Universidad Autónoma de Madrid

Departamento de Informática

Computer Systems Project
Assignment 3

Roberto Marabini
Alejandro Bellogín

Changelog

Version[1]	Date	Author	Description
1.0	10.08.2022	RM	First version.
2.0	26.10.2022	RM	Change heroku → render.com.
2.1	5.12.2022	RM	Renumbering assignment 4 -> 3
2.2	22.12.2022	AB	Translation to English
2.3	29.1.2023	RM	Review before upload to *Moodle*
2.4	12.2.2023	RM	Change Elephantsql by neon.tech
2.5	18.4.2023	RM	Change gameUpdateParticipant/<int:publicid> to gameUpdateParticipant/

[1] Version control is made using 2 numbers $X.Y$. Changes in Y denote clarifications, more detailed descriptions of some aspect, or translations. Changes in X denote deeper modifications that either change the provided material or the content of the assignment.

Contents

1 Goal

Many of you may be familiar with the "Kahoot" platform, that allows creating questionnaires with questions and answers (`https://kahoot.com/`). This tool makes learning concepts easier as if it was a contest.

We would like to implement a web application that includes the basic functionality of "kahoot", and allows preparing "on-line" questionnaires. We will mostly be using *Django* and *Vue.js* environments to create the proposed application. In the `https://kahoorclone.onrender.com` and `https://kahootclone-render-vue.onrender.com` URLs you may see the implementation we did of this application as a previous step before writing this assignment. Create a user to explore the application.

1.1 Requirements

Overall, the application we want to create must:

1. Allow creating and identifying users. Users are people who create and manage the questionnaires.

2. Allow users, through a collection of forms, creating questionnaires.

3. Allow users showing and managing the created questionnaires. That is, adding, removing, and modifying questionnaires with their corresponding questions and answers.

4. Allow users to run a questionnaire. That is, to make an instance of the questionnaire public so that some participants could solve it. The participants may connect to the questionnaire instances and answer the questions themselves, but they could not modify the questionnaires. The participants do not require to log into the system to answer the questionnaries.

5. Manage how participants access the public instances of the questionnaires.

6. Be in charge of counting the responses given by each participant and presenting the score of every one of them.

7. Besides, the administrative part, that is, any task carried out by the users, will be implemented in *Django*.

8. The interface of the participants will be done in *Vue.js* connected to a REST API created in *Django*.

The code must

1. Satisfy the style criteria highlighted by the *Flake8* utility.

2. Use versions of the python module listed in the "requirements.txt" file (*Django*) and in the makefile option "requirements" (*Vue.js*) available in *Moodle*.

3. Store the information in a database created with the *PostgreSQL* database manager and stored in `https://neon.tech`.

1.2 Control version system: *git*

You must use *git* as a tool for version control. Among the material to submit in this assignment, you will have to include the *git* repository used; it must contain frequent accesses (*git commit*) of BOTH members of the pair. There must exist at least a weekly access by each member of the pair. It is important the repository is PRIVATE, **any pair who creates a public repository will automatically fail the assignment**.

2 Implementation

In this assignment, we will focus on creating the necessary functions and the interface to create the questionnaires, while the next assignment will be devoted to implement the infrastructure used by the participants.

To start working, you will need to create a *Django* project called *kahootclone*, a base "template" similar to `base.html` (see auxiliary files in the repository `https://github.com/rmarabini/` psi-alumnos) from where all the "html" files we create in the future will inherit. Finally, by using *PostgreSQL*, create a database in `https:`

`//neon.tech`. Modify the `settings.py` files from *Django* to use this database. Remember that the file `settings.py` should not include information that may help potential hackers, as it could be the `https://neon.tech` password. Therefore, read the `https://neon.tech` from enviroment variables as documented in listing 1.

2.1 Makefile

During the development of the assignment, it is common to frequently run some commands. To automate the process, you may use the command `make` together with the `makefile` file that is available in the material of this assignment and that we will use to grade your submissions.

In `makefile` the following operations, among others, are defined:

- `create_super_user`: creates a user with admin priviledges. The user name and password are `alumnodb`. This is equivalent to `python3 ./manage.py createsuperuser`
- `populate`: populates the database, equivalent to `python3 ./manage.py populate`
- `runserver`: equivalent to `python3 manage.py runserver 8001`
- `update_models`: equivalent to `python3 manage.py makemigrations; python3 manage.py migrate`
- `dbshell`: launch the *PostgreSQL* client, `psql` (./manage.py dbshell)
- `shell`: launch python with the *Django* environment already loaded (./manage.py shell)

2.2 Users

Before describing the data model of the project *kahootclone*, let us implement the user management. User management is key in a large number of applications, so we will make a design that can be reused later on easily. To the extent possible, we will use the implementation that comes with the *Django* distribution. We need to

implement the services for `log-in` (start the session), `log-out` (close the session), and `sign-up` (create a new user). We will start creating an application called `models` where the code will be stored.

Let us now describe in detail each requested service:

login For the `login` service, there is no need to create any function but to reuse the available views in `django.contrib.auth.urls` as shown in the first assignment. When accessing the log-in page, a form must be shown where the name of the user (username) and the password (password) will be collected. If the identification is positive, the user will be redirected to the "homepage", otherwise, the "login" page will reappear together with the corresponding error message. Do not use directly the `User` class defined in `django.contrib.auth.models` but create your own `User` model by inheriting from `AbstractUser`.

```
class User(AbstractUser):
    ''' Default user class, just in case we want
    to add something extra in the future'''
    # remove pass command if you add something here
    pass
```

logout For the `logout` service there is no need to create any function but to reuse the views available in `django.contrib.auth.urls` as shown in the 1st assignment. When selecting `logout` from the menu, the user will be redirected to the "homepage" and their session will be closed.

signup might be implemented reusing the `UserCreationForm` form (`from django.contrib.auth.forms import UserCreationForm`) together with a small view, see examples in `https://simpleisbetterthancomplex.com/tutorial/2017/02/18/how-to-create-user-sign-up-view.html`). When accessing the "sign-up" page you must show a form that will ask for the name of the user (`username`) and the password twice (`password1` and `password2`). Once the new user was created, they must be automatically connected.

In every service to be implemented, do not use the "templates" that are offered by default, instead you should personalize them to inherit from `base.html`.

The "homepage" must check if the user is "conectado", by showing a link to the services `login` and `sign-up` in the negative case, and a link to the `logout` service in the positive case.

The variables `LOGIN_REDIRECT_URL` and `LOGOUT_REDIRECT_URL` (settings.py) may help you with the required redirections in `login` and `logout`.

2.2.1 "Testing"

To verify the correct implementation of the different services, a set of tests (not necessarily complete) is provided that your code must satisfy. These tests must be understood as additional requirements of the project.

In particular, in the `models/test_authetication.py` file, 4 tests are provided that verify the correct operation of the `log-in`, `log-out`, and `sign-up` functionalities.

IMPORTANT NOTE: When going through the implementation, a TDD (test-driven development) strategy is recommended, trying to satisfy one by one and following the established order of each test. This applies to the entire implementation of the project for the provided tests.

2.2.2 Git

As usual, you must make sure to save your code periodically in *Github*. Use it to share your code with the other member of the pair. Do not use other options like "Live share" from *Visual Studio* or *Dropbox*.

Some of the most frequently used commands to manage the repository are listed next.

```
git status # list new or modified files
git add filename.py  # add a new file to the git framework
# save current version of file filename1.py in local repository:
git commit -m 'authentication services done' filename1.py
git push  # update remote repository
```

2.3 Data model

Apart from users, our project needs to manage questionnaires with their questions and answers, together with the scores of each participant.

The data model that will support the application will follow the ORM (Object Relational Mapping) scheme from *Django*. We need questionnaires, questions, answers, scores, etc. We show next a relational scheme that contains the minimum design you must use in your implementation. You may add any entity or attribute you consider necessary, but **do not remove** any of the proposed ones:

Questionnaire(**questionnaire_id**, title, created_at, updated_at, user↑)
Question(**question_id**, question, questionnaire↑, created_at, updated_at, answerTime)
# it is assumed each `question` appears in a unique `questionnaire`
# `question.question` is the string of characters with the question to be made
# `question.answerTime` is an integer equal to the time, in seconds, that it is offered to answer to this question
Answer(**answer_id**, answer, question↑, correct)
# `answer.answer` is the string of characters with the answer
# `correct` is a boolean that indicates if this answer is correct or not
# It is assumed there exists only one correct answer for every question
# The form to introduce the answers must take into account this limitation, that is,
# there must be only one correct answer per question
# It is assumed each answer is only related with a unique question.
game(**game_id**, questionnaire↑, created_at, state, publicId, countdownTime, questionNo)
# the model `game` is explained in detail at the end of this section
Participant(**participant_id**, game↑, alias, points, uuidP)
# `alias` is how the participant will be known in the game
# `points` stores the scores of the participant in the game
# Each participant is related with a unique game
# `uuidP` is a string that stores unique identifiers.
# It uses a UUIDField field to store and fill it with uuid.uuid4
# It will be used to identify a participant
Guess(**guess_id**, participant↑, game↑, question↑, answer↑)
# `guess` is each of the responses provided by a participant
# Note that `question` and `game` are redundant as they could be obtained from
# `answer` and `participant`, but they make the database queries easier.

In the scheme, the primary and foreign keys are denoted using bold and the symbol ↑, respectively. All the dates (`created_at` and `updated_at`) must be initialized by default to the instant when the object was created. Besides, `updated_at` must be

updated automatically every time the object is stored.

Every time someone wants to play, an object of type `game` must be created. In this class, the attribute `state` may take the following values

WAITING = 1 Default state when a `game` is created, the system waits for the participants to join the game.

QUESTION = 2 Shows the next available question together with its answers

ANSWER = 3 After some period of time (`question.answerTime`) the correct answer and a ranking with the players' scores will be shown.

LEADERBOARD = 4 When the game ends, the ranking with the final result is shown.

