. Ambit/Role	Student	
7. Number of people in the environment	10+	
8. Visual impariments	No	[✓ Yes]
9. Hearing impariments	[✓ No]	Yes
10. Do you smoke?	[✓ No]	Yes
11. Do you conduct a healthy lifestyle?	No	[✓ Yes]
12. Does an unsatisfactory Indoor Environmental Quality significantly reduce your work productivity?	No	[✓ Yes]
13. Does an unsatisfactory Indoor Environmental Quality significantly reduce your well-being?	No	[✓ Yes]

Would you provide information about your behaviour?

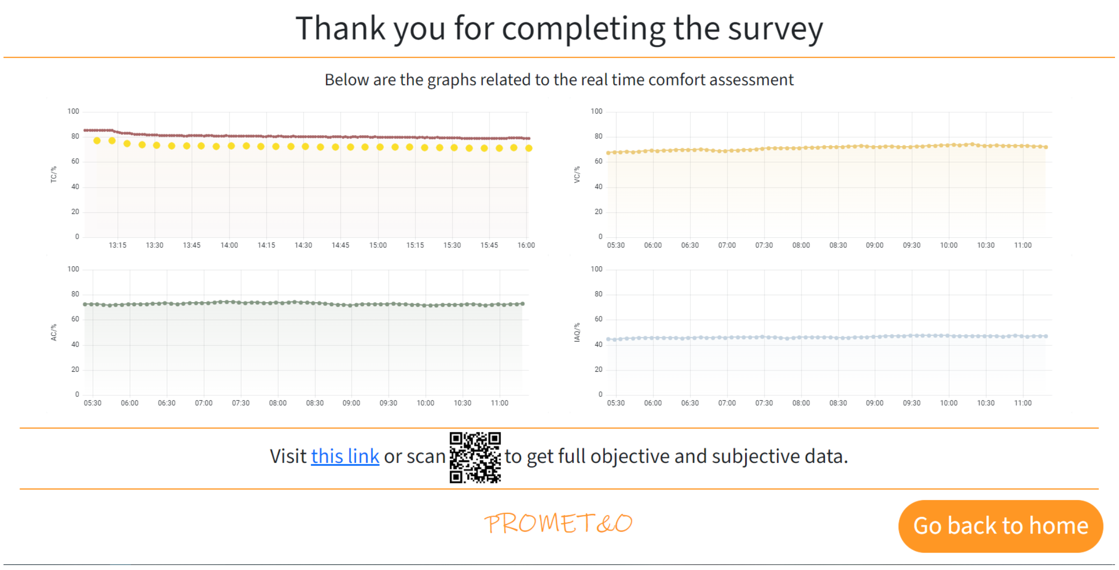
Do you have control on...?		
windows opening and closing	No	[✓ Yes]
solar shading	No	[✓ Yes]
electric lightings	No	[✓ Yes]
heating system	[✓ No]	Yes
cooling system	[✓ No]	Yes
reducing annoyance from noise	[✓ No]	Yes
Do you think it's important to have control on...?		
windows opening and closing	No	[✓ Yes]
solar shading	No	[✓ Yes]
electric lightings	No	[✓ Yes]
heating system	No	[✓ Yes]
cooling system	No	[✓ Yes]
reducing annoyance from noise	No	[✓ Yes]

Figure 5.11: Personal questions' survey

When the survey is complete, sendDataToServer function is called:

```
function sendDataToServer(sur) {
if((user===undefined || user === null) && props.anon===null)
{
    console.log("No user assigned")
    return
}
if(Object.keys(sur.data).length === 0) {
    localStorage.setItem("previousPersonal", true)
    navigate("/thanks")
}
else
{
    let data = sur.data;
    data.personalID = generateResponseId()
    data.user = user===null ? props.anon : user
    data.timestamp = Date.now()
    let init = {
        body: data,
    }
    API.post("userTokenAPI", "/personal", init).then(resp=>{
        localStorage.setItem("previousPersonal", true)
        navigate("/thanks")
    }).catch([...])
}
}
```

This function checks whether a user has been assigned, which should always occur, either in the form of an anonymous user or a logged one.
Another check that is performed is how many answers have been filled: if zero answers have been completed, we simply redirect the user to the next page. Otherwise, a post to "/personal" containing the provided answers is performed before redirecting the user.
Upon completion, a “thank you” page will be displayed. Inside of it, data about the four aspects of environmental comfort are shown in graphs and info about where to know more are provided in the form of a link and a qr code.



