

ALMA MATER STUDIORUM · UNIVERSITÀ DI
BOLOGNA

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

**Sensorworker:
An Integrated Crowdsensing
Platform**

Relatore:
Dott.
FEDERICO MONTORI

Presentata da:
FABIO MIRZA

Sessione IV
Anno Accademico 2021/2022

To Claudia ...

Abstract

Sensorworker is an innovative Mobile Crowdsensing platform integrated with the services offered by the popular Crowdsourcing portal Microworkers. The platform allows for the creation and management of Crowdsensing campaigns using the workers available on Microworkers in order to address a critical issue in crowdsensing systems, namely the high participation requirement. The thesis clarifies the meaning of Crowdsourcing, Mobile Crowdsensing, and how Sensorworker solves various critical issues. The system architecture, sensors used, and database structure are examined. Subsequently, an analysis of the interaction between different users and the platform itself is carried out. Finally, the structure of the back-end server and the results of the platform tests are presented.

Introduction

We are surrounded by intelligent devices, capable of connecting to wide networks through which fast and reliable information transfer is possible. Furthermore, such devices are often times equipped with powerful and precise sensors, embedded into their hardware in order to provide interesting and novel features to the user. These conditions have created a perfect environment for the rise of a specific technology, known as *Mobile Crowdsensing* (MSC), defined as the paradigm in which a community leverages mobile devices with sensing and computing capabilities to collectively share data and extract intelligence in order to study and measure phenomena of common interest [7].

As the name implies, MSC is composed of three core concepts

Mobile : the participators and the devices used for the sensing activity must be on moving status.

Crowd : the participators must form a large group of individuals, each equipped with sensing capabilities. Different preferences of the participators might lead to an uneven distribution of data [8].

Sensing : the participators are required to perform some simple sensing task without massive computation.

Sensorworker have been developed to perform MSC activities in a controller and integrated environment in order to harness the wide user base of micro-job platforms such as Microworkers, which provide a convenient and

centralized location to channel their *workers* into MSC activities. Sensorworker allows the creation of configurable MSC campaigns, which are transparently forwarded into Microworkers, leaving the so-called *Crowdsourcer* with the only task of rating the work performed by the workers. In this paper, we will explore the architectural needs and complexities required for the development of Sensorworker and its integration with other platforms. We will review the user interactions and the requirements to effectively store and retrieve the sensory data produced by the workers. We will conclude with some interesting tests on the platform in order to understand the benefits and challenges of Sensorworker.

Contents

Introduction	i
1 State of Art	1
1.1 Crowdsourcing	1
1.1.1 Microworkers.com	3
1.2 Mobile Crowdsensing	3
1.3 Sensorworker	5
2 System Architecture	6
2.1 Technologies and MVC	6
2.2 ER Model and Database	8
2.2.1 Crowdsourcer	9
2.2.2 Campaign	10
2.2.3 Job	12
2.2.4 Sensor Data	13
2.3 Sensors	13
2.3.1 Basic Sensors	14
2.3.2 Other Sensors	16
3 Interactions Flow and Client Implementation	17
3.1 Crowdsourcer Interactions	17
3.2 Worker Interactions	22
3.2.1 Accessing Sensorworker	23

4	Back-end Implementation	30
4.1	Controllers	30
4.2	Routers	35
4.3	Services	35
5	Testing	37
5.1	Results	38
	Conclusions	41
	Bibliography	42

List of Figures

2.1	Model-View-Controller pattern for Sensorworker	7
2.2	Sensorworker pseudo E-R model	9
3.1	Registering page for the crowdsourcer	18
3.2	Campaign creation form	19
3.3	Crowdsourcer Dashboard	21
3.4	Rating panel	22
3.5	Slot status transitions	23
3.6	Preview iframe	25
3.7	Non-preview iframe	26
3.8	SensorCollectQR page	27
3.9	Worker interaction flowchart	28
3.10	SensorCollect page	28
5.1	Testers confidence level on technological know-how	38
5.2	Expected payment for Campaign 1	39

Listings

4.1	Base controller	30
4.2	getFilteredCampaigns function	31
4.3	getCampaigns function	32
4.4	getActiveJobs function	33
4.5	postReading function	33
4.6	guard function	34
4.7	Router for the crowdsourcer controller	35
4.8	Guard on the Crowdsourcer router	35

Chapter 1

State of Art

In order to explain the functionality and purpose of Sensorworker, we first need to introduce a few concepts and their critical issues.

1.1 Crowdsourcing

Today, we live in a society where connectivity between individuals is so ubiquitous that it feels transparent and natural. With the dramatic rise in communication, especially after the advent of the so-called Web 2.0, user-generated content and the "Power of The Crowd"[11] have contributed to the creation of an immense pool of data. Different methods have been developed to harness this collective knowledge. One of these methods, named **crowdsourcing**, represents the act of a company or institution taking a function once performed by employees and outsourcing it to an undefined (and generally large) network of people in the form of an open call where the incentive to participate can be monetary and/or non-monetary in nature [4]. Crowdsourcing has also been defined as the application of Open Source principles to fields outside of software [6].

However, these two definitions often fail to describe some other critical aspects that crowdsourcing activities usually require. Specifically, it is possible to define models for crowdsourcing based on how the crowd is gathered,

what it is asked to do, and how it is asked [12]:

Collective intelligence Following this model, a crowd is gathered and conditions are created for that crowd to share their knowledge [6]. Examples can extend from a company that uses an employee suggestion box to a worldwide internet-enabled brainstorming session.

Crowd creation Often confused with the previous model, crowd creation happens when a company turns to its users to create, or co-create, a product or service [6]. A notable example is Threadless.com, where users are asked to create designs for t-shirts and other print products for the crowd who then evaluates the creative work. The design that successfully complete this process are then produced and sold by Threadless.com [4].

Crowd voting This model uses the crowd's judgment to organize a vast quantity of information. It is usually applied together with collective intelligence and crowd creation to quickly parse the vast amount of contributions that can take place. Companies such as HP employ this model for market predictions [12].

Crowdfunding Internet has allowed people to participate in micro-lending: the lending of small sums to help in the creation of a product or service. This can range from helping a band fund its first CD to financing new and exciting tech products [12].

As mentioned above, participation in crowdsourcing activities is usually encouraged through incentives of different kinds. Their nature can be intrinsic to the potential crowdsourcees (e.g., for fun or for willingness to help), or extrinsic, where the activity is purely a means to an end (e.g., to gain benefits such as money or reputation) [4].

Thus, crowdsourcing is accepted as an innovative form of value creation used by firms to tap into an enormous potential of competence and knowledge

[4]. However, the implementation of this disruptive method comes with its drawbacks [4], notably

- Difficulties in calculating project costs
- Uncertainty of crowd structure
- Consideration of legal framework conditions
- Creation of a motivating incentive structure

We will now see a platform that can partially offset these disadvantages.

1.1.1 Microworkers.com

Microworkers.com is a framework to access the crowd where potential crowdsourcers can submit crowdsourcing campaigns and individual tasks [5]. Specifically, Microworkers.com's most interesting feature is the wide and diverse user base it offers to crowdsourcers, with emphasis on the location of the *worker*, the individual member of the crowd. Campaigns can be built and published by the crowdsourcer, with different constraints such as time limits or region locks. The tasks performed by the workers can range from search engine optimization to content creation to product surveys [5]. Once the task is performed and evaluated, the worker is rewarded a sum of money defined by the crowdsourcer during the creation of the campaign.

This guided framework allows the crowdsourcer to specifically select which kind of worker it is wanted for which campaigns, as well as know the cost of the activity beforehand. Most of the complexity regarding the management and logistics of crowdsourcing is delegated to Microworkers.com, leaving the benefits of such methods to the issuer of the campaign.

1.2 Mobile Crowdsensing

Another consequence of the pervasive integration of the Web within our lives is the evolution of an embedded internet, better known as the *Internet*

of Things (IoT), which aims at sensing and interconnecting various physical objects and their surroundings in the real world more comprehensively and on a larger scale [9]. These objects can occur as smartphones, wearable smart devices, sensor-embedded gaming systems, or in-vehicle sensing devices [2].

Especially smartphones and wearables have become essential for our daily activities, from business to entertainment [1]. Replicating the sensing capabilities of an IoT network comprised of these devices using traditional sensor networks for large-scale and fine-grained sensing would require the deployment and maintenance of a large number of sensor nodes, which is logistically and economically undesirable [9]. For example, let's consider the CitySee project: 100 sensor nodes and 1096 relay nodes need to be deployed for CO₂ monitoring of 1 km² [10]. To implement this system in a large urban space, such as the city of Rome with an area of about 1200 km², it would require 120,000 sensor nodes and around 1,300,000 relay nodes to reach full coverage and communication connectivity.