The attribute `game.publicId` is a unique integer number (as the primary key) in the $[1 - 10^6]$ range. This number is shown to the participants and used to identify the game they must join. It must be automatically generated when creating the game by using a generator of random numbers. Showing the users a random number instead of the integer with values $1, 2, 3, \ldots$ increases slightly the security of the application, as it makes it more difficult to access to a `game` to participants that have not been invited.

To make the implementation easier we have minimized the many-to-many relationships, this means, a question will only exist in a unique questionnaire, a participant may only play a unique game, etc. Obviously, it is possible to create two identical question objects, except for the primary key, and assign them to two different questionnaires.

Recall that:

- *Django* creates automatically a primary key called `id` for every model, so you should not create it explicitly.
- a function `__str__` must be created for every model.
- all the created models must be accessibles using *Django* admin interface available in `http://localhost:8001/admin/`.

- all the dates must store both the date and the hour in the created or modified object using an attribute whose type is `DateTimeField`.
- the easiest way to fill the `game.publicId` and `participant.points` attributes is to redefine the function `save` from the `Game` and `Guess` models, respectively.
- once the participant has submitted an answer (`guess`) it is not possible to modify it.

2.4 Testing and coverage

To verify the correct implementation of the different models, the `models/test_models.py` file is provided with a set of tests (not necessarily complete) that your code must satisfy. These tests must be understood as additional requirements to the project.

As you know, *testing* requires the creation of an auxiliary database. Therefore they are only possible when using a local database.

Listing 1: Access to TESTING variable in `settings.py`

```
# settings.py
# SECURITY WARNING: keep the secret key used in production secret!
SECRET_KEY = os.environ.get('SECRET_KEY', default='your secret key')

# SECURITY WARNING: don't run with debug turned on in production!
# by default debug is set to true locally and to false in render
if 'DEBUG' in os.environ:
    DEBUG = os.environ.get('DEBUG').lower() in ['true', 't', '1']
else:
    DEBUG = 'RENDER' not in os.environ

# To run the tests: export TESTING=1, or to use the app: unset TESTING
# To see the current value just type echo $TESTING
DATABASES = {}
POSTGRESQL_URL = 'postgres://alumnodb:alumnodb@localhost/psi'
# please do not include sensitive information as the neon
```

```
# password in settings.py, just read it from the enviroment

if 'TESTING' in os.environ:
    # do not check variable DATABASE_URL
    # just use local postgres
    db_from_env = {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'psi',
        'USER': 'alumnodb',
        'PASSWORD': 'alumnodb',
        'HOST': 'localhost',
        'PORT': '',
    }

else:
    # dj_database_url will check for the
    # variable DATABASE_URL.
    # It should point to NEON but during
    # development it may be interestng
    # to have access to a local postgres database
    # so if DATABASE_URL is not defined use POSTGRESQL_URL
    db_from_env =
        dj_database_url.config(
            default=POSTGRESQL_URL, conn_max_age=500)

DATABASES['default'] = db_from_env

# add render host to allowed host
ALLOWED_HOSTS = ['localhost']
RENDER_EXTERNAL_HOSTNAME = os.environ.get('RENDER_EXTERNAL_HOSTNAME')
if RENDER_EXTERNAL_HOSTNAME:
    ALLOWED_HOSTS.append(RENDER_EXTERNAL_HOSTNAME)
```

Once you have managed to run successfully all the tests, execute the `coverage` command

```
coverage erase
coverage run --omit="*/test*" --source=models ./manage.py test \
     models.tests_models
coverage report -m -i
```

see the coverage of `models/models.py` file and if it is not 100%, add the required tests to reach that value. (The requirement of a 100% coverage applies exclusively to the code you create. Ignore the code created by *Django* or the one provided by your teachers.)

2.5 Testing the models

To be able to verify the correct operation of the web application, we need to store test data in such application.

Populating the database Create a "script" called `populate.py` that generates objects of the different models and persist them in the project database. Use as guide the file called `populate.py` that is provided together with the documentation of this assignment (see the repository). This "script" has a structure that allows to be invoked through the command line `python3 ./manage.py populate` (or `make populate`).

The file must be located in the folder `management/commands` (the "path" is given from the folder that contains the *models* application). If the folder does not exist, create it.

The use of the module Faker is suggested to generate data (see `https://zetcode.com/python/faker/`).

2.6 Services: creating questionnaires

Let us now continue with the implementation of the web application by creating the services that will allow to create/delete/modify questionnaires. Use *class based views*,

that is, model our views as classes that inherit from classes with type `CreateView`, `UpdateView`, etc. Start creating an application called `services` and implement there the new views.

We now list the views to be implemented. For each view, we first show its alias (`name` in the `urls.py` file), the URL it will be connected to, the test that can be used to verify the view and, finally, the relevant part of the returned `context` dictionary together with a brief description of the expected behavior and the "template" to be used. In the URL we have removed the common part to all the views, that is, `/services`. Obviously, each view must call a "template" and show the result to the user, in this assignment we will not assess the aesthetic of these "templates" but: (1) you must implement at least the minimum necessary to test the views and (2) all the "templates" must inherit from a base "template". We recommend you to use some "CSS framework" as "Bootstrap" or "Bulma". Bulma is simpler but Boostrap is more complete.

home | ″ (empty string) | `test01_home` | If the user is connected, it returns a listing with THEIR last five questionnaires sorted by the attribute `updated_at`, otherwise it returns nothing. If the user is not connected, in the "template" the "login" and "signup" options will be shown (Fig. 1). If the user is connected, besides the listing with the 5 returned questionnaires, the following options will be offered: "log-out", create a new questionnaire, and show all the user questionnaires (Fig. 2). In the same way, it should be possible to select one of the questionnaires and access a page where they are shown in detail.

questionnaire-detail | `questionnaire/<int:pk>` | `test02_questionnaireDetail` | It returns the questionnaire whose primary key is pk if the questionnaire was created by the connected user, otherwise it returns an error message, finally if there is no connected user, the login page will be shown. The "template" must show the title of the questionnaire together with every question that takes part of it (Fig. 3). From the returned page by this view, it must be possible to remove/edit/create `questions`, and running a questionnaire (create a `game`).

questionnaire-list | `questionnairelist/` | `test04_questionnaireList` | It returns a listing with all the questionnaires that belong to the connected user. If no user is connected,

the login page will be shown. The "template" must show the list of returned questionnaires (Fig. 4) and offer the possibility to remove any questionnaire.

questionnaire-remove | `questionnaireremove/<int:pk>` | `test03_questionnaireRemove` | It removes the questionnaire whose id is pk if it belongs to the connected user, if the questionnaire was not created by the connected user it returns an error message, finally, if there is no connected user, the login page will be shown. Confirmation must be requested before the removal. After removing the questionnaire the page `questionnaire-list` will be shown.

questionnaire-update | `questionnaireupdate/<int:pk>` | `test05_questionnaireUpdate` | It modifies the questionnaire whose id is pk if it belongs to the connected user, if the questionnaire was not created by the connected user it returns an error message, finally if there is no connected user, it will show the login page. This view modifies the attributes of `questionnarie` but not its foreign keys (such as `question`) that will be modified using another view. Fig. 5 shows an example of form used to create a questionnaire. After modifying the questionnaire, the page `questionnaire-detail` will be shown, which contains the newly modified questionnaire.

questionnaire-create | `questionnairecreate/` | `test06_questionnaireCreate` | It creates and returns a new questionnaire belonging to the connected user. If there is no connected user, the login page will be shown. The "template" must show the created questionnaire and offer the possibility to add/remove/modify questions. Fig. 5 shows an example of form used to create a questionnaire. After creating the questionnaire, the page `questionnaire-detail` will be shown, which contains the newly created questionnaire.

question-detail | `question/<int:pk>` | `test12_questionDetail` | It returns the question whose primary key is pk if the question was created by the connected user, if the question was not created by the connected user then it returns an error message, if there is no connected user the login page will be shown. The "template" must show the question and its answers. Similarly, the "template" must offer the

possibility to delete/edit/add answers to the question (Fig. 7). If the number of answers is four, the option to add an answer must not be offered.

question-remove | `questionremove/<int:pk>` | `test13_questionRemove` | It removes the question whose id is pk if it was created by the connected user, if the connected user did not create the question then an error message is returned, if there is no connected user the login page is returned. Confirmation must be requested before removal. After removing the question, the page `questionnaire-detail` with id=pk must be shown.

question-update | `questionupdate/<int:pk>` | `test15_questionUpdate` | It modifies the question with id=pk if it belongs to the connected user, if the question does not belong to the connected user then an error message is returned, if there is no connected user, the login page is shown. The template must show the question and its answers after the modification.

This view modifies the attributes of `question` but not its foreign keys (such as `answer`) that will be modified using another view. Fig. 6 shows an example of form that could be used to modify a question. After the question is modified, the page `question-detail` with id=pk will be shown.

question-create | `questioncreate<int:questionnaireid` | `test16_questionCreate` | It creates and returns a question linked to the questionnaire with id=questionnaireid. If the questionnaire was not created by the connected user, then an error message is returned, if there is no connected user, the login page is returned. Fig. 6 shows an example of a form that could be used to create a question. After creating the question, the page `question-detail` including the newly created question will be shown.

answer-create | `answercreate/<int:questionid>` | `test26_answerCreate` | It creates and returns an answer linked to a question with id=questionid. If the question was not created by the connected user an error message is returned, if there is no connected user, the login page is returned. Fig. 8 shows an example of a form used to create an answer. After creating the answer, the page `question-detail` with `id=answer.question.id` will be shown.

answer-remove | `answerremove/<int:pk>` | `test23_answerRemove` | It removes the answer with id=pk if it was created by the connected user. If the answer was not created by the connected user, an error message is returned, if there is no connected user the login page is returned. After removing the answer, the page `question-detail` with `id=answer.question.id` will be shown.

answer-update | `answerupdate/<int:pk>` | `test25_answerUpdate` | It modifies the answer with `id=pk` if it belongs to the connected user. If the answer was not created by the connected user, an error message will be returned, if there is no connected user the login page is returned. Fig. 8 shows an example of a form that can be used to modify an answer. After modifying the answer, the page `question-detail` with `id=answer.question.id` will be shown.

game-create | `gamecreate/<int:questionnaireid>` | `test36_gameCreate` | It creates and returns a game linked to the questionnaire with `id=questionnaireid`. If the questionnaire was not created by the connected user, an error message is returned, if there is no connected user, the login page is returned. It is not necessary to implement this now, but by using Ajax, the "template" should periodically connect to the server (view `game-update-participant`) and show the list of participants while they are being incorporated. The view must store a session variable with the game identifier (`game.id`) which will be used in the future.