Thank you for completing the survey

Below are the graphs related to the real time comfort assessment

Visit this link or scan to get full objective and subjective data.

Figure 5.12: Thank you page

The Thank You page has actually a much more complex structure than what may appear. The useEffect invoked at loading, in fact, represents the end of a multitude of possible paths, which are all analyzed and specific actions are taken. More precisely:

- the user should either be logged (because he or she reached the page through his profile, from the personal questions' survey) or accessing from an authenticated device (because he or she reached the page either at the end of the IEC survey as a logged user or at the end of the personal questions' survey as an anonymous user). Moreover, either the IEC survey or personal survey should have been completed. If none of these cases apply, the user is redirected to home.

- If the user is authenticated and IEC answers are available, elaboration of the received data is performed.
  First, data is elaborated through the function "evaluateComfort", which turns subjective data into percentage indices. This data is then published over a MQTT topic named [multisensor]/questionnaire.

- Afterwards, the received answers to the IEC survey are then sent to the server through the use of a POST request to the path "/survey" of "userTokenAPI". This kind of request requires both the answers as its body and the deviceJWT in the Authorization header.

- Afterwards, in case of success, a GET request is forwarded to

"/survey/user?user=[user]" together with the user jwt Authorization header in order to count how many survey answers have been provided by the user. If the previous one was the first answer, a mail is sent to the user through a POST onto
https://822240w7r0.execute-api.eu-west-3.amazonaws.com
/sampledev/sendEmail
forwarding destination email address, email body and subject.
At this stage, it is expected that the user is logged and thus the jwt will be available.

- If the user is not logged, his or her personal answers have already been sent, thus the page does nothing else.

The user is later allowed to be redirected to home by clicking a "Go Back Home" button.

5.2 Registered device

If the device is, instead, authenticated, the homepage that will greet the user is the following:

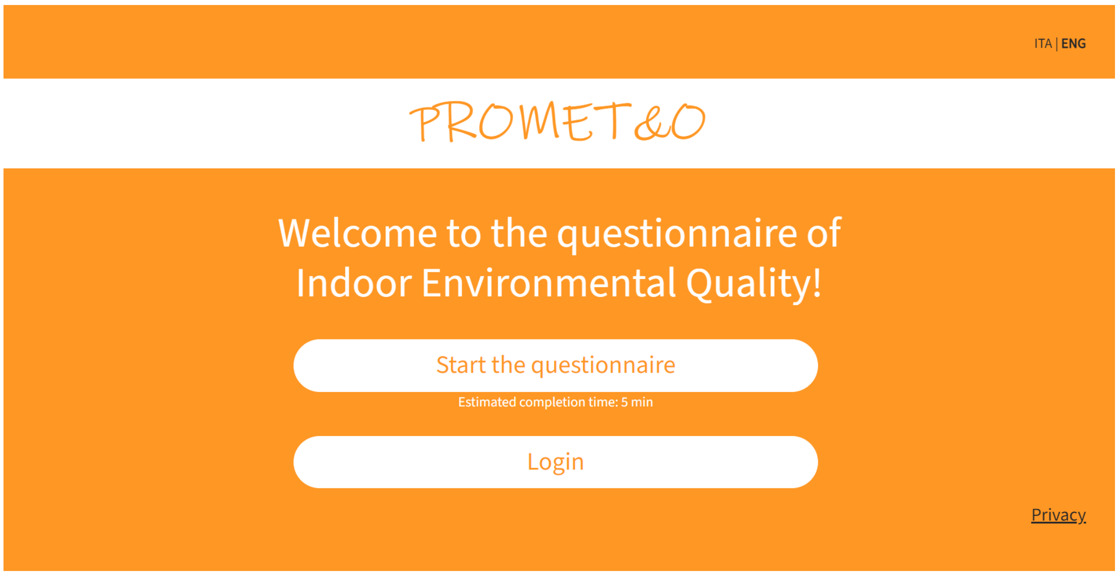

ITA | ENG
PROMET&O
Welcome to the questionnaire of Indoor Environmental Quality!
Start the questionnaire
Estimated completion time: 5 min
Login
Privacy

Figure 5.13: Homepage of a registered device

The questionnaire can be completed even by a non-registered user by clicking “Start the questionnaire”