The paradigm that allows ordinary people to contribute data sensed or generated from IoT devices, aggregates and fuses the data in the cloud for crowd intelligence extraction and people-centric service delivery is known as **Mobile Crowdsensing** (MSC) [3]. MSC is suggested as a more flexible and efficient data acquisition, analysis, and application model compared to fixed wireless sensor networks [8].

MSC has many similarities to other research areas. In particular, inspired by the crowdsourcing model detailed in the previous section, crowdsensing specifically aims at the sensing targets, harnessing the highly-distributed mobile devices from ordinary users that can be served as the sensors to collect and upload data [8].

Building a sustainable MSC system requires many solutions to different types of challenges, especially how to recruit enough participators or how to achieve optimal distribution and data quality under budget [8].

1.3 Sensorworker

Sensorworker proposes a Mobile Crowdsensing platform that resolves the issues presented in the previous section by integrating with Microworkers, the micro-task platform detailed above. Microworkers allows developers to build programmatic software in order to create and manage crowdsourcing campaigns, specifying requirements for the workers and allowing a limited budget to be utilized to the greatest extent possible by reviewing in advance the costs of the services and paying only for the crowdsourcer-approved sensing data. Sensorworker integrates within the Microworkers campaigns in a seamless way, providing a web-based interface that does not require any installation or plugin in order to let the worker focus on the tasks at hand.

Chapter 2

System Architecture

In this chapter, we will explore the high-level system designs applied during the development of Sensorworker, starting from the technologies employed.

2.1 Technologies and MVC

As previously mentioned, Sensorworker is a web application comprised of a **front-end** which communicates with a **back-end** that stores and retrieves data from a **database**. These elements are implemented following a Model-View-Controller (MVC) architectural pattern, based on the principle of "separation of concerns" which builds a clear division between the software's business logic and the display.

The main technologies used in each of these components are listed below and will be further explored in the rest of the paper:

- **React:** The front-end framework used for the development of the client. It is component-based and equipped with built-in state management as well as being expandable with many open-source libraries such as React Router for client-side routing or Material UI for ready-made user interfaces. React uses a syntax extension known as JSX, which

is then compiled to pure Javascript, HTML, and CSS in order to be understandable for browsers.

- **Node.js:** The back-end framework adopted for the development of the server. It is one of the most popular javascript framework for back-end development and its use is made easier with libraries such as Express for route handling and Mongoose to interact with a MongoDB database.
- **MongoDB:** A source-available NoSQL database that uses JSON documents and collections to store information instead of classic SQL records ad tables. Following the project’s requirement, different document Schemas have been defined in order to store specific kinds of information (e.g. data about workers, about crowdsourcers).

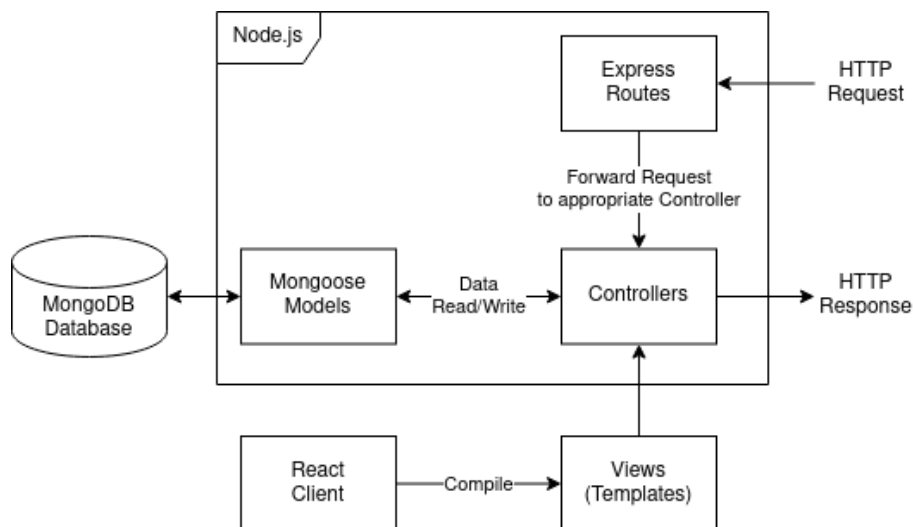


Figure 2.1: Model-View-Controller pattern for Sensorworker

As mentioned before, Sensorworker follows an MVC architectural pattern, where the model is embodied by the Mongoose models that define the collections stored in the Database and through which the data can be manipulated. Controllers use these models to fulfill the requests from the front end

which are intercepted by the Express routers. Based on the kinds of requests the controller receives, it can respond with views or with HTTP response.

2.2 ER Model and Database

Before going any further, we will explore the entities that interact with Sensorworker and the relationships that occur between them.

There are mainly four actors:

- **Crowdsourcer:** The crowdsourcer is the user of the platform, who issues the campaigns, with the appropriate constraints, for the workers on Microworkers to perform. They are also responsible for viewing and rating the data submitted by the workers who take part in their campaigns.
- **Campaign:** The campaigns are connected to and defined by the crowdsourcers. Sensorworker Campaigns are different from the Microworkers campaigns. Each crowdsourcer can have multiple Sensorworker Campaigns. Below we will look more into the relations between Sensorworker's campaigns and Microworkers's campaigns
- **Job:** Jobs are the entity that represents the work done by the Microworker's worker. Understandably, multiple jobs can relate to the same campaign since they usually require the participation of more than one worker. Jobs are the link between the Microworker's worker and the work performed by them within Sensorworker.
- **Sensor Data:** The actual data generated by the worker are held as sensor data. These entities are related to jobs and based on the campaign requirements, multiple sensor data can be related to the same job.

Despite the fact that the database is NoSQL, it is possible to summarize the entities and the relations described so far with the pseudo E-R model

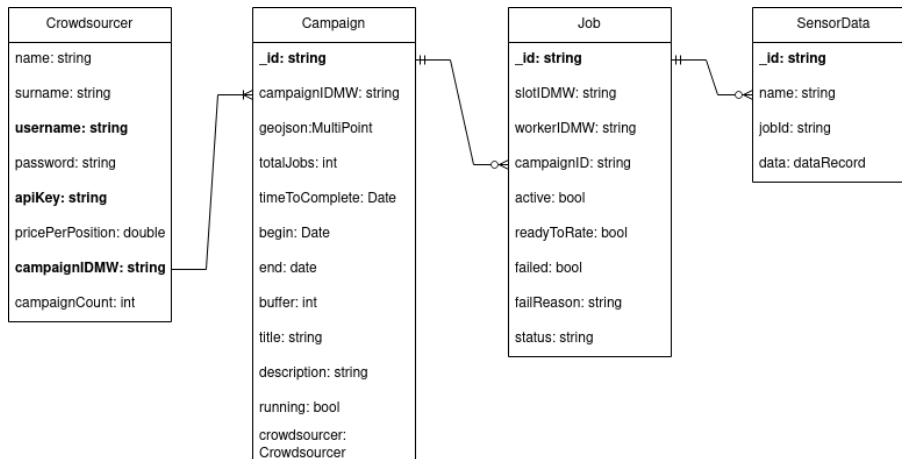


Figure 2.2: Sensorworker pseudo E-R model

shown in Figure 2.2. How and when these entities are created will be explored in the chapter about the flow of execution. Every field for each entity is explained below

2.2.1 Crowdsourcer

Within the database, the crowdsourcer collection holds information about the crowdsourcers who create an account in Sensorworker.

- **name:** the attribute containing the name of the crowdsourcer.
- **surname:** string holding the surname of the crowdsourcer.
- **username:** the username defined by the crowdsourcer during the registration process, it is one of the attributes that can uniquely identify a crowdsourcer.
- **password:** password used by the crowdsourcer to login into Sensorworker.
- **apiKey:** this string is provided by the crowdsourcer during the account creation process. It is the key used to interact with Microworker's API

endpoint on behalf of the crowdsourcer in order to create a Microworkers campaign and to update the number of available positions for it.

- **pricePerPosition**: it is the price that the crowdsourcer is willing to pay to add new positions to the Microworkers campaign. It must be noted that Microworkers adds a 10% charge for each position, and this is taken into account when setting this value.
- **campaignIDMW**: This attribute contains the ID of the Microworkers campaign associated with the crowdsourcer. Specifically, Sensorworker automatically creates a new Microworkers campaign whenever a new crowdsourcer registers on the platform, and the former's id is stored in this field. From then on, every Sensorworker campaign created by that crowdsourcer will be related to that Microworkers campaign. That means that there is a one-to-many relationship between Microworkers campaigns and Sensorworker campaigns. As a matter of fact, whenever a worker takes part in a Microworkers campaign created by Sensorworker, it must then select between the Sensorworker campaigns of the specific crowdsourcer who is the owner of that Microworkers campaign. Similar to the username, this attribute can be used to uniquely identify a crowdsourcer since there is a one-to-one relationship between the crowdsourcer and its related Microworkers campaign.
- **campaignCount**: the number of Sensorworker campaigns created by this crowdsourcer.