Some advices on how to implement the views

As it was mentioned at the beginning of the section, the requested views must be based on classes. To implement the requested requirements you may need to overwrite some of the default views for these classes, such as: `get_queryset`, `get_object`, `form_valid`, etc. `get_queryset` could be used to restrict the objects a view may have access to those produced by the validated user, `get_object` allows to manipulate the object (e.g., `questionnaire`) before being passed to the "template", `form_valid` allows to examine and modify the content of the form that will be passed to create an object before creating it.

3 Summary of the work to be done during the first half of the assignment

Create a *Django* project called *kahootclone* that includes the applications *models* and *services*, satisfying the following requirements:

- The project will include an admin page (*Django* interface in path `http://hostname:8001/admin/`) that allows to introduce and remove data. Both the user name and password for admin user must be *alumnodb*.
- Data must be persisted in a *PostgreSQL* database stored in `https://neon.tech`.
- The *models* application will include the data model described in subsection 2.3 included in script `populate.py`.
- The created code must satisfy the tests defined in `models.tests_authentication` and `models.test_models`.
- The "coverage" of files `models.py` must be 100%. The request of 100% coverage applies exclusively to the code you create. Ignore the code created by *Django* or the one provided by the teachers.
- The *services* application will include at least all the services related to the questionnaire (those whose alias starts with *questionnarie*) and the one needed to create the starting page (alias *home*) (see section 2.6).
- The created code must satisfy at least the first six tests defined in `services.test_services.py`.
- All the Python code you write must satisfy the style requirement highlighted by the *flake8* utility. This requirement is not extensible to the code generated automatically by *Django* or the code provided by your teachers.
- The views (`views.py`) must be implemented using classes.

4 Services: using the questionnaires

In the previous section, we implemented the necessary infrastructure to create questionnaires. We now are going to create the part of the services in charge of using the questionnaires by the participants. While for the questionnaire creation only one browser is needed, to play it is needed to keep two browsers open. The first one, common to all participants, shows the questions and scores, whereas the second one is used to provide the answers. Let us describe now the views needed to show the questions and scores. We basically need a view for each of the following actions:

1. create the game (this view was created in the previous part) and wait until the participants join.
2. show a page that warns the participants that the game will start shortly.
3. show in an ordered and consecutive way the questions of the selected questionnaire.
4. show, after every question, the score.
5. when the questionnaire ends, show the ranking with the best scores.

The last four steps can be combined and implemented in a single view.

To start a game, we suggest you add a button to the web page that shows the details of the questionnaire. When selecting this button a game will be created (`game`) with `game.state=WAITING` and a page will be shown that includes the identifier (`publicId`) of the game, that will be used by the participants to join (Fig. 9). This page will be refreshed periodically, updating the lists of the participant aliases who have connected to the game (Fig. 10). Together with the participant aliases, the page will include a button that allows to start the game, showing first a page that will warn the participants that the game is going to start (Fig. 11) and, after that, will show the questions (and the answers) in a consecutive way (Fig. 12). After showing a question, the application will wait `question.answerTime` seconds and, automatically, the correct answer and the score will be shown (Fig. 13). The screen that shows the score will have a button that allows to go to the next question or to

show the ranking/leaderboard with the score of the participants, in case there are no more questions (Fig. 14). Obviously, it should not be possible to answer a question once the correct answer has been shown.

List of views to be implemented

game-create | `gamecreate/<int:questionnaireid>` | | This view was already described in the previous section, and it must be expanded in such a way that every 2 seconds it calls to `game-updateparticipant` and updates the list of participants in the game. In `https://stackoverflow.com/questions/32702758/using-ajax-in-django-to-display-time-of-day-every-second` you may see an example of how to use Ajax to call periodically a URL.

game-updateparticipant | `gameUpdateParticipant/>` | `test01_gameUpdateParticipant` | It returns a list of participants that have joined the game, and that will be used to update the participant list (Fig. 10). This view does not reload the entire page, but only the area devoted to show the list of participants. The game identifier will be obtained through a session variable.

game-count-down | `gamecountdown/` | `test02_gameCountdown` | This view manages the rest of the requirements. It checks the value of `game.state`, updates it correspondingly, creates the necessary variables and returns in each case a different "template". In this way, (a) if the initial state of `game.state` is `WAITING`, it first shows a countdown (Fig. 11) followed by a question (Fig. 12) and updates the state to `ANSWER`, (b) if the state is `QUESTION` it shows a question (with its answers) (Fig. 12) and updates the state to `ANSWER`, (c) if the state is `ANSWER` it shows the score (Fig. 13), updates the variable `game.questionNo` and updates the state to `LEADERBOARD` if it is the last question or to `QUESTION`. (d) If the state is `LEADERBOARD` it shows the ranking (Fig. 14). To implement this view it will be useful to redefine the methods `get_context_data` and `get_template_names` in `gamecountdown`. IMPORTANT: the test `test02_gameCountdown` assumes a session variable exist where the state of the game is stored, and it checks how it is being modified. This test is heavily dependent on the implementation, so you may modify it and create something functionally equivalent adapted to your implementation.

From the point of view of the participants, the application improves a lot if sound is added to the pages used to play. A simple way to do it is by adding in the templates code similar to:

```
<audio controls loop autoplay hidden>
    <source src="{% static 'audio/lobby.mp3' %}"
        type="audio/mpeg">
Your browser does not support the audio element.
</audio>
```

By default, browsers have the sound play disabled, unless the user accepts it explicitly. In `https://support.mozilla.org/en-US/kb/block-autoplay` you may see how to enable this characteristic for a specific browser.

4.1 *Render.com*

Finally, deploy the project in *Render.com* and populate the database using the script populate. As it was described in the first assignment, in production (`settings.DEBUG=FALSE`), *Django* is not designed to serve static files, but it assumes some web server will be in charge of doing it and its only responsibility is to create the URLs that point to that data. In the following we describe how to serve static files in the different possible scenarios:

Local execution and DEBUG variable in settings.py set to TRUE *Django* will serve the static files without needing any further setting.

Local execution and DEBUG=FALSE The easiest way to serve the static content will be to start the server with the `-insecure` flag. That is:

```
./manage.py runserver --insecure 8001
```

Execution in *Render.com* and DEBUG=FALSE The static files must be part of the git repository that is uploaded to *Render.com* (they cannot be generated after doing `git push` even if `./manage.py collectstatic` is run afterwards). Besides, if you want to serve binary static files (for example, music) you have to incorporate the `whitenoise` module, as it was described in the first assignment.

4.2 Testing and coverage

To verify the correct implementation of the different models and services, several test files are provided, which include a set of tests (not necessarily complete) that your code must satisfy. These tests must be understood as additional requirements to the project.

Once you have managed to successfully run the tests, run the `coverage` command for the applications (`models` and `services`).

```
coverage erase
# xxxx is the name of the application to be tested
coverage run --omit="*/test*" --source=xxxx ./manage.py test xxxx
coverage report -m -i
```

Check the coverage of the files `models.py` and `views.py`. If it is not 100%, add to the test files those tests needed to reach that value. (The 100% coverage requirement applies exclusively to the code you create. Ignore the code created by *Django* or the one provided by your teachers.)

5 Work to be presented when the assignment ends

- Make sure your code satisfies all the provided tests. **Unless specified otherwise, it is not acceptable to modify the code in the tests**.
- Implement all the tests you consider necessary to cover the developed functionality. These tests must be implemented in a file called `test_additional.py`.
- Include a file called coverage.txt in the project root that contains the result of running the command `coverage` for all the tests.
- Deploy and test the application in *Render.com* in production mode (DEBUG=False, SECRET_KEY and DATABASE_URL in an environment variable).
- Upload to *Moodle* the obtained file when running the command `zip -r ../assign4_final.zip .git` from the root of the project. Remember you have to add and "commit" the files to git before running that command. If you want to check the content of the zip file is correct, you may do it by running the command: `cd ..; unzip assign4_final.zip; git clone . tmpDir; ls tmpDir`.
- Verify the variable `ALLOWED_HOSTS` from file `settings.py` included in the submission includes your deployment path in *Render.com* (if it does not appear, we will grade the assignment as if the project was not deployed in *Render.com*). In the same way, check in *Render.com* that your user name and password for admin are *alumnodb*.

6 Evaluation criteria

Note: When grading this assessment, the aesthetics will **NOT** be considered (it will be evaluated in the next assignment).
To pass with 5 points it is necessary to satisfy the following criteria completely:

- All the needed files to run the application have been submitted on time.
- The code was stored in a git repository and this repository is private.
- The file uploaded to *Moodle* contains a git repository.
- The script `populate.py` exists and is functional.
- The application can be executed locally.
- When running the tests in local, the number of fails is not larger than four and the code that satisfies the tests is functional.
- The code of the tests was not modified.
- The application works against the database created in `https://neon.tech` and implemented using *PostgreSQL*.
- The database admin application is deployed and accessible in the local *Django* server using as user name and password *alumnodb*.
- It is possible to create and remove objects belonging to all the requested models using the admin application.
- IMPORTANT: we need `https://neon.tech` URI in order to grade this assignment. Please write it down in a file called "env" and place it in the project root directory.