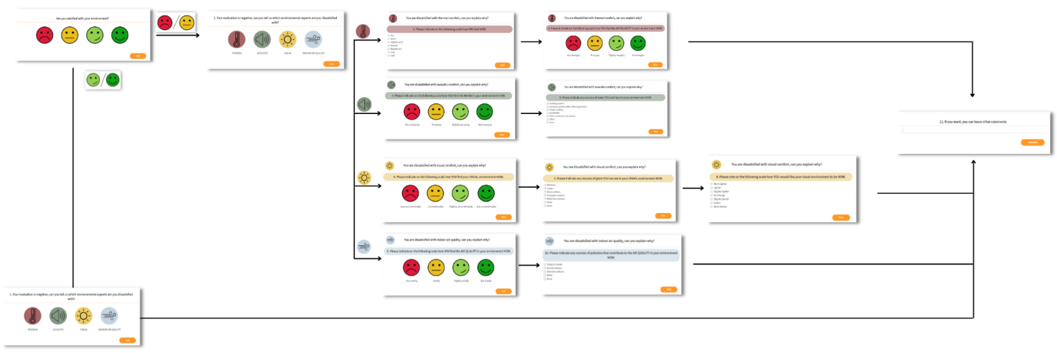
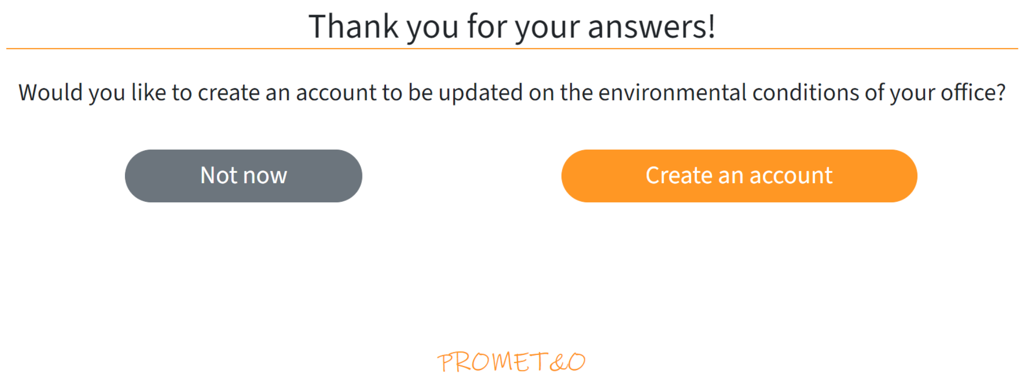


Figure 5.14: IEC survey structure

Completing successfully the questionnaire will lead an unlogged user to the following page:

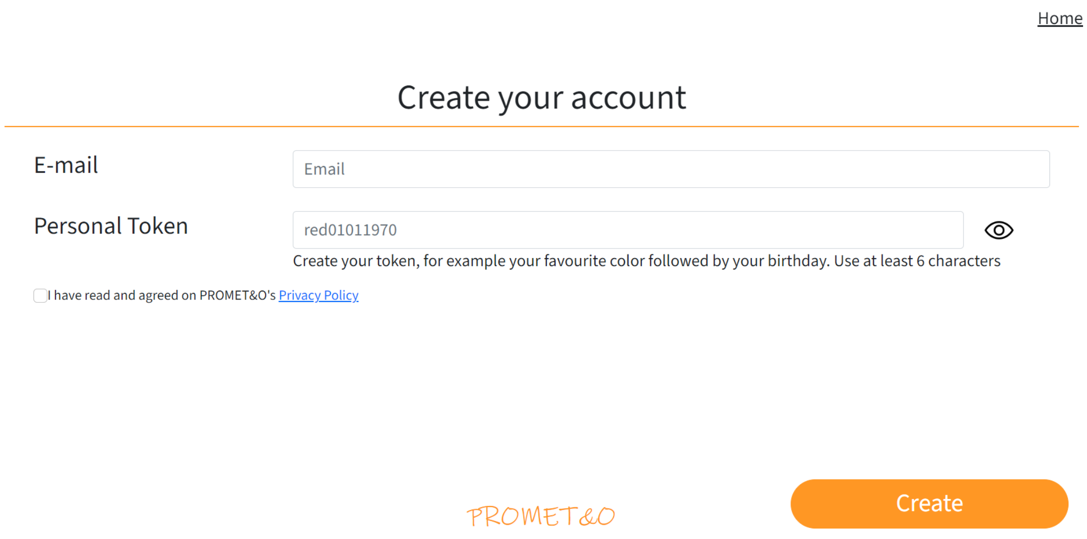

Thank you for your answers!