2.2.2 Campaign

This collection contains all the information required to define a Sensorworker campaign.

- **_id**: it is the Id that uniquely identifies every Sensorworker campaign. it is automatically generated by the database.

- **campaignIDMW**: the value of this field determines which crowdsourcer is the owner of the Sensorworker campaign. This is because, as previously explained, the campaignIDMW attribute in the crowdsourcer entity can be used to uniquely identify a crowdsourcer. This way, a crowdsourcer can be the owner of many different Sensorworker campaigns.
- **geojson**: This field describes the geographical area in which the worker must be located in order to be able to send sensory data for the campaign. It is basically an array of coordinates, which specify a polygon as the boundary of the area.
- **totalJobs**: a number that indicates how many jobs must be allocated for this campaign. It is used to add positions to the Microworkers campaign whenever a new campaign is created.
- **timeToComplete**: this field suggests how much time is available to the worker when they perform a job for this campaign.
- **begin**: the date in which this campaign can be taken part into. Specifically, it is the date from which this campaign can be listed when a worker starts the Microworkers campaign associated with the crowdsourcer.
- **end**: similar to the previous field, this attribute holds the date at which this Sensorworker campaign should not be available anymore.
- **buffer**: a number that indicates how distant the worker is allowed to be from the geojson in order to still be eligible to take part in this campaign. The buffer is expressed in kilometers, and it is used as one of the criteria when choosing which campaigns are available for which worker. Once they take part in the Sensorworker campaign, they have to be within the polygon defined by the geojson to perform the actual sensing work.
- **title**: the title of the campaign.

- **description**: a string containing the description of the campaign.
- **running**: a boolean value that indicates whether the campaign is running or not. This value is computed when querying a campaign and it is not actually saved within the database.
- **crowdsourcer**: The id of the crowdsourcer that's the owner of this Sensorworker campaign. When required, it is populated with the actual information about the crowdsourcer (username, apikey, etc).

2.2.3 Job

Every Sensorworker campaign has a `totalJobs` field, which indicates how many jobs must be performed for it. The job collection within the database keeps the information about the work performed by the Microworkers worker, joining their id with a particular slot.

- **_id**: the id that uniquely identifies a job. Similar to the campaign's id, it is generated by the database when a job is created.
- **slotIDMW**: when a worker takes part in a Microworkers campaign, it is assigned a slot. The slot's id is saved in this field. The reason for this requirement is explained in the chapter about the flow of interaction.
- **workerIDMW**: as the name suggests, this field saves the id of the worker who performed this job. It is useful to check if the worker has some other active job for the Sensorworker campaign related to this job.
- **campaignID**: the id of the Sensorworker campaign for which the worker is performing the job.
- **active**: as mentioned before, this field is a boolean value that tells whether the job is actively being worked on by the worker.
- **readyToRate**: This field is similar to the previous one. it is a boolean value which indicates whether the job is ready to be rated.

- **failed**: boolean value that tells whether the job has failed. The chapter about the flow of interaction will explain the situations in which this field is used.
- **failReason**: when the previous field is set as true, this field must be non-null. It explains why the job has failed.
- **status**: similar to the **running** field of the campaign entity, this field is not actually saved in the database. Instead, this field is computed based on the value of **active**, **readyToRate**, and **failed** fields.

2.2.4 Sensor Data

As previously anticipated, this entity, with some exceptions, holds the actual data sent by the Microworkers' workers. Every job must have some Sensor Data related to it, based on the requirements of the Sensorworker campaign. For every sensor required in the **sensors** field of the campaign, each job must have corresponding Sensor Data.

- **_id**: a string that uniquely identifies each Sensor Data.
- **name**: the name of the sensor of which this Sensor Data keeps the readings.
- **jobId**: this field relates the Sensor Data to a specific Job.
- **data**: this field is especially important since it saves the readings of the sensors, as well as the timestamp and the location from which the reading originates. The exact structure of this field will be explained in the next section.

2.3 Sensors

Sensors are a vital part of the Sensorworker platform. They provide the sensing data required by the crowdsourcers and are the main value con-

tributed by the workers.

As Sensorworker is a web application, it is limited to the sensors available to a web browser. These sensors and their interfaces are defined through a specific API, the **Sensor API**. Sensorworker employs Sensor API in order to interact with a group of sensors, internally called **Basic sensors**, as opposed to the other group of sensors defined as **Audiovideo sensors**.

2.3.1 Basic Sensors

Sensor API exposes the sensors to the web app through a simple, practical, and uniform interface. Within these sensors, it is possible to distinguish 2 categories: physical sensors and virtual sensors, where the latter is algorithmically computed on the former.

Below we list the physical sensors:

- **Gyroscope**: This interface provides on each reading the angular velocity of the device along all three axes, as well as the timestamp.
- **Accelerometer**: the accelerometer provides on each reading the acceleration applied to the device along all three axes, together with the timestamp.
- **Magnetometer**: this interface provides, on each reading, information about the magnetic field as detected by the device's primary magnetometer sensor, followed by its timestamp. Similar to the previous two interfaces, the readings are defined along the three axes.
- **Ambient Light Sensor**: this interface returns an integer representing the current light level or illuminance of the ambient light around the hosting device, as well as the timestamp of the reading.

We can see that three out of the four physical sensors share the same structure for their readings. This will be exploited when the interactions with the sensors will be defined.

The following are the virtual sensors:

- **Absolute Orientation:** this sensor's readings describe the device's physical orientation in relation to the Earth's reference coordinate system. They are expressed as Hamilton's quaternions and they are based on the values of the Accelerometer and Gyroscope.
- **Relative orientation:** this sensor's readings describe the device's physical orientation without regard to the Earth's reference coordinate system. As the previous sensor, it is expressed in quaternions and is based on the Accelerometer and Gyroscope.
- **Linear Acceleration:** as the name suggests, this sensor is based on the Accelerometer and, as a consequence, its readings are structured as tridimensional values. The difference between Linear Acceleration and the Accelerometer lies in the fact that the former excludes the contribution of gravitational acceleration from the acceleration applied to the device.
- **Gravity:** This sensor provides the same value as the Accelerometer. Its existence is due to the fact that it is logically separated from Linear Acceleration, and it represents a sensor that only expresses the gravitational acceleration applied to the device.

With the virtual sensors as well, it is possible to differentiate two classes based on the structure of their readings. The first two sensors provide readings structured as quaternions and the last two express their readings as three-dimensional data.

Permissions

So far we have discussed which sensors are available to a web browser through Sensor API. However, since web browsers are available on a multitude of devices, not all of them equipped with the described sensors, it is not possible to take their presence for granted. Therefore, Sensorworker is required to check their existence, with the added complexity that some browser

reports their availability even when it is not really present. Lastly, even when the sensor is actually present, the owner of the device can still disable its access to the web browser. In order to tackle all the cases, Sensorworker performs these three checks for every Basic Sensor

1. Verify that the worker has allowed access to the sensors.
2. Verify that the sensor actually exists.
3. Access the sensor and check whether it actually sends any readings.

These checks must be performed before letting the worker send any data through Sensorworker.

2.3.2 Other Sensors

The sensors explained so far are either real or virtual sensors, used through the interface made available by Sensor API. In this section, we will discuss some features that mobile devices are usually equipped with but don't exactly represent a conventional sensor. These features are the **videocamera**, capable of shooting *images* and recording *videos*, and the **microphone**, capable of recording *audio* files. Combining these two it is possible to shoot *audiovideo* files.

The permissions regarding the video camera and the microphone are handled similarly as the Basic sensors, with the added complexity that the worker can not only deny access to the web browser but also the specific web application, which in our case is Sensorworker.

Finally, we mention a different sensor, the **GPS**, which is not considered stand-alone sensory data, but it is read at every reading of every other sensor that has been described so far. This is a requirement of the system since the position of every worker must be checked against the `goejson` defined in the campaign in which the worker who's sending the data is taking part into.