If the following criteria are accomplished, a grade up to 6.9 might be achieved:

- All the criteria in the previous paragraph are totally satisfied.

- The application is deployed in *Render.com*. In the file `settings.py` the path to *Render.com* is assigned to the variable `ALLOWED_HOSTS`. Besides being deployed, the application works correctly in *Render.com*.
- The code submitted to *Moodle* is identical to the one deployed in *Render.com*.
- The database admin application is deployed and accessible in *Render.com* using as user name and password *alumnodb*.
- It is possible to create and remove objects belonging to all the requested models using the admin application.
- All the "templates" inherit from `base.html`.
- IMPORTANT: If you update the variable `ALLOWED_HOSTS` using an enviroment variable add it to the "env" file.

If the following criteria are accomplished, a grade up to 7.9 might be achieved::

- All the criteria in the previous paragraph are totally satisfied.
- *Render.com* is deployed in production mode. `DEBUG=FALSE` and `SECRET_KEY` is not stored in `settings.py`.
- All the views (classes/methods implemented in `views.py`) inherit from classes.
- When the tests are executed, the number of fails is not larger than two and the code that satisfies the tests is functional.
- The code is readable, efficient, well-structured, and commented.
- The tools provided by the framework are used.
- The following are examples of the previous points:
  - Every form that involves a "model" is created in such a way that it inherits directly or indirectly from class `forms.Form`.

  - The searches are done by the database. Do not not load all the elements of a table and implement the search in the views defined in `view.py`.
  - The errors are properly processed and understandable message errors are returned.
  - The code presents a consistent style and the functions are commented including their author. Note: the author of a function must be unique.
  - The style criteria highlighted by *Flake8* are applied in a coherent way. *Flake8* does not return any error when executed on the code programmed by the student.

- It is impossible to impersonate a user (or participant) without knowing their user name and password (or `game.publicId`). For example: (a) it is not possible to modify a questionnaire/question/answer without previously doing a login and accessing directly to the corresponding URL, (b) it is not possible to create answers `guess` without knowing the `game.publicId`, etc.

If the following criteria are accomplished, a grade up to 8.9 might be achieved:

- All the previous criteria are accomplished completely.
- Every test and all the run checks output success results.
- If we reduce the size of the browser window or use the zoom, all the elements in the page are still accessible and no functionality is lost.

To aim for the maximum grade, the following criteria must be accomplished:

- All the previous criteria are accomplished completely.
- The coverage for the files that contain the models, views, and forms is over 99%.
- Sound in pages seen by participants was implemented.

Note: Late submission → take away a point for each late day (or fraction) in the submission.

Note: The code used in the assessment of the assignment will be the one submitted to *Moodle*. Under no circumstance, the existing code in *Render.com*, *Github*, or any other repository will be used.

A Images

Figure 1: Example of "homepage", the user is not connected.

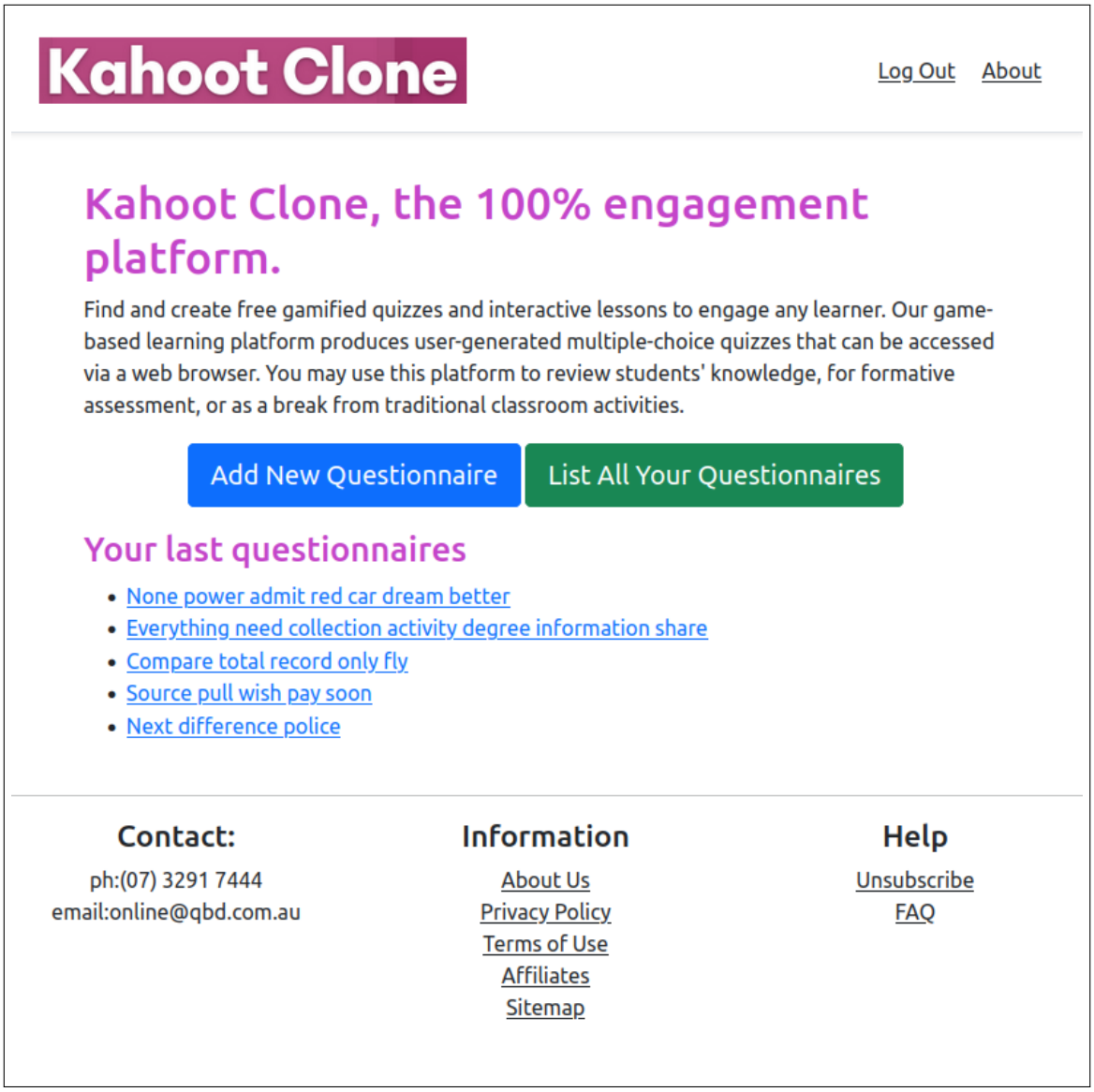

Figure 2: Example of "homepage", the user is connected.

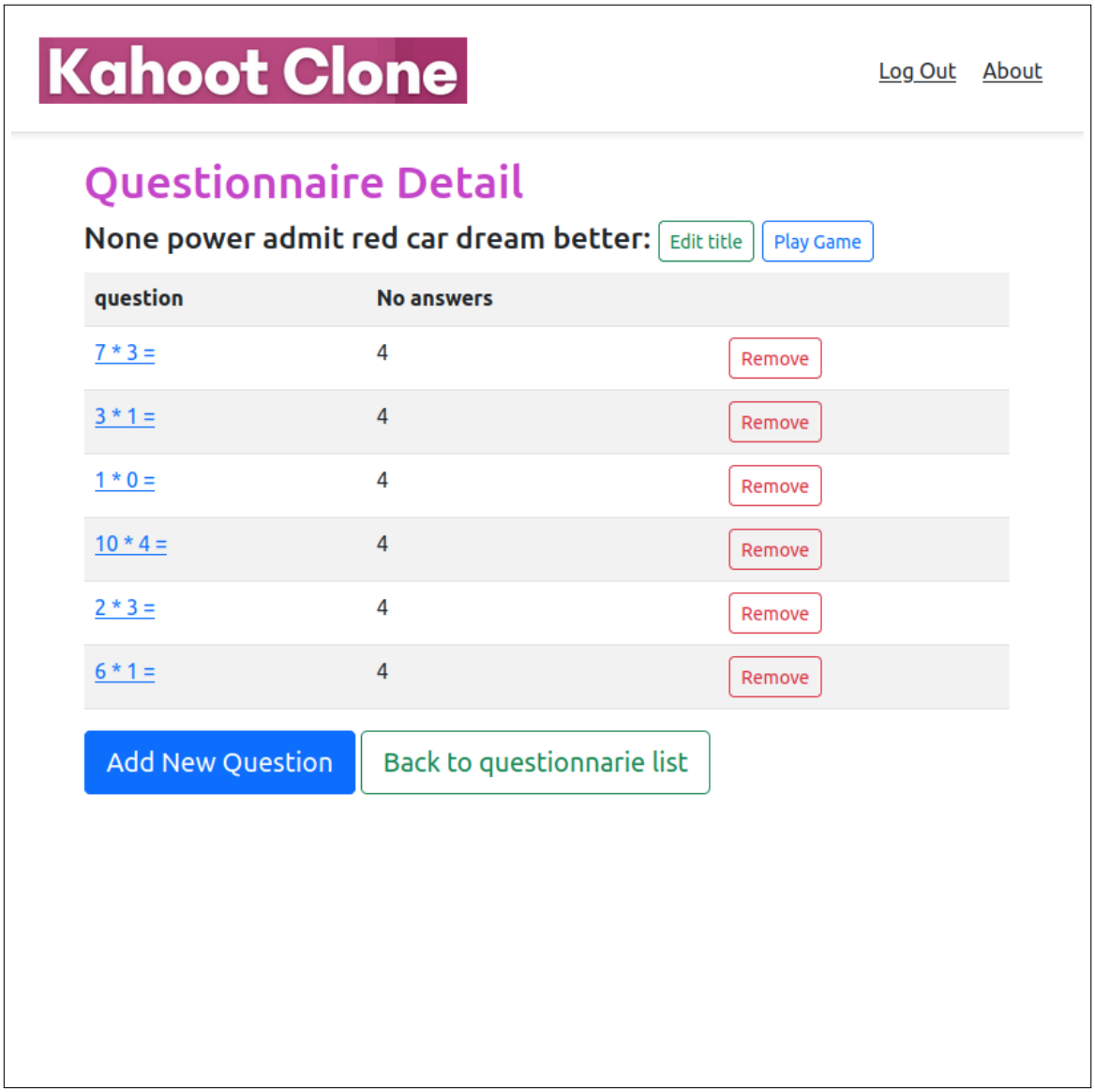

Figure 3: Page showing the questions that belong to a questionnaire.