Would you like to create an account to be updated on the environmental conditions of your office?

Figure 5.15: End of questionnaire page for unlogged users

By selecting “Not now” the user will be led to the lifestyle survey page in order to provide anonymous information about his habits.
Before redirecting the user to the personal survey page, the answer he or she gave anonymously are sent to the database through the use of a POST request to "/survey" and the same anonymous id is used as the identificator of the user to whom the personal answers provided belong.

By selecting “Create an account” the user will be redirected to the sign in page.


Home

Create your account

E-mail	Email
Personal Token	red01011970

Create your token, for example your favourite color followed by your birthday. Use at least 6 characters

☐ I have read and agreed on PROMET&O's Privacy Policy

Figure 5.16: Sign in page

By inserting a mail address and creating a personal token, after accepting PROMET&O’s Privacy policy, the user will be registered inside the application and will be able to log in whenever he pleases.
A POST request is placed onto the path "/token" of "userTokenAPI". There, checks are performed in order to understand whether somebody with the same email or token already exists. If it’s the case, an error message is returned, otherwise the new data is inserted into the database. Together with the chosen email and token, the server also receives a 6 letters code to be used for verification and two booleans:

- "active", set to false, which represents whether the account has been verified
- "subscribed", set to true but temporarily unused, which represents whether the account has accepted to receive email communications from PROMET&O

Afterwards, a mail with a verification link is sent to the indicated email address

through the API available at the address:
https://822240w7r0.execute-api.eu-west-3.amazonaws.com
/sampledev/sendEmail
If answers are also available to the Indoor Environmental Comfort survey, they are sent to the database with a POST to "/survey"
After signing in, a thank you message is prompted and the user is led back to home.

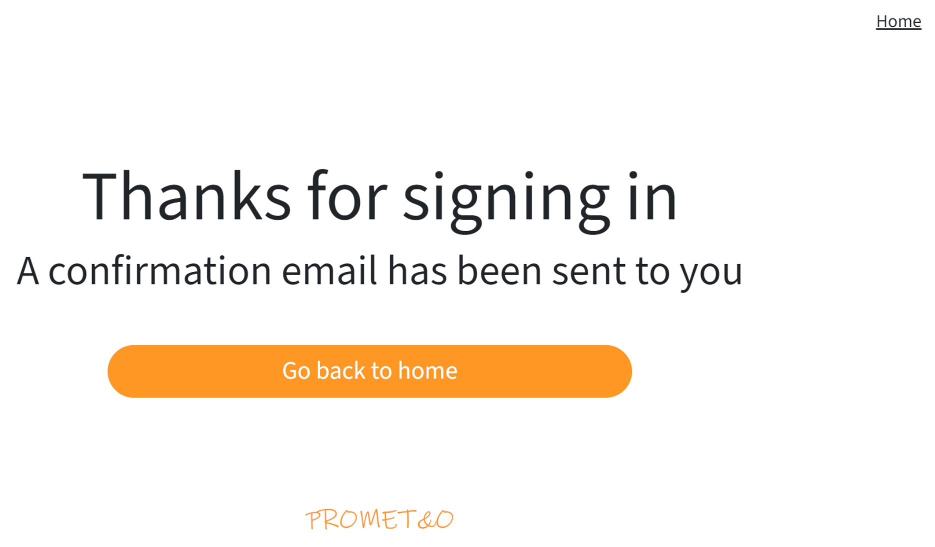


Figure 5.17: End of registration page

By clicking on the verification link received through email, the user is sent to the Verification.js page.
Here, a PUT is sent to "/token". Checks are performed in order to understand whether the email and code are actually associated and to understand if the user had been verified previously. If the code matches and the user was not verified yet, the user is set as active and a message displaying the successful verification of the account is displayed.
Otherwise, if the account is active, a message saying that the account has already been verified will be displayed.
Finally, if no match was found, the account verification is marked as unsuccessful and a related message will appear.

