

UNIVERSIDAD AUTÓNOMA DE MADRID

DEPARTAMENTO DE INFORMÁTICA

Proyecto de Sistemas Informáticos

Práctica - 3

Roberto MARABINI
Alejandro BELLOGÍN

Registro de Cambios

Versión ¹	Fecha	Autor	Descripción
1.0	10.08.2022	RM	Primera versión.
2.0	26.10.2022	RM	Cambio heroku → render.com
2.1	5.12.2022	RM	Renumerar práctica 4 ->3
2.2	22.12.2022	AB	Traducción al inglés
2.3	29.1.2023	RM	Revisión antes de subir la guía a <i>Moodle</i>
2.4	12.2.2023	RM	Cambiar Elephantsql por neon.tech

¹La asignación de versiones se realizan mediante 2 números *X.Y*. Cambios en *Y* indican aclaraciones, descripciones más detalladas de algún punto o traducciones. Cambios en *X* indican modificaciones más profundas que o bien varían el material suministrado o el contenido de la práctica.

Índice

1. Objetivo	3
1.1. Requerimientos	3
1.2. Sistema de Control de versiones: <i>git</i>	4
2. Implementación	4
2.1. Makefile	5
2.2. Usuarios	6
2.2.1. “Testing”	7
2.2.2. Git	7
2.3. Modelo de datos	8
2.4. Testing y coverage	11
2.5. Probando los Modelos	13
2.6. Servicios: creación de cuestionarios	14
3. Resumen del trabajo a realizar durante la primera mitad de la práctica	18
4. Servicios: uso de los cuestionarios	19
4.1. <i>Render.com</i>	22
4.2. Testing y coverage	22
5. Trabajo a presentar al finalizar la práctica	24
6. Criterios de evaluación	25
A. Imágenes	29

1. Objetivo

Muchos de vosotros estaréis familiarizados con la plataforma “Kahoot”, que permite la creación de cuestionarios con preguntas y respuestas (<https://kahoot.com/>). Esta herramienta facilita el aprendizaje de conceptos como si se tratara de un concurso.

Se desea implementar una aplicación web que contenga la funcionalidad básica de “kahoot” y permita la realización de cuestionarios “on-line”. Principalmente usaremos los entornos *Django* y *Vue.js* para realizar la aplicación propuesta. En los URLs <https://kahooclone.onrender.com> y <https://kahootclone-render-vue.onrender.com> podéis ver la implementación que realizamos de esta aplicación como paso previo a la escritura de esta práctica. Cread un usuario para explorar la aplicación.

1.1. Requerimientos

A muy grandes rasgos, la aplicación que deseamos crear deberá:

1. Permitir la creación e identificación de usuarios. Los usuarios son las personas que crean y mantienen los cuestionarios.
2. Permitir a los usuarios, mediante una colección de formularios, crear cuestionarios.
3. Permitir a los usuarios mostrar y gestionar los cuestionarios creados. Esto es, añadir, quitar y modificar cuestionarios con sus correspondientes preguntas y respuestas.
4. Permita a los usuarios ejecutar un cuestionario. Esto es, hacer pública una instancia de un cuestionario para que un conjunto de participantes pueda resolverlo. Los participantes pueden conectarse a instancias de los cuestionarios y responder a las preguntas de los mismos pero no pueden modificar los cuestionarios. Los participantes no necesitan “loguearse en el sistema” para contestar al cuestionario.

5. Gestionar el acceso de los participantes a las instancias públicas de los cuestionarios.
6. Llevar la contabilidad de las respuestas dadas por cada participante y presentar la puntuación de los distintos participantes.
7. Además, la parte administrativa, esto es, cualquier labor llevada a cabo por los usuarios, se implementará en *Django*.
8. El interfaz de los participantes se realizará en *Vue.js* conectándose a un API REST creado en *Django*.