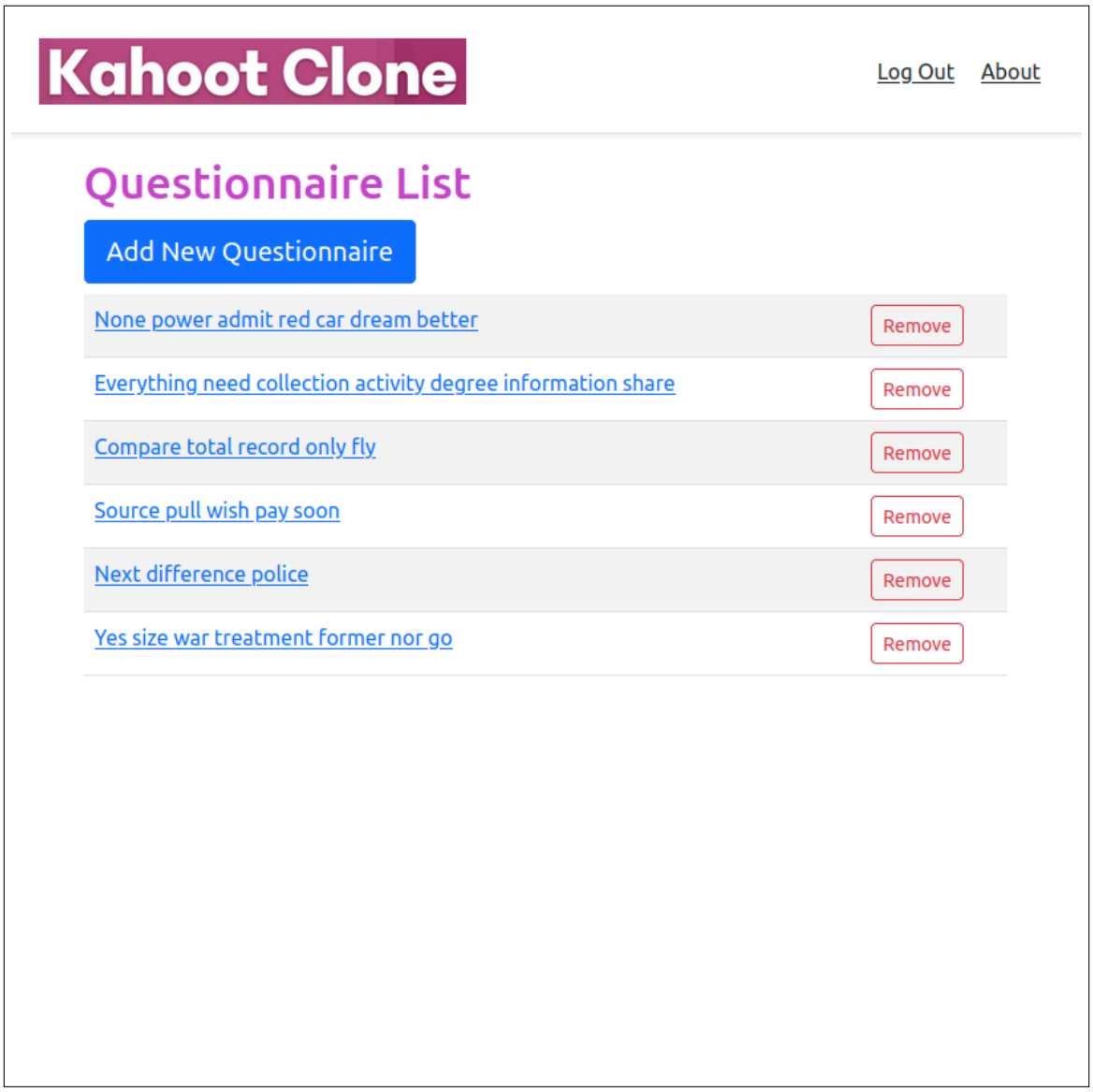

Figure 4: Page showing a listing with all the questionnaires that belong to the connected user.

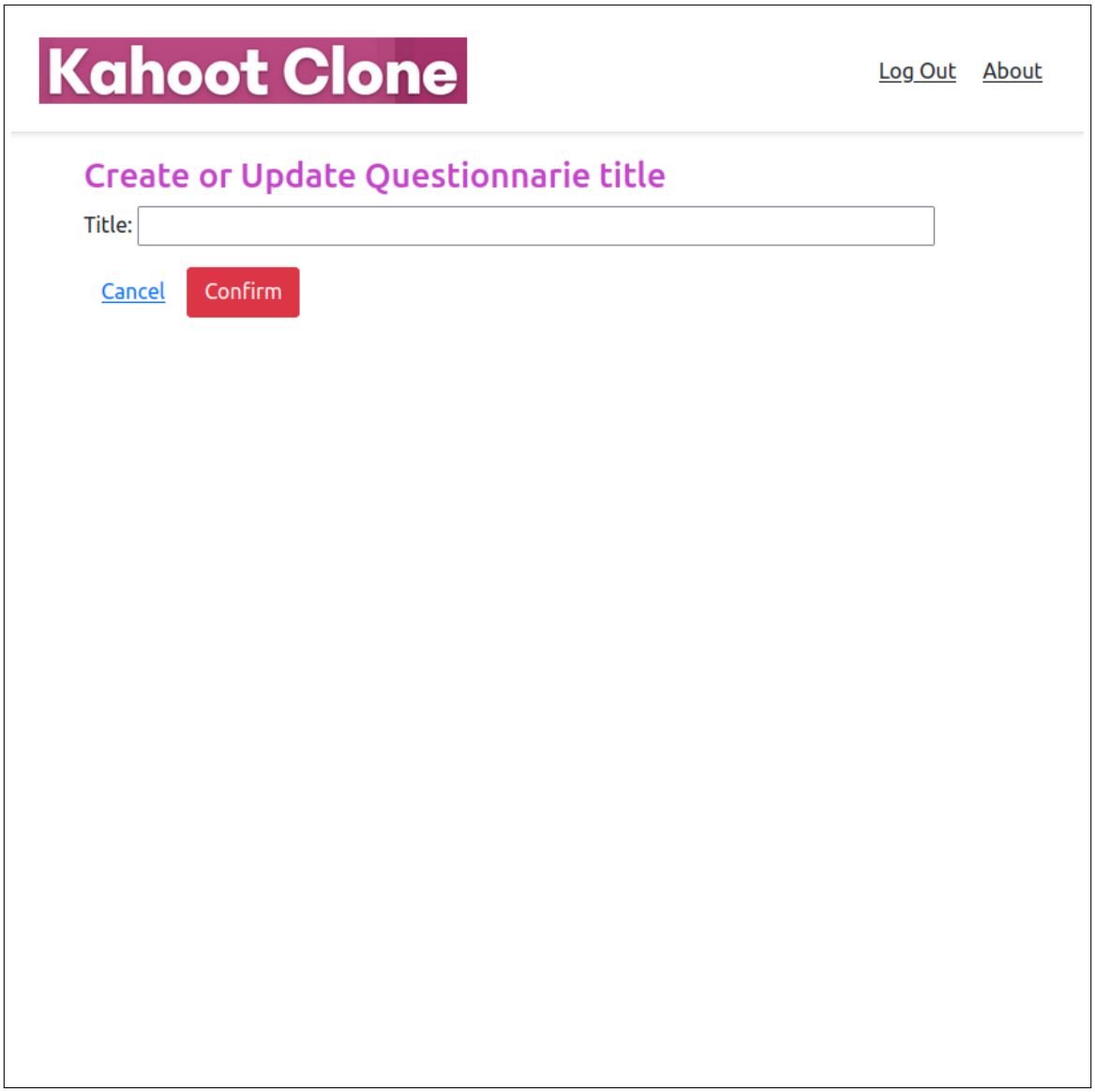

Figure 5: Page used to create or modify a questionnaire.