Chapter 3

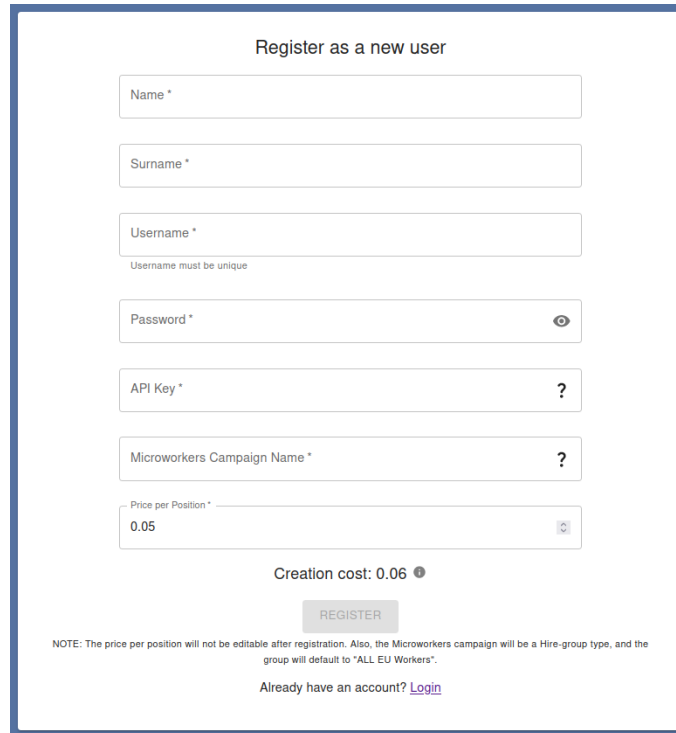
Interactions Flow and Client Implementation

In this chapter, we will finally discuss the flow of interaction, both from the crowdsourcer's and the worker's point of view. Sensorworker's primary users are the crowdsourcers, thus they must interact only with the interfaces made available to them by Sensorworker. On the other hand, the workers who want to take part in a Sensorworker campaign are required to do it through Microworkers. As a consequence, the interaction between the worker and Sensorworker is vastly more complex compared to the crowdsourcer's, since in the former's case the web app must be woven together with the user interface and the navigation system of Microworkers.

3.1 Crowdsourcer Interactions

Since the crowdsourcers are the primary user of the platform, they are confined to the web pages available to them. This makes the handling of their interaction rather manageable. As previously mentioned, a crowdsourcer that wants to register on Sensorworker must already have an account on Microworkers in order to harness its userbase.

Crowdsourcer registration



Register as a new user

Name *

Surname *

Username *

Username must be unique

Password *

API Key * ?

Microworkers Campaign Name * ?

Price per Position *
0.05

Creation cost: 0.06

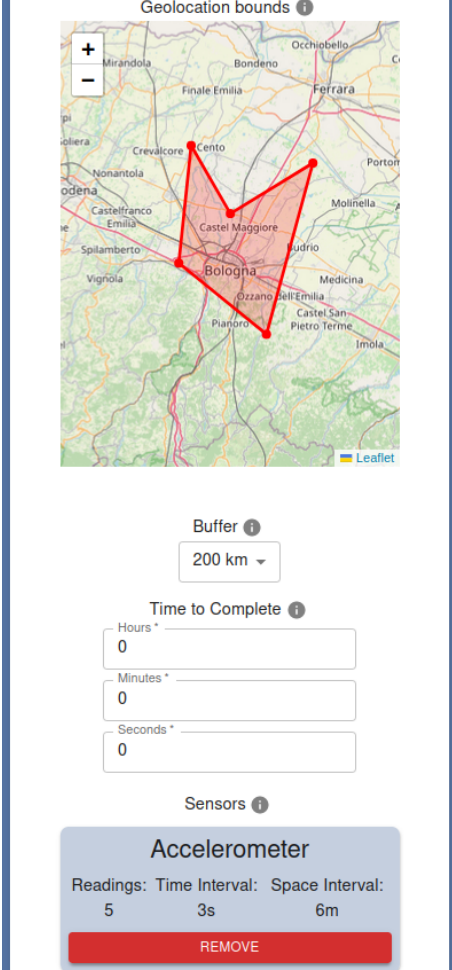
NOTE: The price per position will not be editable after registration. Also, the Microworkers campaign will be a Hire-group type, and the group will default to "ALL EU Workers".

Already have an account? [Login](#)

Figure 3.1: Registering page for the crowdsourcer

When a crowdsourcer registers, it is asked for a number of details. Specifically, the `username` is checked for duplicates in the database, in which case the crowdsourcer is notified and the registration process becomes unavailable. If every field is properly filled, a final check is performed on the `apiKey` in order to ensure that it is actually a key associated with a Microworkers account. If this checkup is successful, Sensorworker create a new Microworkers campaign equipped with an external template that is used to direct the potential workers to the list of Sensorworker campaigns that will be made by this Crowdsourcer, where they can choose which one to take part in. Finally, a new crowdsourcer document will be created in the database with the `campaignIDMW` field set as the id of the newly created Microworkers campaign. Figure 3.1 shows the registration form presented upon connecting.

Campaign creation



The screenshot displays a web interface for creating a campaign. At the top, a map titled "Geolocation bounds" shows a red polygon defining a region around Bologna, Italy. Below the map, there are several input fields and a sensor selection section. The "Buffer" is set to 200 km. The "Time to Complete" section has three input fields for Hours, Minutes, and Seconds, all currently set to 0. The "Sensors" section shows "Accelerometer" selected, with a "REMOVE" button below it. The sensor details are: Readings: 5, Time Interval: 3s, Space Interval: 6m.

Geolocation bounds ⓘ

Buffer ⓘ

200 km ▾

Time to Complete ⓘ

Hours *

Minutes *

Seconds *

Sensors ⓘ

Accelerometer

Readings: Time Interval: Space Interval:
5 3s 6m

REMOVE

Figure 3.2: Campaign creation form

Once registered, the crowdsourcer can create Sensorworker campaigns. The crowdsourcer must provide enough information to build a new Campaign document in the database. This document will be associated with the crowdsourcer by storing in the campaign's `campaignIDMW` field the id of the Microworkers campaign that has been created during the crowdsourcer's registration. Figure 3.2 shows some relevant fields required during the campaign creation process, in particular

- **Geolocation bounds:** the crowdsourcer can select a polygon to specify the area in which a worker must be to perform the sensing work. The bounds are stored in the `geojson` field of the campaign document and it consists of an array of coordinates, also known as a *Multipoint* geometry object.
- **Buffer:** as previously explained, the buffer indicates how far the worker can be from the `geojson` polygon to be able to *take part* to the campaign, but not *perform* the sensing required. To do the latter, they must be within the polygon. This implementation has been chosen once considered that the location provided by the IP address might be approximate and, as it will be explained in the worker's client implementation when they are in the process of choosing a Sensorworker campaign, the platform does not have access to the worker's GPS position. To alleviate the chances of a false negative, that is to say, a worker who is within the `geojson` bounds but with an IP address reporting a position outside it, the buffer system has been implemented.
- **Sensors:** As shown in Figure 3.2, every sensor listed in the `sensors` field of the Campaign document must have three constraints:
 1. The number of readings for that sensor.
 2. How many seconds must elapse between readings.
 3. How many meters of distance must be between one reading and its previous.

The section about the worker's client implementation will explain how these constraints are enforced.

- **Number of Positions:** another field to note is the number of positions to add for this campaign. These new positions will be added to the Microworkers campaign related to the Sensorworker campaign. Before creating the campaign, the cost of adding these positions will be computed and shown to the crowdsourcer.

Crowdsourcer dashboard

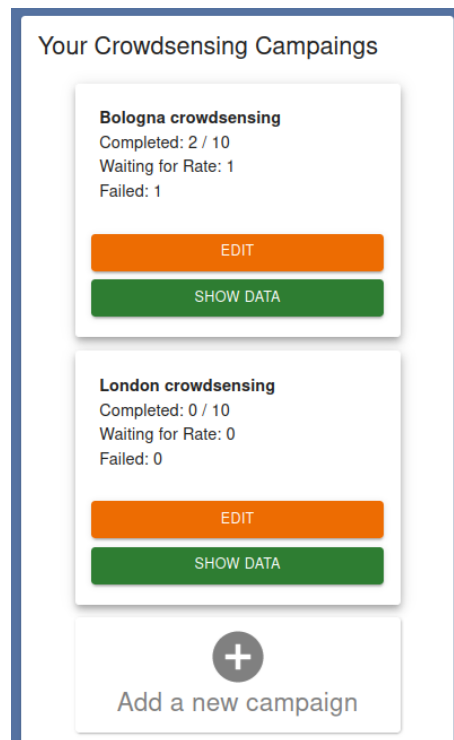


Figure 3.3: Crowdsourcer Dashboard

Once the crowdsourcer has created some campaigns, they can be accessed through the dashboard, which shows at a glance some statistics for each Sensorworker campaign. The only editable data for the campaigns are the title and the description.

It must be noted that the number of failed jobs can exceed the total number of positions available for the campaign since they will not be taken into account when computing how many free positions are still available for each campaign. The calculation process that returns the number of free positions will be explained in the worker's client implementation.