El código debe

1. Satisfacer los criterios de estilo marcados por la utilidad *Flake8*.
2. Usar las versiones de los módulos de python que se enumeran en el fichero “requirements.txt” (*Django*) y en el fichero makefile option “requirements” (*Vue.js*) accesibles en *Moodle*.
3. Almacenar la información en una base de datos creada con el gestor de bases de datos *PostgreSQL* y almacenada en <https://neon.tech>.

1.2. Sistema de Control de versiones: *git*

Debéis utilizar *git* como herramienta de control de versiones. Entre el material a entregar en esta práctica tendréis que incluir el repositorio de *git* utilizado y este debe contener accesos (*git commit*) frecuentes de AMBOS miembros de la pareja. Al menos debe existir un acceso semanal de cada miembro de la pareja. Es importante que el repositorio sea PRIVADO, **todo pareja que cree un repositorio público será automáticamente suspendida.**

2. Implementación

En esta práctica nos concentraremos en crear las funciones y el interfaz necesario para crear los cuestionarios mientras que en la práctica siguiente se procederá a

implementar la infraestructura usada por los participantes.

Para empezar a trabajar os hará falta crear un proyecto de *Django* llamado *kahootclone*, una “template” base similar a `base.html` (véase ficheros auxiliares en el repositorio de <https://github.com/rmarabini/psi-alumnos>) de la cual heredarán todos los ficheros “html” que creemos en el futuro. Finalmente, usando *PostgreSQL*, cread una base de datos en <https://neon.tech>. Modificar el fichero `settings.py` de *Django* para que use esta base de datos. Recordad que `settings.py` no debería incluir información que pueda ser usada por los hackers como la password de la base de datos creada en <https://neon.tech> por ello leedla en `settings.py` desde una variable de entorno (ejemplo en el listado 1).

2.1. Makefile

Durante el desarrollo de la práctica es normal ejecutar de forma repetitiva algunos comandos. Para automatizar el proceso se puede usar el comando `make` junto con el fichero `makefile` que tenéis disponible en el material de esta práctica y que usaremos para corregir vuestras practicas.

En `makefile` se han definido, entre otras, las siguientes operaciones:

- `create_super_user`: crea un usuario con permisos de administración. El nombre de usuario y la clave son `alumnodb`. Equivalente a `python3 ./manage.py createsuperuser`
- `populate`: puebla la base de datos, equivalente a `python3 ./manage.py populate`
- `runserver`: equivalente a `python3 manage.py runserver 8001`
- `update_models`: equivalente a `python3 manage.py makemigrations; python3 manage.py migrate`
- `dbshell`: lanzar el cliente de *PostgreSQL* `psql` (`./manage.py dbshell`)
- `shell`: lanza python con el entorno de *Django* cargado (`./manage.py shell`)

2.2. Usuarios

Antes de describir el modelo de datos del proyecto *kahootclone* vamos a implementar la gestión de usuarios. La gestión de usuarios es clave en un gran número de aplicaciones así que haremos un diseño que pueda ser reusado posteriormente con facilidad. En la medida de lo posible usaremos la implementación que viene con la distribución de *Django*. Necesitamos implementar los servicios de **log-in** (iniciar sesión), **log-out** (cerrar sesión) y **sign-up** (crear un usuario nuevo). Comenzaremos creando una aplicación llamada **models** donde se guardará el código.

Pasamos a describir en detalle los diversos servicios solicitados:

login Para el servicio **login** no hace falta crear ninguna función sino reusar las vistas disponibles en `django.contrib.auth.urls` tal y como se mostró en la práctica 1. Al acceder a la página de log-in se debe mostrar un formulario que recogerá el nombre de usuario (`username`) y la clave (`password`). Si la identificación es positiva el usuario será redirigido a la “homepage” en caso contrario reaparecerá la página de “login” junto al correspondiente mensaje de error. No uséis directamente la clase `User` definida en `django.contrib.auth.models` sino crear vuestro propio modelo `User` heredando de `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 Para el servicio **logout** no hace falta crear ninguna función sino reusar las vistas disponibles en `django.contrib.auth.urls` tal y como se mostró en la práctica 1. Al seleccionar **logout** desde el menú el usuario será redirigido a la “homepage” y su sesión será cerrada.