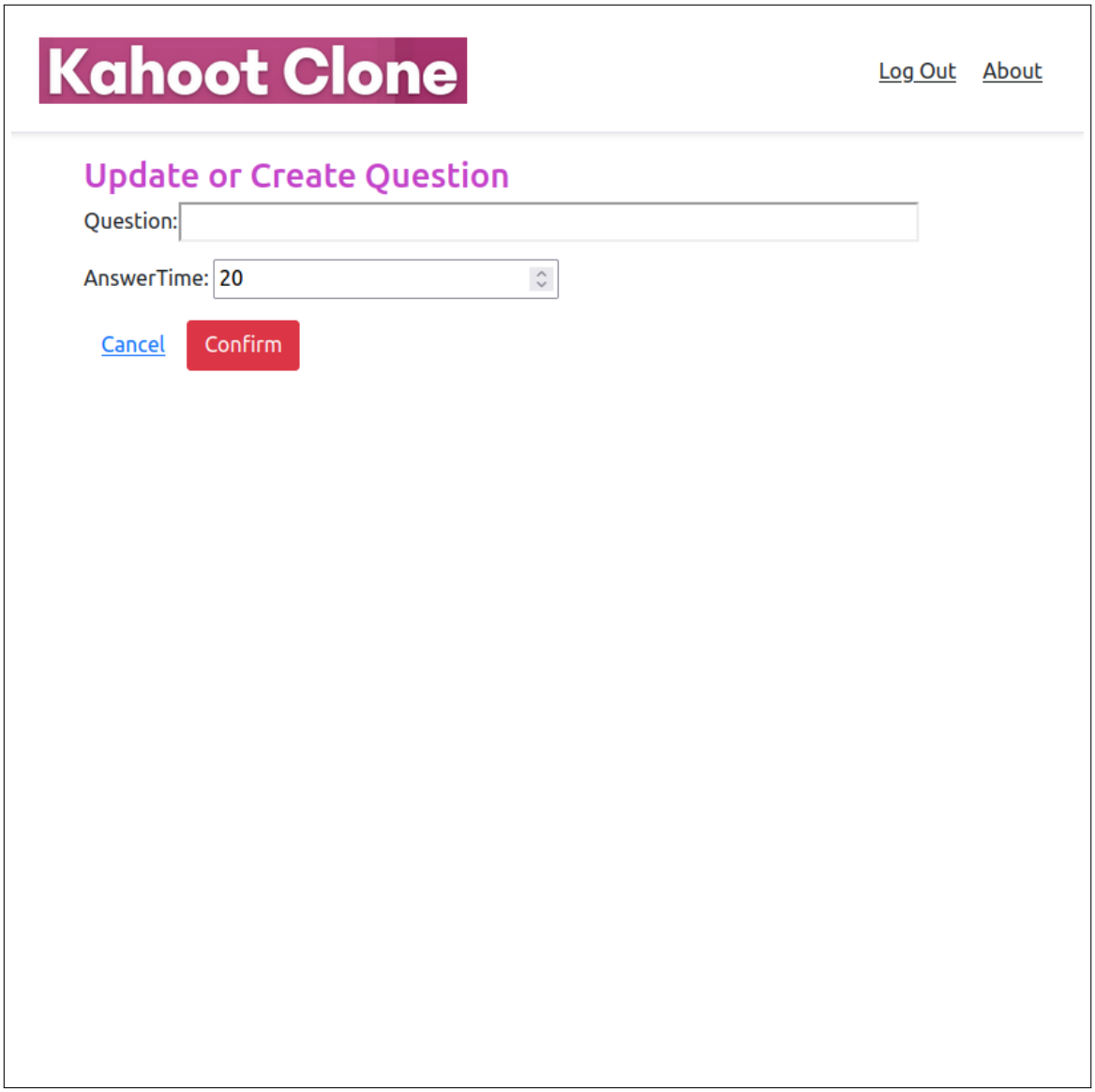

Figure 6: Page used to create or modify a question.

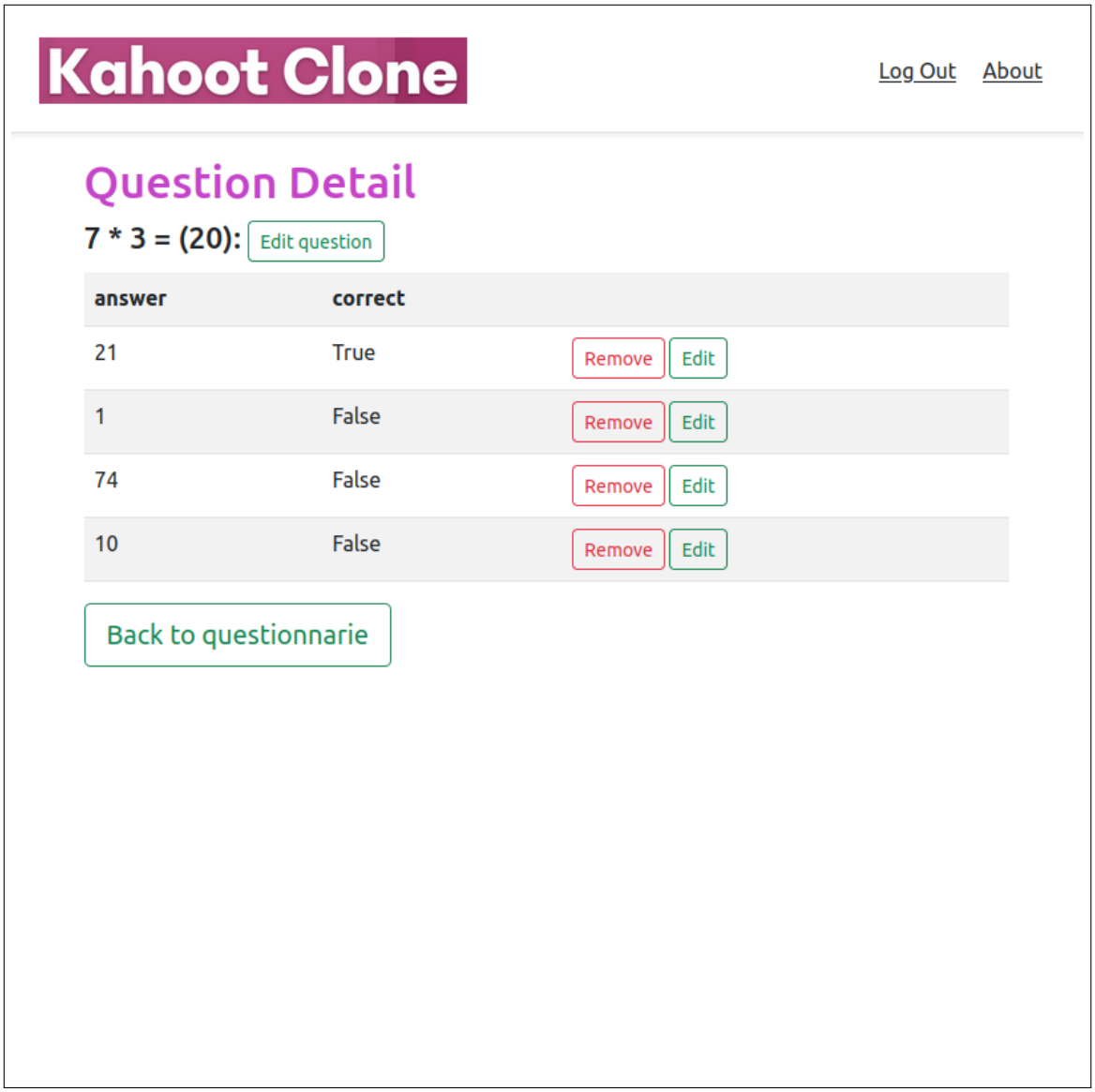

Figure 7: Page showing the answers that belong to a question.

Figure 8: Page used to create or modify an answer.

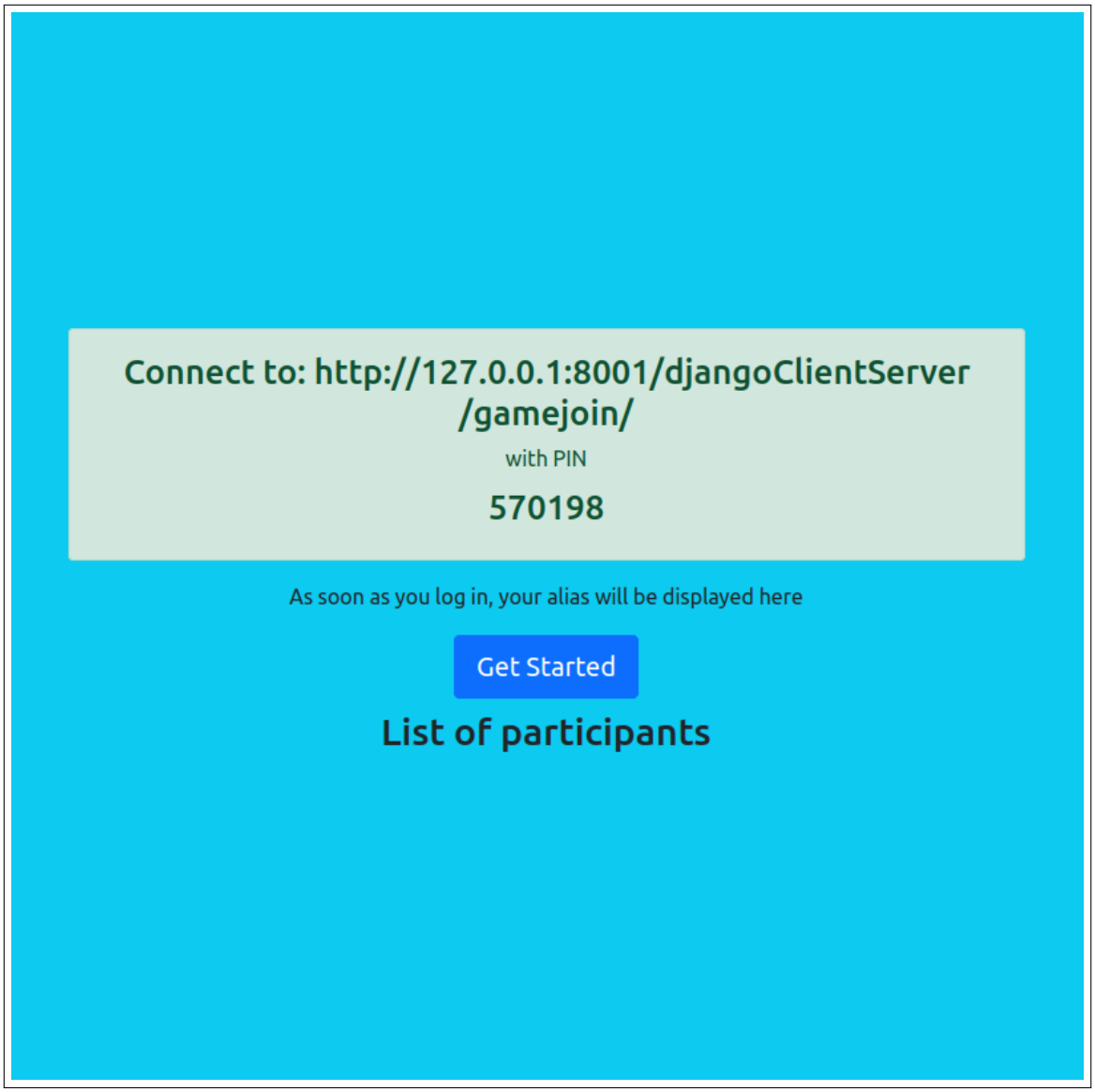

Figure 9: Result of running the view that creates a `game`.