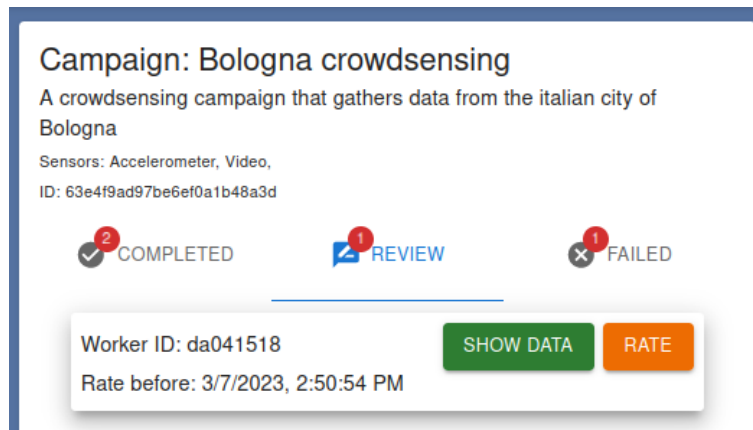


Figure 3.4: Rating panel

Jobs rating

When a job has been completed, it must be then rated by the crowdsourcer. The crowdsourcer can assess the data provided by any job in the **REVIEW** panel of each Sensorworker campaign. Using these data, the crowdsourcer can give two types of ratings: *OK* and *NOK*. If they give an *OK* rating, the job will be moved to the **COMPLETED** panel and the worker will be compensated for their service, On the other hand, if the crowdsourcers gives a *NOK* rating, the job will be moved into the **FAILED** panel and the worker won't be paid, but instead will be notified about the errors on Microworkers. Another way through which a job can end up in the failed tab happens when a worker doesn't complete the job within the amount of time stored in the `timeToComplete` field of every campaign. At the end of the next chapter, we will explain how this delayed check is performed.

3.2 Worker Interactions

So far we have discussed the interaction that a crowdsourcer can have with the Sensorworker platform. However, these interactions are simple when compared to the interactions that the worker has to have with the platform,

especially when considering how they access it.

3.2.1 Accessing Sensorworker

We first need to discuss how the worker can access the Sensorworker platform through Microworkers. As explained in the previous section, whenever a crowdsourcer registers on the platform, a new Microworkers campaign is created on their behalf using their `apiKey`. This new campaign is set with an **External Template**, which is a URL that links to a web page where the worker should perform the tasks for that Microworkers campaign. This URL is shown to the worker through an **iframe**, to which are passed 3 URL parameters

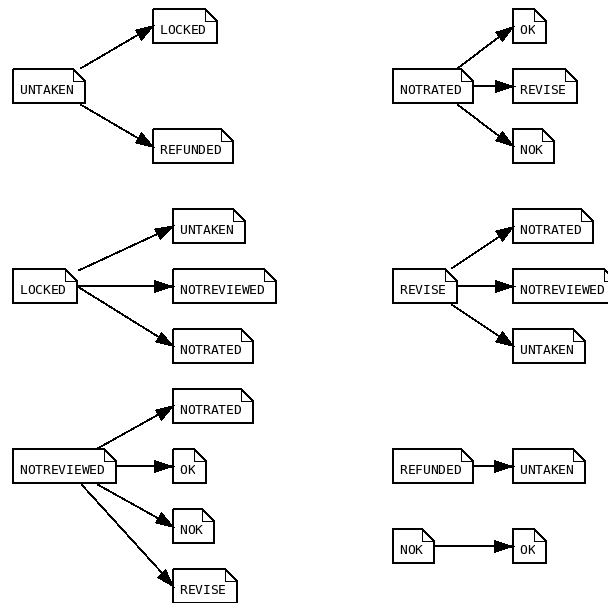


Figure 3.5: Slot status transitions

- **campaignId**: the id of the Microworkers campaign.
- **workerId**: the id assigned to the worker within Microworkers.

- **slotId**: the id of the slot. Slots are similar to positions, but they are used to keep track of how many workers are actively working on a certain campaign. Slots can have different statuses, and can also transition between them. Some are shown in Figure 3.5. In the next section, we will discuss how the slot's statuses are employed within Sensorworker.

The iframe is shown in two situations, each with different values for the parameters

As preview : this iframe is shown as a preview before the worker has actually taken part in the Microworkers campaign. In this case, the workerId and the slotId are set as dummy values.

As non-preview : in this case, the worker has actually taken part in the Microworkers campaign, and the workerId and slotId hold real values.

This distinction will allow Sensorworker to adopt a different behavior in each case.

Taking part in a campaign

When a worker opens a Sensorworker-generated Microworkers campaign, they are presented with something similar to Figure 3.6, where the iframe has been directed to a web page within the Sensorworker platform called **CampaignList**. CampaignList shows a list of Sensorworker campaigns as well as their main details such as title, description, required sensors, and time to complete. Since the worker has not pressed the "Accept and Start" button on the bottom-left, the iframe is shown in preview mode and CampaignList is provided with the dummy values. In this case, since CampaignList is being viewed through an iframe and does not have access to the GPS sensor, it uses the IP address of the worker to compute an approximate location that sends to the back end in order to find which Sensorworker campaigns are available based on that location, as well as how many jobs have already been

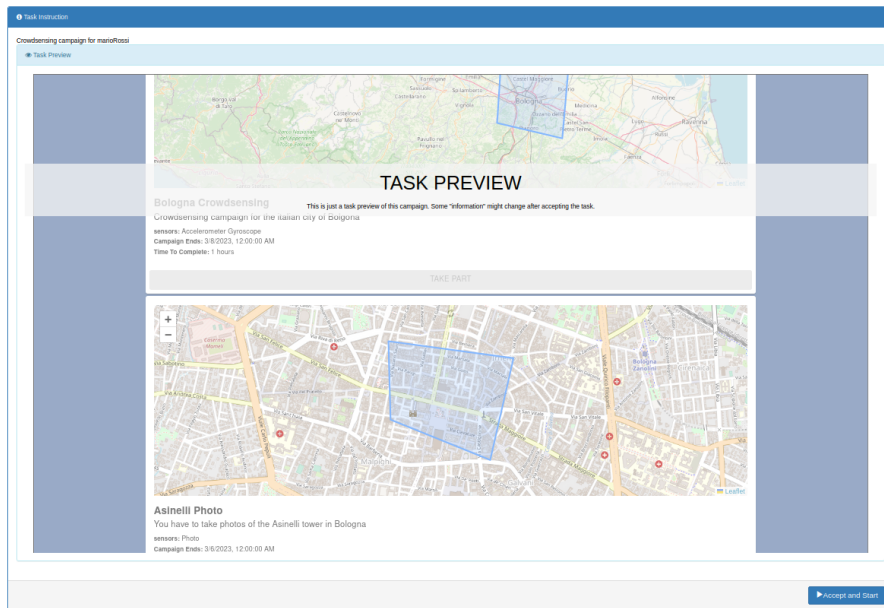


Figure 3.6: Preview iframe

completed. Once it receives a response, CampaignList shows the list of the available campaigns but keeps the "Take Part" button disabled since the worker has not actually started the job. At this moment in the interaction, no slot has been assigned to the worker

When the worker starts the Microworkers campaign, the iframe transitions from preview to non-preview and, as a consequence, CampaignList is provided with the actual workerId and slotId. This also means that a slot has been taken by this worker and the "Take Part" buttons have been enabled for every worker. At this point, the worker has 10 minutes available to choose and take part in one of the Sensorworker campaigns.

Once the worker has chosen, the iframe is redirected to a web page called **SensorCollectQR**, shown in Figure 3.8, which holds a QR code for the web page within the Sensorworker platform that actually has the capabilities to collect and send sensory data from the browser. The choice to present a QR code is due to the fact that in order to access the sensors, the worker must be moved outside an iframe. It is also not possible to assume that the worker

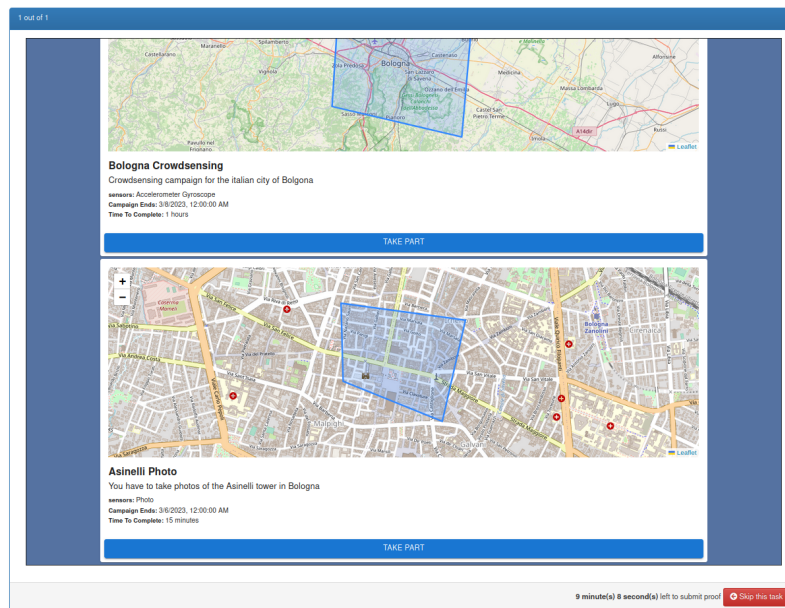


Figure 3.7: Non-preview iframe

is already on a mobile device, and the QR code is a convenient method to transition to a smartphone.