Chapter 6

Conclusions

During this work of thesis, several possible combinations for the implementation of PROMET&O's web application have been analyzed.
Through an accurate analysis of possible tools, both for the frontend and the backend, several options have been taken into consideration. Internet, in fact, showcases a large amount of possible, valid solutions.

The usage of Amazon's Web Services for the serverless implementation of our backend has posed several challenges because of the novelty in the approach, but has also proved to be a reliable solution, adaptable to a multitude of possible use cases.
This makes it so that both the present scenario, with a limited amount of users, and possible future developments including a larger user pool makes this solution viable and interesting.

DynamoDB, AWS Lambda's functions and API Gateway are able to support ever changing scenarios thanks to their automatic scaling of resources.
Amplify allows to deploy new application versions in a matter of minutes, without needing to halt the application and granting a seamless transition from a software version to another.
Cognito, in the meantime, is a useful tool in order to manage device authentication.
Route53 allowed us to purchase a domain and generate new subdomains,
paris.prometeo.click and ns01.prometeo.click, to deploy our applications.
That very same domain has also been used by Amazon SES to send emails to our users.

Through the Amplify CLI, the creation and correlation of these resources in order to be able to save data in our database, read information and interact with our users has been intuitive and simple.

Amazon's web interface, moreover, allowed us to keep track of the state of our resources and create different environments which correspond to different flavours of our application.
The creation of "paris" and "ns01" as distinguished versions of the same project made it so that the data from Paris, which is more of a development stage, do not get mixed up with the ones from ns01, which was a testing implementation deployed at a customer's office.
In doing so, the two paths have been able to evolve independently, according to the needs of the head of project and our customers without interference.

SurveyJS, instead, represents a simple, powerful tool for the automatic implementation of surveys.
Surveys can be heavily customized by choosing among a wide range of possible question types, defining logical paths among questions and integrate the existing aesthetic with additional style rules defined in order to make sure that our survey appears exactly like desired.
Thanks to SurveyJS we were able to implement the Indoor Environmental Comfort survey in a way that specific questions are displayed according to the user's previous answers, for example displaying the question "Your evaluation is positive, can you tell us which environmental aspects you consider particularly satisfying?" only if the question "Are you satisfied with the thermal, acoustic, visual, and air quality conditions in your environment?" received a positive or mildly positive response.
At the same time, in the personal questions' survey, question number 6 "Ambits/Roles" displays different options depending on the answer provided to the previous question, "Intended use of the building".
This allows to have a large control on how the survey experience is built for the user, in order to be able to obtain large pieces of information in a quick, intuitive way.
The use of colors and icons in order to guide the user should also make the survey experience simple, fast and pleasant, objectives which were primary in the context of the project, as they help stimulate a proactive approach in the buildings' occupants.
Their role, in fact, is fundamental in order to collect high quality data and, in the future, be able to elaborate correlations among environmental conditions, habits and perceived comfort.

A proactive approach in the users is required since the earliest test stages.
Their opinion not only on Environmental Comfort, but also on User Experience, achieved through a personal feedback loop and critical confrontation will lead to improvements in the platform that will grant a more and more pleasant interaction with the web portal deployment after deployment.

Bibliography

[1] Amazon Web Services Inc. *What is Amazon API Gateway?* URL: https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html. (accessed: 01.12.2022) (cit. on p. 9).

[2] Amazon Web Services Inc. *What is AWS Lambda?* URL: https://docs.aws.amazon.com/lambda/latest/dg/welcome.html. (accessed: 01.12.2022) (cit. on p. 9).

[3] Amazon Web Services Inc. *What is Amazon DynamoDB?* URL: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html. (accessed: 01.12.2022) (cit. on p. 9).

[4] Amazon Web Services Inc. *What is Amazon Cognito?* URL: https://docs.aws.amazon.com/cognito/latest/developerguide/what-is-amazon-cognito.html. (accessed: 01.12.2022) (cit. on p. 9).

[5] Amazon Web Services Inc. *What is Amazon Route 53?* URL: https://docs.aws.amazon.com/Route53/latest/DeveloperGuide/Welcome.html. (accessed: 01.12.2022) (cit. on p. 9).

[6] Amazon Web Services Inc. *What is Amazon SES?* URL: https://docs.aws.amazon.com/ses/latest/dg/Welcome.html. (accessed: 01.12.2022) (cit. on p. 9).