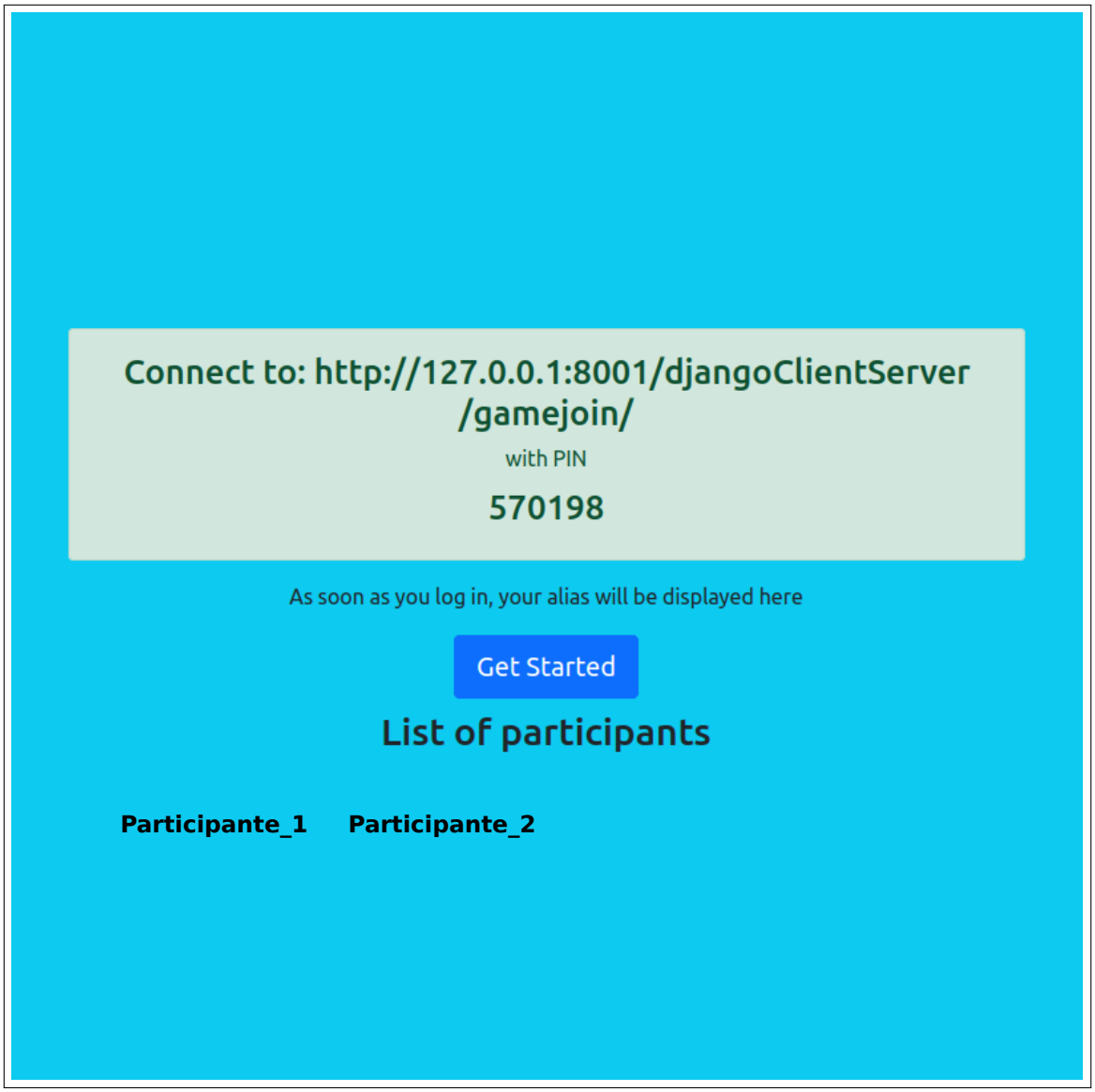

Figure 10: Result of running the view that updates the list of participants.

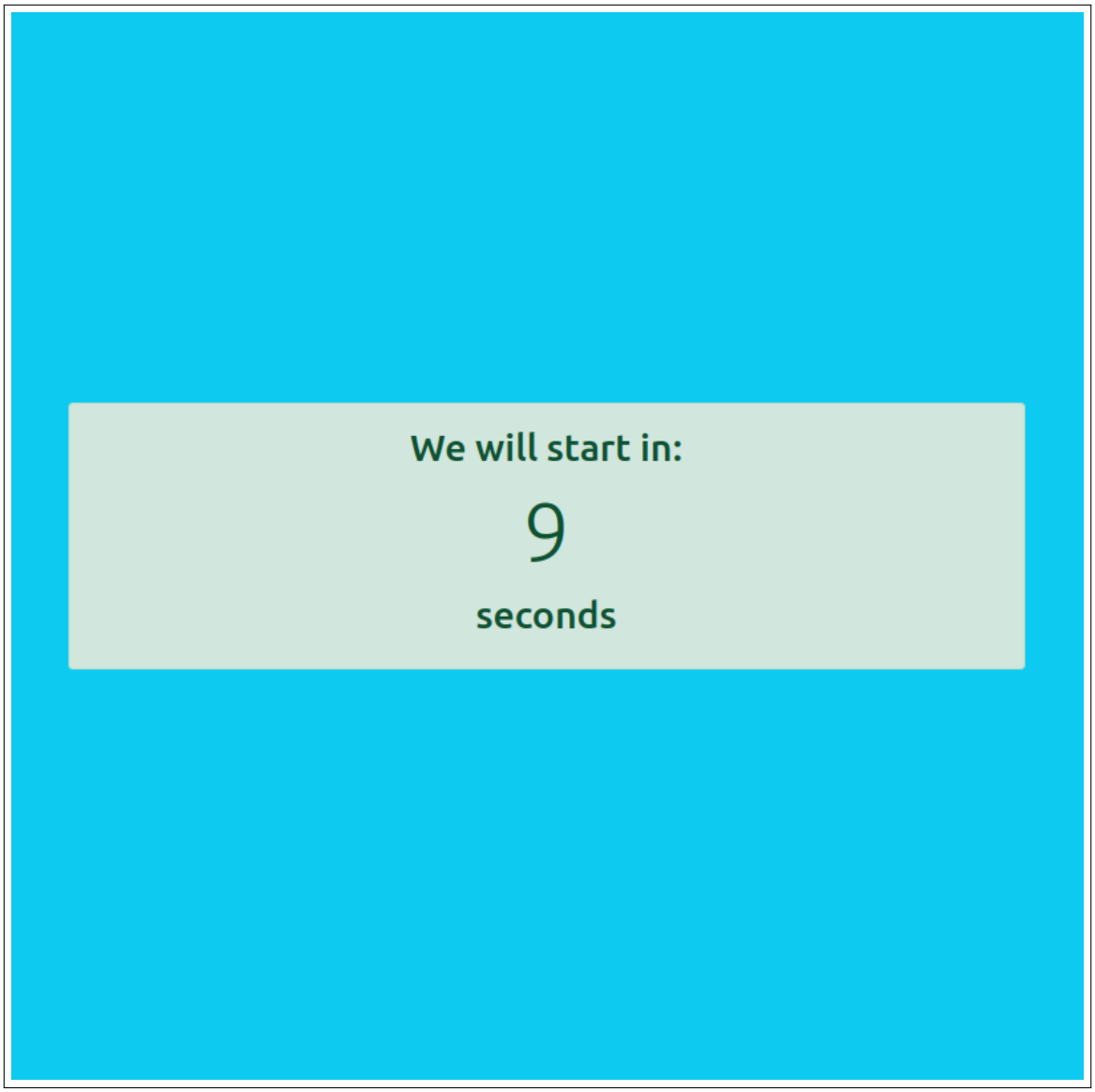

Figure 11: Page that notifies the participants that the game will start shortly.