When a worker chooses a Sensorworker campaign, they also create a Job document in the database with the required data and submit the proof of their work to Microworkers, as to say they "conclude" they transition their lock from *Locked* to *NotRated*. This does not mean that their work is actually over, and they still have to follow the QR code and send the required data. This implementation has been done to mitigate the case in which all the available slots for a Microworkers campaign, which represents a group of Sensorworker campaigns, get locked with workers who do not go forward with the selection. This prevents other workers from taking part in the Microworkers campaign.

Giving only 10 minutes to choose from the Sensorworker campaign and submitting the proof once the worker chooses the campaign is a way to decouple the slotting system of Microworkers from jobs that need to be performed for the Sensorworker campaigns.

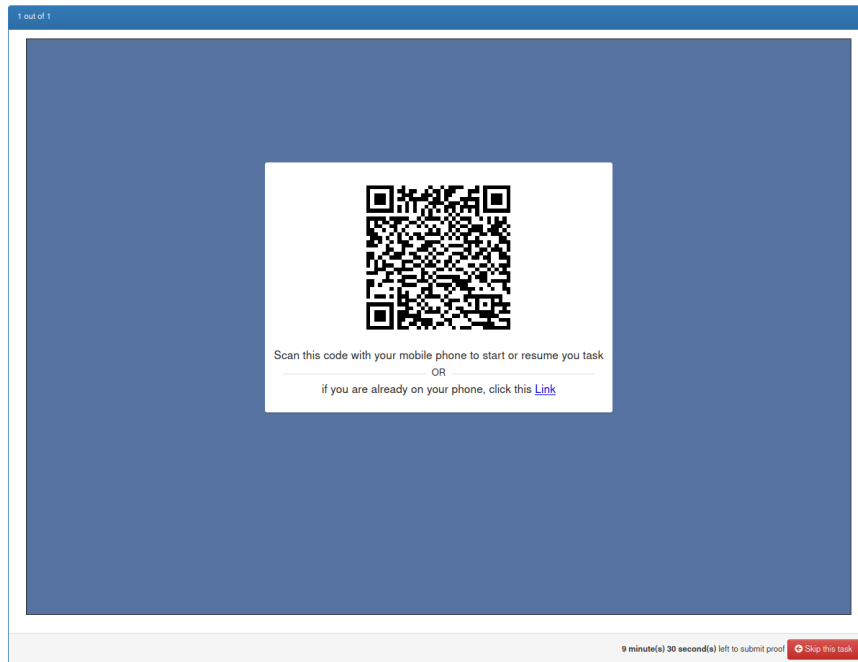


Figure 3.8: SensorCollectQR page

It must be noted that CampaignSelect performs another check if it detects that it is being run in a non-preview iframe. This check uses the `workerId` to verify if there are any active jobs, that are jobs with the `active` field set to `true`, in which case it directly transitions to SensorCollectQR, blocking them from submitting the proof for this new slot that they have taken.

This implementation allows for each worker and each group of Sensor-worker campaigns from the same crowdsourcer, to have only one active job at a time.

Figure 3.9 shows a summary of the worker interaction described so far, starting from when they land on the preview iframe to when they move onto SensorCollectQR.

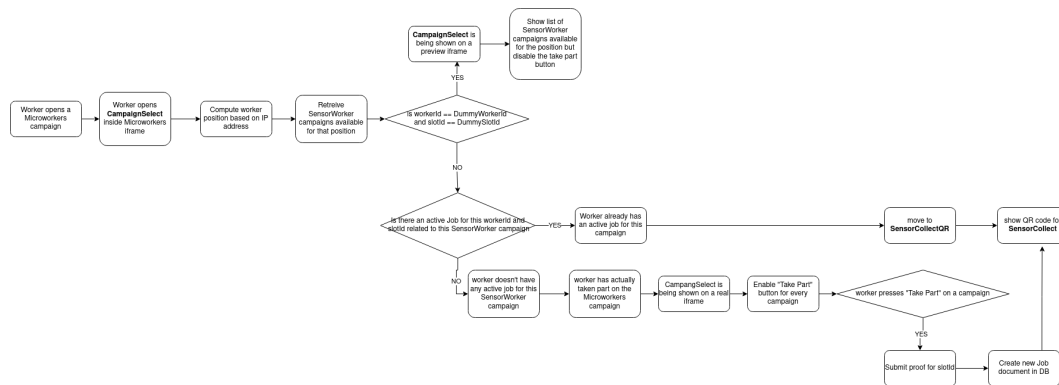


Figure 3.9: Worker interaction flowchart

Collecting Data

The transition from Microworkers to Sensorworker is the most complex and delicate of the interactions. Once the worker has access to the QR, they can transition onto **SensorCollect**, the web page where all the sensor permissions detailed in the previous chapter are checked and from where the worker can send the sensory data.

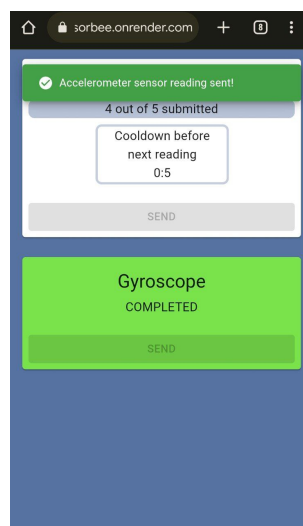


Figure 3.10: SensorCollect page

Figure 3.10 shows how the SensorCollect page could be structured for a Sensorworker campaign that requires only the Gyroscope and the Accelerometer sensors.

Chapter 4

Back-end Implementation

During chapter 3, we have gone through the interaction and the implementation of the client, paying particular attention to the transition of the worker from Microworkers to Sensorworker. In this chapter, we will focus on the back-end implementation that manages the data and responds to the various API calls.

Following the MVC architecture detailed in chapter 2, we will now explore the implementation of three different components of the back-end architecture: **Controllers**, **Routes**, and **services**

4.1 Controllers

Base controller

Since controllers are required to respond using HTTP responses, all the controllers *extend* a specific `base.controller`. This controller is equipped with a series of functions to provide different HTTP responses.

```
export default class BaseController {
  ...
  public static ok<T>(res: express.Response, data?: T) {
    if (!!data) {
      res.type("application/json");
      return res.status(200).json(data);
    } else {
      return res.sendStatus(200);
    }
  }
  ...
}
```

```
public static badRequest(res: express.Response, message?: string) {
  return BaseController.jsonResponse(
    res,
    400,
    message ? message : "Bad request"
  );
}
...
}
```

Listing 4.1: Base controller

For example, listing 4.1 shows that the `ok` function is responsible for returning the HTTP response 200 together with optional data, and the `badRequest` function, when called by the other controllers, responds to the request with the code 400 and an error string, which defaults to "Bad Request".

Campaign controller

This controller extends the base controllers and is concerned with everything regarding Sensorworker Campaigns. In particular, it implements the `getFilteredCampaigns` function shown in listing 4.2. This function returns the filtered campaigns based on the current date and the position provided through the request parameter. Specifically, it searches for all the campaigns whose `begin` date field is in the past and the `end` date field is in the future. Once returned, it further filters the campaigns, keeping only those that contain the worker's position also considering the amount of buffer the campaign provides. In order to do this, the original polygon selected by the crowdsourcer is "enlarged" using the `turf` library, and the position of the worker is compared against it. It must be noted that the function ends by calling a method in the base controller.

```
static async getFilteredCampaigns(
  req: Request,
  res: Response,
  next: NextFunction
) {
  const campaigns = await Campaign.find({
    campaignIDMW: req.query.campaignIDMW,
    begin: { $lt: new Date() },
    end: { $gt: new Date() },
  })
  .populate("sensors completedJobs activeJobs readyToRateJobs failedJobs")
  .exec();

  const point = turf.point([Number(req.query.lat), Number(req.query.lng)]);
```

```

const filteredCampaigns = campaigns.filter((campaign) => {
  if (
    campaign.totalJobs ==
    campaign.completedJobs + campaign.activeJobs + campaign.readyToRateJobs
  ) {
    return false;
  }

  const points = [
    [...campaign.geojson.coordinates, campaign.geojson.coordinates[0]],
  ];
  const polygon = turf.buffer(turf.polygon(points), campaign.buffer);
  return turf.booleanPointInPolygon(point, polygon);
});
super.ok(res, filteredCampaigns);
}

```

Listing 4.2: getFilteredCampaigns function

Crowdsourcer controller

This controller handles all the API calls from the crowdsourcer client, including the creation of Sensorworkers campaigns, fetching the campaign's data, rating the jobs, and creating the session when the crowdsourcer logs in. It must be noted that the route that uses this controller is protected, which means that to access them the crowdsourcer must be logged in to the platform.

```

static async getCampaigns(req: Request, res: Response, next: NextFunction) {
  try {
    const crowdsourcer = await CrowdSourcer.findOne({
      apiKey: req.session.apiKey,
    })
      .select("campaignIDMW")
      .exec();

    const campaigns = await Campaign.find({
      campaignIDMW: crowdsourcer?.campaignIDMW,
    })
      .populate("sensors completedJobs activeJobs readyToRateJobs failedJobs")
      .lean()
      .exec();
    super.ok(res, campaigns);
  } catch (err: any) {
    super.fail(res, err);
  }
}

```

Listing 4.3: getCampaigns function

Listing 4.3 shows how the Sensorworker campaigns are fetched from the database, and the function ends with the usual calls to the base controller's methods both in case of success and in case of failure.