[7] Google. *Make data-driven decisions, in Google Sheets.* URL: https://www.google.com/sheets/about/. (accessed: 01.12.2022) (cit. on p. 12).

[8] The Apache Software Foundation. *CouchDB Relax.* URL: https://couchdb.apache.org/. (accessed: 01.12.2022) (cit. on p. 12).

[9] Oracle. *NoSQL Database Cloud Service.* URL: https://www.oracle.com/database/nosql/. (accessed: 01.12.2022) (cit. on p. 13).

[10] The International Business Machines Corporation (IBM). *IBM Cloudant.* URL: https://www.ibm.com/cloud/cloudant. (accessed: 01.12.2022) (cit. on p. 13).

[11] Inc. Couchbase. *Couchbase.* URL: https://www.couchbase.com/. (accessed: 01.12.2022) (cit. on p. 13).

[12] Florian pharindoko Fuß. *json-serverless*. URL: https://github.com/pharindoko/json-serverless. (accessed: 01.12.2022) (cit. on p. 13).

[13] Inc. DataStax. *DataStax*. URL: https://www.datastax.com/. (accessed: 01.12.2022) (cit. on p. 14).

[14] restdb.io. *RestDB.io*. URL: https://restdb.io/. (accessed: 01.12.2022) (cit. on p. 14).

[15] Inc. MongoDB. *MongoDB Atlas Database*. URL: https://www.mongodb.com/atlas/database. (accessed: 01.12.2022) (cit. on p. 14).

[16] LimeSurvey GmbH. *LimeSurvey- Turn questions into answers*. URL: https://www.limesurvey.org/. (accessed: 01.12.2022) (cit. on p. 16).

[17] Devsoft Baltic OÜ. *SurveyJS - Javascript Libraries for Surveys and Forms*. URL: https://surveyjs.io/. (accessed: 01.12.2022) (cit. on p. 16).

[18] Google LLC. *Get insights quickly, with Google Forms*. URL: https://www.google.it/intl/en-GB/forms/about/. (accessed: 01.12.2022) (cit. on p. 17).

[19] formsflow.ai. *Best Free Open Source Low Code Platform | Formsflow.AI*. URL: https://formsflow.ai/. (accessed: 01.12.2022) (cit. on p. 18).

[20] Form.io LLC. *Form.IO - A Form and Data Management Platform*. URL: https://www.form.io/. (accessed: 01.12.2022) (cit. on p. 18).

[21] Zoho Corporation Pvt. Ltd. *Form Builder | Create Free Online Forms - Zoho Forms*. URL: https://www.zoho.com/forms/. (accessed: 01.12.2022) (cit. on p. 19).

[22] Budibase. *Budibase | Build internal tools in minutes, the easy way*. URL: https://budibase.com/. (accessed: 01.12.2022) (cit. on p. 19).

[23] Jotform Inc. *Free Online Form Builder Form Creator | Jotform*. URL: https://www.jotform.com/. (accessed: 01.12.2022) (cit. on p. 19).

[24] Inc. Gitana Software. *Alpaca Forms - Easy Forms for jQuery*. URL: http://www.alpacajs.org/. (accessed: 01.12.2022) (cit. on p. 19).

[25] OhMyForm. *OhMyForm | OhMyForm is the best open source form solution for the web*. URL: https://ohmyform.com/. (accessed: 01.12.2022) (cit. on p. 19).

[26] Inc. Formium. *Formik: Build forms in React, without the tears*. URL: https://formik.org/. (accessed: 01.12.2022) (cit. on p. 20).

[27] Inc. Amazon Web Services. *Amplify CLI - AWS Amplify Docs*. URL: https://docs.amplify.aws/cli/. (accessed: 01.12.2022) (cit. on p. 24).

[28] Amazon Web Services Inc. *AWS Pricing Calculator*. URL: `https://calculator.aws/#/`. (accessed: 01.12.2022) (cit. on p. 53).

[29] Amazon Web Services Inc. *Deploy a Node.js Web App*. URL: `https://aws.amazon.com/getting-started/hands-on/deploy-nodejs-web-app/`. (accessed: 21.11.2022) (cit. on p. 53).

[30] Inc. Meta Platforms. *React – Una libreria JavaScript per creare interfacce utente*. URL: `https://it.reactjs.org/`. (accessed: 01.12.2022) (cit. on p. 61).