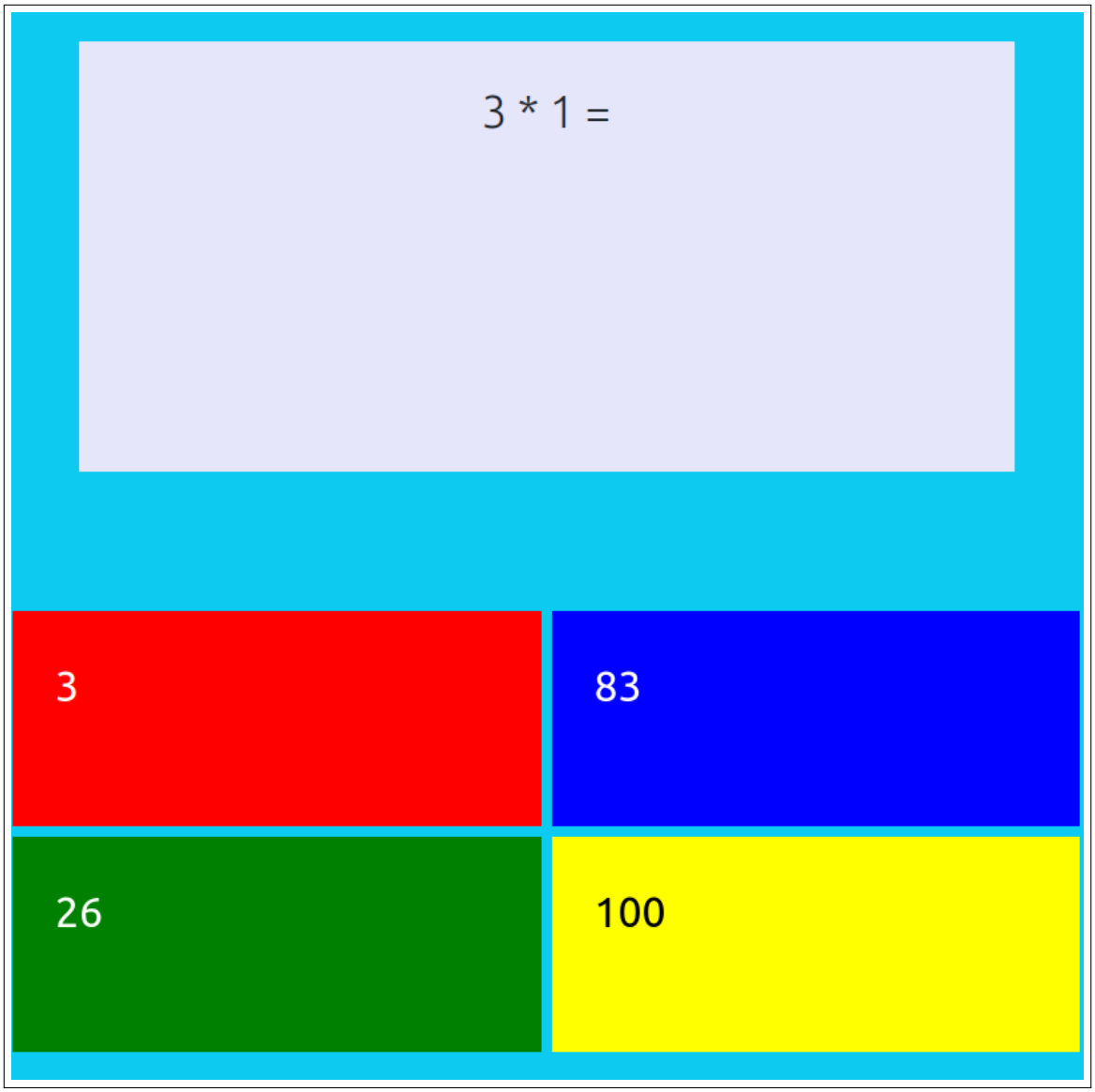

Figure 12: Page showing a question and its answers to the participants.

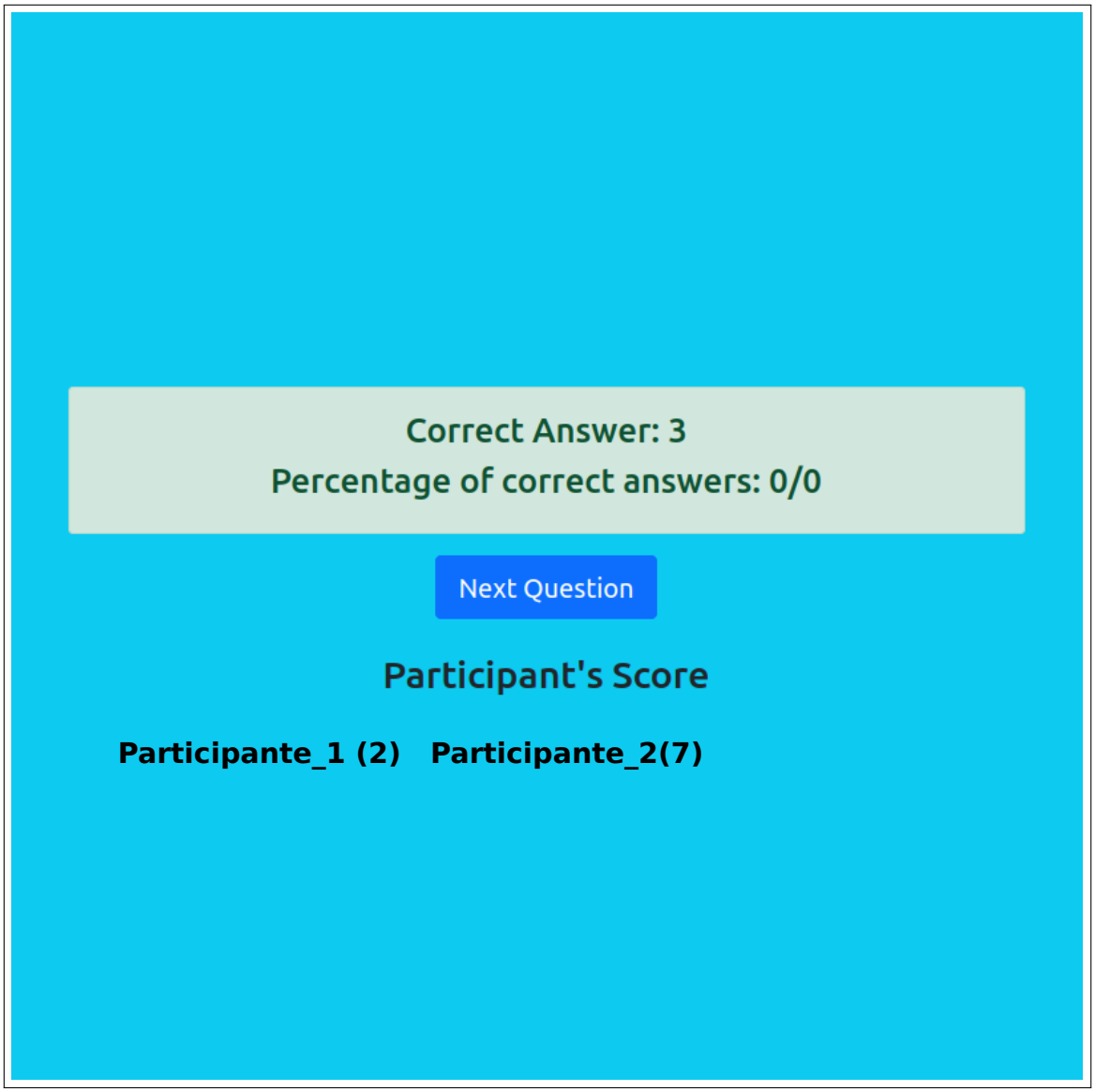

Figure 13: Page showing the result of a question and the score of the participants.

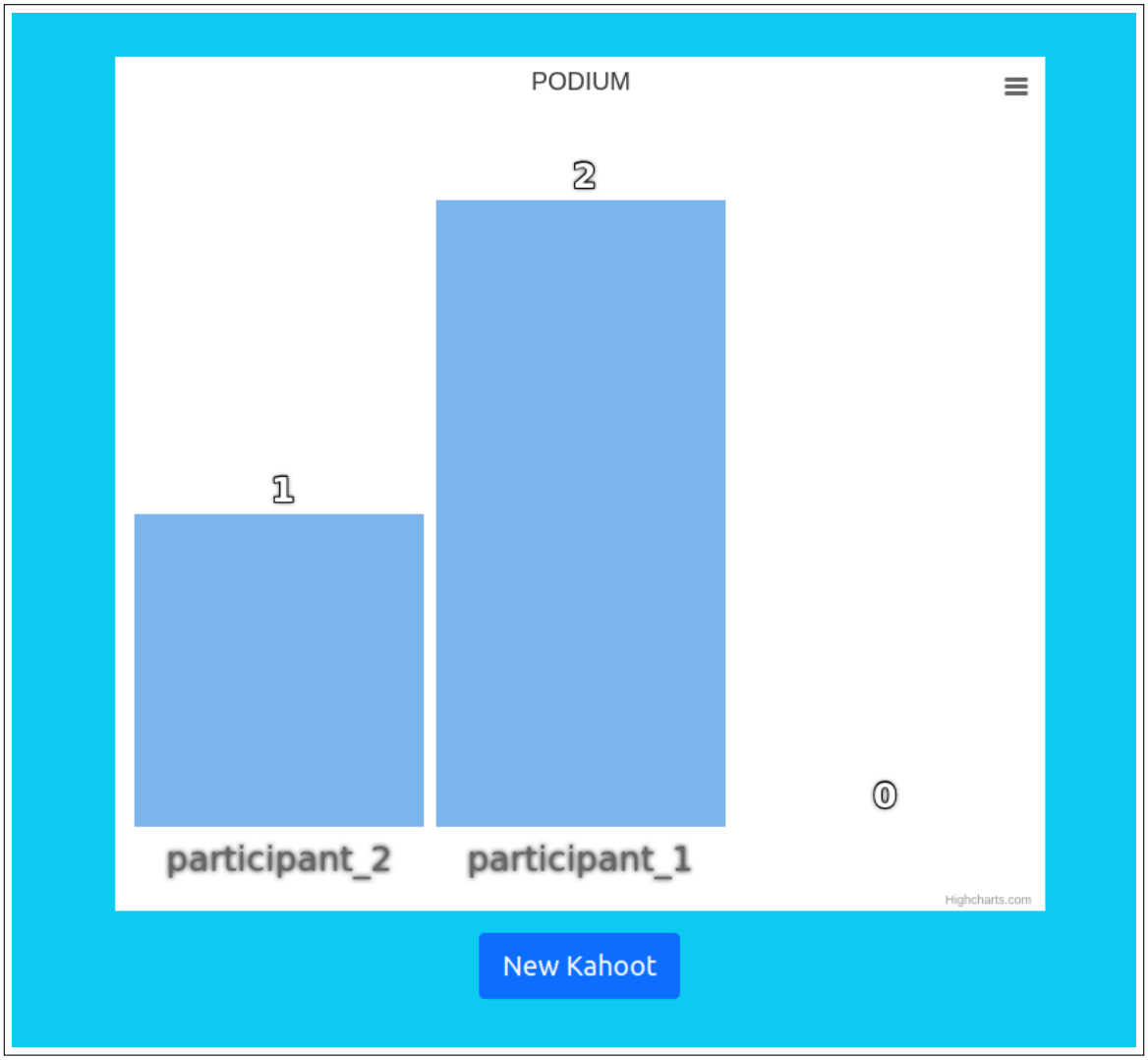

Figure 14: Page showing the final score (ranking).