Job controller

As the name suggests, this controller is concerned with API calls related to the Jobs. These methods are mainly called by the worker's client.

```
static async getActiveJobs(req: Request, res: Response, next: NextFunction) {
  try {
    const campaigns = await Campaign.find({
      campaignIDMW: req.query.campaignIDMW,
    }).select({ id: 1 });

    const jobs = await Job.find({
      campaignID: { $in: campaigns },
      workerIDMW: req.query.workerId,
      active: true,
    });
    if (jobs.length > 1) {
      throw new Error("multiple active for same MW campaign");
    }
    super.ok(res, jobs[0]);
  } catch (err: any) {
    super.fail(res, err);
  }
}
```

Listing 4.4: getActiveJobs function

Listing 4.4 shows one of the methods in this controller, which is used by the **CampaignList** page to perform the check detailed in chapter 3. This function searches for all the Sensorworker campaigns that are associated with the Microworkers campaign provided with the `req` parameter. These campaigns are then used as the filter in order to search for any active jobs. If one is found, it is sent to the client in order to enforce its completion to the worker who started it.

Sensor Data controller

This controller is used only by the worker's client. It stores the method that handles the API calls from the **SensorCollect** web page, which sends and fetches the sensory data of a worker.

```
static async postReading(req: Request, res: Response, next: NextFunction) {
  if (req.headers["content-type"]?.includes("multipart/form-data")) {
    next();
  } else {
    const doc = await SensorData.findOneAndUpdate(
      {
        name: req.body.name,
        jobId: req.body.jobId,
      },
      {
        $push: { data: req.body.state },
      }
    );
  }
}
```

```
    },  
    { upsert: true, new: true }  
  ).exec();  
  super.ok(res, doc);  
}  
}
```

Listing 4.5: postReading function

Listing 4.5 shows an example of a function in this controller. `postReading` is called whenever the worker presses "SEND" on a sensor, where the first `if` statement checks whether the sent data is a `multipart` stream of data, which represents a photo, video, or audio file. In this case, the method calls the "next" function in the router, which is a function with the sole objective of storing this special kind of data in the database. We will discuss how this is done in the section about services. On the other hand, if the sent data is from one of the other sensors, it is pushed inside the `data` field of the `SensorData` document of that sensor for this job.

Session controller

This controller has the only task of guarding the Crowdsourcer's route and has only one method

```
public static async guard(req: Request, res: Response, next: NextFunction) {  
  if (!req.session.apiKey) {  
    super.unauthorized(res);  
  } else {  
    next();  
  }  
}
```

Listing 4.6: guard function

This function is responsible for checking if the calling client has an active session, that is to say, the crowdsourcer is logged in. If this check fails, the controller redirects the client to the login page by signaling that they are not authorized.

4.2 Routers

Every controller has an associated router. The routers are responsible for routing the execution toward the correct Controller's method.

```
router.get("/getInfo", CrowdSourcerController.getInfo);
router.post("/createCampaign", CrowdSourcerController.createCampaign);
router.get("/getCampaigns", CrowdSourcerController.getCampaigns);
router.post("/editCampaign", CrowdSourcerController.editCampaign);
router.get("/getJobs", CrowdSourcerController.getJobs);
router.get("/getSensorData", CrowdSourcerController.getSensorData);
```

Listing 4.7: Router for the crowdsourcer controller

Listing 4.7 shows some of the routing paths that direct to the crowdsourcer's controller. It must be noted that the guard on the crowdsourcer routes is set with the following function call

```
app.use("/api/crowdsourcer", SessionController.guard,
        crowdsourcerRouter);
```

Listing 4.8: Guard on the Crowdsourcer router

This means that only when the `SessionController.guard` method shown in figure 4.6 invokes the `next()` function, the execution can progress into the crowdsourcer's router.

4.3 Services

The back-end architecture uses three different services in order to implement some critical functionalities

GriffS Service : Since the database used to store the readings of the sensors is a document-based NoSQL server, it isn't fit to save files with arbitrary dimensions like photos, video, and audio recordings. In order to do so using MongoDB, the GridFS service has been employed. This

service takes a stream of data sent from the client, selects a "Bucket" into which it stores chunks of the streamed data, and finally creates a special document that points to the first chunk of the data required to restore the initial file. Specifically, the GridFS service uses three buckets: photos, videos, and audio recordings. When a file is requested from the client, it is sent as a stream.

Microworkers API service : During the interactions with Sensorworker, there are many times that require some Microworkers API calls. For example, when a crowdsourcer registers into the platform, a Microworkers campaign must be created on their behalf. Only the backend is allowed to directly interact with the Microworkers APIs, and in order to ease this interaction, the Microworkers API Service has been employed. This service has been created through a process of code generation based on the swagger file of the Microworkers APIs, and it essentially turns the API calls into function calls, providing an easy-to-use class that takes the required parameters in order to invoke the API.

AgendaJS service : As anticipated in chapter 3, if a worker doesn't complete a job within the amount of time detailed in the `timeToComplete` field in the Sensorworker campaign document, the job is marked as failed and the slot is rated as NOK. This check must be performed when the `timeToComplete` expires, and in order to do so the AgendaJS library has been used. This library allows the creation of something similar to *CRONJOBS*, that is to say, a function that is run on a specific moment in the future. Using this service, a function that checks the completion of the job is set to run at the `timeToComplete` every time a job is created. Thanks to this, the job can be automatically rated as failed when the function runs and verifies that the job has not been completely fulfilled.

Chapter 5

Testing

In order to test the Sensorworker platform, three different campaigns with varying degrees of requirements have been administered to the same group of 10 testers, all based in the city of Bologna with different traveling routes around the city. All testers have been given a general idea of the functionalities of the Sensorworker platform and have been guided through creating an account on Microworkers in order to take part in the tests. All the testing campaigns require 5 readings from every required sensor, and they have been structured as follows

- **Campaign 1:** The geographical bounds of this campaign are limited to the city center, with the least amount of buffer (100km). Only the Basic sensors have been required (accelerometer, gyroscope, etc) and there are no temporal or spatial requirements imposed on them
- **Campaign 2:** This campaign is geographically bound to a specific part of the suburbs, which is around 15 km away from the city center. The buffer has been extended to 300 km and all the sensors (both Basic and Audiovideo) are required to perform this task, asking the testers to take photos, videos, and audio recordings of specific things detailed in the description of the campaign (photos of street signs, videos of trees, an audio recording of a car passing by). No temporal or spatial constraints have been put in place.

- **Campaign 3:** For this campaign, the geographical bounds are once again put within the city center, which has been reported as the most frequented place by the testers. The buffer has been set to 500 km and all the sensors are required, once again asking for the same specific audio and video readings for Campaign 2. For every sensor, a temporal constraint between 10 to 30 seconds and a spacial constraint between 1 to 5 meters have been set.

5.1 Results

First, before considering the feedback provided by the testers, we must look into the tester group demographic.

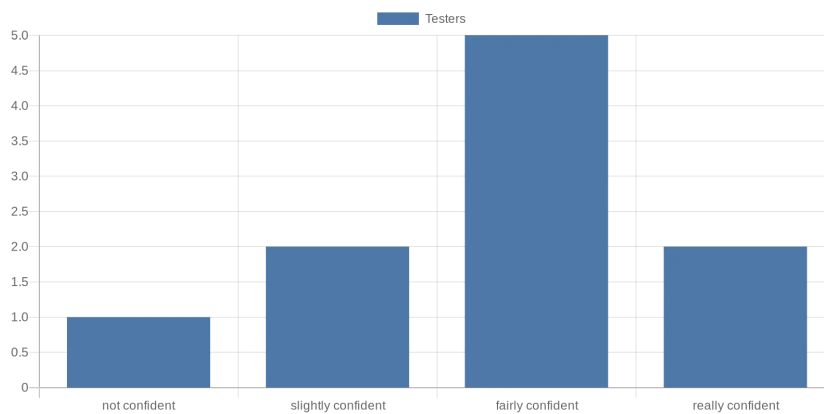


Figure 5.1: Testers confidence level on technological know-how

As we can see in Figure 5.1, most consider themselves to be fairly confident, with 7 out of 10 testers placing in the upper categories. Seeing these first results, it seems reasonable to expect a low failure rate.

The age of the tester group ranges from 16 to 65, with most of them concentrated between 20 to 30 years of age.

Only 3 out of 10 testers are in possession of a personal vehicle (excluding bicycles), so most of them are limited in their transportation method, especially when it comes to working on Campaign 2.

All testers reported frequently visiting the two locations required in the campaigns.

Finally, for the purpose of the test, a job is considered successful as long as all the readings of all the required sensors are fulfilled, without considering the quality of the sensory data.

Campaign 1

This first campaign is structured to be the easiest and most approachable for inexperienced workers. The success rate has been 80%, as most of the testers have successfully sent the data while being within the geographical confines. One of the testers whose task failed reported that it wasn't clear that their job was not completed yet. The other tester reported difficulties with providing access to the sensors on their smartphone.

In regard to the data, it must be noted that each successful tester sent the readings from a single location, in the span of a few seconds. This suggests that having no temporal or spatial constraints on the sensors can have poor data significance as consequence.

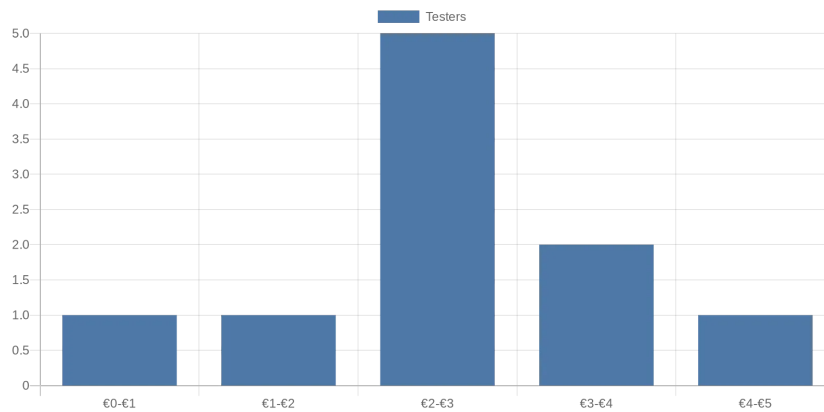


Figure 5.2: Expected payment for Campaign 1

Figure 5.2 shows how much compensation the testers would expect after performing Campaign 1, with an average range between 2.1 € and 3.1 €.

Campaign 2

Campaign 2 has been designed to test how the difference in the location might impact the performance of the campaign, as well as the added difficulties in the sensing task due to the audio and video sensors being now required.

The success rate for Campaign 2 has been 60%. Of the 4 failed jobs, 3 testers reported difficulties with interacting with the wide amount of sensors. Especially the audio Sensor has been singled out as a cause of confusion since the testers were not able to clearly hear the recordings due to poor quality and did not send the readings as they considered them inadequate.

When considering the data, the basic sensors reading suffer from the same condition as in Campaign 1, that is to say, each worker sent the reading from the same location. It must be noted that the photos and the videos sent mostly follow the instructions provided in the description of the campaign, with only one tester seemingly not noticing the instructions.

When it comes to compensation, many testers have asked more due to the added complexity of the audio and video sensors, with an average range between 2.3€ and 3.3€.

Campaign 3

This final campaign has been designed to study the impact of temporal and spatial constraints on the sensors.

The success rate for this campaign has been 40%, with many of the testers who failed their job reporting that the spacial constraints have been the most impactful factor for their missing readings. When queried about the situation in which they performed the job, they reported that they were stationary and in a single location, and were not willing to move in order to meet the spacial constraints required after the first reading.

In regards to the compensations, the testers reported that they would perform this kind of job for higher pay compared to the previous two campaigns, with an average between 3.2€ and 4.2€.

Conclusions

From these tests, we can infer some tendencies that start to emerge. More specifically, the most costly requirement is the temporal and spatial constraints, which if not adequately incentivized, leads to a high failure rate.

During the development of Sensorworker, different technologies, and methodologies have been employed in interesting and innovative ways, especially in order to integrate with the Microworkers platform.

The crowdsensing system offered by Sensorworker still suffers from notable and open issues which afflict many MSC platforms, some of which are:

- Problems concerning anonymization and the privacy of the user data. Despite the abstraction offered by Microworkers, Sensorworker still depends on the data sent by the browsers of the workers' mobile devices, which can potentially be an attack surface for malicious actors.
- Sensor accessibility is highly dependent on third-party browser developers (e.g. Google, Firefox), which can limit or expand the potential sensing capabilities of the platform. In this case, running on a browser is a trade-off between ease of use and platform features

These concerns aside, the MSC platform itself can be improved in regard to User experience and can greatly benefit from a robust and efficient deployment system through the use of multiple container instances.

The tests performed in the final chapter highlight how the most costly requirement for crowdsensing campaigns are the spatial and temporal constraints which, if poorly compensated, lead to a high failure rate. The geo-

graphical conditions are met as long as it intercepts the usual routes of the potential worker.

Everything considered, Sensorworker offers a convenient and user-friendly platform to perform and gather crowdsensing data leveraging an existing and reliable user base, which addresses one of the primary concerns in relation to MSC systems and more general crowdsourcing requirements.

Bibliography

- [1] Hossein Falaki, Dimitrios Lymberopoulos, Ratul Mahajan, Srikanth Kandula, and Deborah Estrin. A first look at traffic on smartphones. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 281–287, 2010.
- [2] Raghu K Ganti, Fan Ye, and Hui Lei. Mobile crowdsensing: current state and future challenges. *IEEE communications Magazine*, 49(11):32–39, 2011.
- [3] Bin Guo, Zhiwen Yu, Xingshe Zhou, and Daqing Zhang. From participatory sensing to mobile crowd sensing. In *2014 IEEE International Conference on Pervasive Computing and Communication Workshops (PERCOM WORKSHOPS)*, pages 593–598. IEEE, 2014.
- [4] Larissa Hammon and Hajo Hippner. Crowdsourcing. *Business & Information systems engineering*, 4:163–166, 2012.
- [5] Matthias Hirth, Tobias Hofffeld, and Phuoc Tran-Gia. Anatomy of a crowdsourcing platform-using the example of microworkers. com. In *2011 Fifth international conference on innovative mobile and internet services in ubiquitous computing*, pages 322–329. IEEE, 2011.
- [6] Jeff Howe et al. The rise of crowdsourcing. *Wired magazine*, 14(6):1–4, 2006.
- [7] Robin Kraft, Winfried Schlee, Michael Stach, Manfred Reichert, Berthold Langguth, Harald Baumeister, Thomas Probst, Ronny Han-

- nemann, and Rüdiger Pryss. Combining mobile crowdsensing and ecological momentary assessments in the healthcare domain. *Frontiers in Neuroscience*, 14, 2020.
- [8] Yutong Liu, Linghe Kong, and Guihai Chen. Data-oriented mobile crowdsensing: A comprehensive survey. *IEEE Communications Surveys & Tutorials*, 21(3):2849–2885, 2019.
- [9] Huadong Ma, Dong Zhao, and Peiyan Yuan. Opportunities in mobile crowd sensing. *IEEE Communications Magazine*, 52(8):29–35, 2014.
- [10] Xufei Mao, Xin Miao, Yuan He, Xiang-Yang Li, and Yunhao Liu. Citysee: Urban co 2 monitoring with sensors. In *2012 Proceedings IEEE INFOCOM*, pages 1611–1619. IEEE, 2012.
- [11] Keshab Nath, Sourish Dhar, and Subhash Basishtha. Web 1.0 to web 3.0-evolution of the web and its various challenges. In *2014 International Conference on Reliability Optimization and Information Technology (ICROIT)*, pages 86–89. IEEE, 2014.
- [12] Paul Sloane. *A guide to open innovation and crowdsourcing: Advice from leading experts in the field*. Kogan Page Publishers, 2011.

Acknowledgements

I would like to take this opportunity to express my gratitude to all those who have supported and encouraged me throughout my academic journey. Firstly, I would like to thank my parents for their support, love, and encouragement throughout my academic career. Their constant belief in me has been the driving force behind my success. I would also like to thank Claudia and her family for their endless support and understanding throughout my university career. They have always been there for me, providing me with encouragement and advice. I would like to extend my appreciation to my thesis supervisor, Dott. Federico Montori, for his guidance, expertise, and patience throughout my research. His willingness to share his knowledge and introduce me to new and interesting technologies has been invaluable.

Thank you all for your invaluable contributions to my academic journey.