signup puede ser implementado reusando el formulario `UserCreationForm` (from `django.contrib.auth.forms import UserCreationForm`) junto a una pequeña vista, véase ejemplos en <https://simpleisbetterthancomplex.com/tutorial/2017/02/18/how-to-create-user-sign-up-view.html>). Al acceder a la página de “sign-up” se debe mostrar un formulario que preguntará por el nombre de usuario (`username`) y la clave por duplicado (`password1` y `password2`). Tras finalizar la creación del nuevo usuario este debe conectarse automáticamente.

En todos los servicios a implementar, no uséis las “templates” que se ofrecen por defecto sino que debéis personalizarlas para que hereden de `base.html`.

La “homepage” deberá comprobar si el usuario está “conectado” mostrando un enlace a los servicios `login` y `sign-up` en caso negativo y un enlace al servicio `logout` en caso positivo.

Las variables `LOGIN_REDIRECT_URL` y `LOGOUT_REDIRECT_URL` (`settings.py`) pueden ayudaros con las redirecciones solicitadas en `login` y `logout`.

2.2.1. “Testing”

Para verificar la correcta implementación de los diferentes servicios, se proporciona una batería de test (no necesariamente completa) que debe satisfacer tu código. Estos test deben entenderse como requerimientos adicionales del proyecto.

En concreto, en el fichero `models/test_authentication.py` se proporcionan 4 test que verifican el correcto funcionamiento de las funcionalidades `log-in`, `log-out` y `sign-up`.

NOTA IMPORTANTE: A la hora de realizar la implementación, se recomienda seguir una estrategia TDD (test-driven development) tratando de satisfacer uno a uno y siguiendo el orden establecido cada uno de los test. Esto aplica para toda la implementación del proyecto con los test proporcionados.

2.2.2. Git

Como siempre asegurados de salvar vuestro código periódicamente en *Github*. Usadlo para compartir vuestro código con el otro miembro de la pareja. No uséis otras opciones como puedan ser “Live share” de *Visual Studio* o *Dropbox*.

Se enumeran algunos de los comandos más utilizados para gestionar el repositorio.

```
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. Modelo de datos

Además de usuarios, nuestro proyecto necesita gestionar cuestionarios con sus preguntas y respuestas así como las puntuaciones de cada participante.

El modelo de datos que dará soporte a la aplicación seguirá el esquema ORM (Object Relational Mapping) de *Django*. Necesitamos cuestionarios, preguntas, respuestas, puntuaciones, etc. A continuación se muestra el esquema relacional que contiene el diseño mínimo que debes usar en vuestra implementación. Puedes añadir cualquier entidad o atributo que consideres necesario, pero **no eliminar** ninguno de los propuestos:

```

Questionnaire(questionnaire_id, title, created_at, updated_at, user↑)
Question(question_id, question, questionnaire↑, created_at, updated_at, answer-
Time)
# se asume que cada pregunta (question) aparece en un único cuestionario
(questionnaire)
# question.question es una cadena de caracteres con la pregunta a realizar
# question.answerTime es un entero igual al tiempo, en segundos, que se da para
contestar esta pregunta
Answer(answer_id, answer, question↑, correct)
# answer.answer es una cadena de caracteres con la respuesta
# correct es un booleano que indica si esta respuesta es correcta o no
# Se asume que existe una única respuesta correcta a cada pregunta
# El formulario para introducir las
# respuestas debe tener en cuenta esta limitación, esto es,
# sólo puede haber una respuesta correcta por pregunta.
# Se asume que una respuesta está relacionada con una única pregunta.
game(game_id, questionnaire↑, created_at, state, publicId, countdownTime,
questionNo)
# el modelo game se explica en detalle al final de esta sección
Participant(participant_id, game↑, alias, points, uuidP)
# alias es el apodo por el que se conocerá al participante en el juego
# points guarda la puntuación del participante en el juego
# Cada participante está relacionado con un único juego
# uuidP es una cadena que almacena identificadores únicos.
# usa un campo UUIDField para almacenarla y rellénala con uuid.uuid4
# servirá para identificar al participante
Guess(guess_id, participant↑, game↑, question↑, answer↑)
# guess es cada una de las respuestas dadas por un participante
# nótese que question y game son redundantes pues puede obtenerse a partir de
# answer y participant pero facilitan las consultas a la base de datos.

```

En el esquema las claves primarias y extranjeras se denotan usando negrita y el

símbolo \uparrow respectivamente. Todas las fechas (`created_at` y `updated_at`) deben inicializarse por defecto con el instante de creación del objeto. Además `updated_at` debe actualizarse automáticamente cada vez que se guarde el objeto.

Cada vez que se desea jugar se debe crear un objeto de tipo `game`. En esta clase, el atributo `state` puede tomar los siguientes valores

WAITING = 1 Estado por defecto cuando se crea un `game`, el sistema espera a que los participantes se unan al juego.

QUESTION = 2 Mostrar la siguiente pregunta disponible así como las respuestas a la misma

ANSWER = 3 Tras un periodo de tiempo (`question.answerTime`) se mostrará la respuesta correcta y una clasificación con la puntuación de los jugadores.

LEADERBOARD = 4 Al finalizar el juego se muestra un podio con el resultado final.

El atributo `game.publicId` es un número entero único (como la clave primaria) en el rango $[1 - 10^6]$. Este número se muestra a los participantes y es utilizado para identificar el juego al que deben unirse. Se debe generar de forma automática al crear el juego usando un generador de números aleatorios. Mostrar a los usuarios un número aleatorio en lugar de un entero con valores $1, 2, 3, \dots$ aumenta ligeramente la seguridad de la aplicación pues dificulta acceder al juego (`game`) a participantes que no han sido invitados al mismo.

Para facilitar la implementación hemos minimizado las relaciones muchos a muchos, esto es, una pregunta se encontrará en un único cuestionario, un participante puede jugar un único juego etc. Por supuesto se pueden crear dos objetos pregunta idénticos, salvo en la clave primaria, y asignarlos a dos cuestionarios diferentes.

Recuerda que:

- *Django* crea de forma automática una clave primaria llamada `id` para cada uno de los modelos por lo tanto no la crees tu explícitamente.
- hay que crear una función `__str__` para cada modelo.
- todos los modelos creados deben ser accesibles usando el interfaz de administración de *Django* disponible en <http://localhost:8001/admin/>.

- todas las fechas deben almacenar tanto el día como la hora en que el objeto ha sido creado o modificado usando un atributo de tipo `DateTimeField`.
- la forma más sencilla de rellenar los atributos `game.publicId` y `participant.points` es redefiniendo la función `save` de los modelos `Game` y `Guess` respectivamente.
- una vez que un participante ha emitido una respuesta (`guess`) no podrá modificarla.

2.4. Testing y coverage

Para verificar la correcta implementación de los diferentes modelos, se proporciona el fichero `models/test_models.py` con una batería de test (no necesariamente completa) que debe satisfacer tu código. Estos test deben entenderse como requerimientos adicionales al proyecto.

Recordad que para ejecutar los test *Django* debe ser capaz de crear un base de datos auxiliar. Por ello a la hora de ejecutar los test debes hacerlo contra una base de datos local.

Listado 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 a 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 = {}
```

```
POSTGRES_URL = 'postgres://alumnodb:alumnodb@localhost/psi '
# please do not include sensitive information as the neon
# password in settings.py, just read it from the environment

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 interesting
    # to have access to a local postgres database
    # so if DATABASE_URL is not defined use POSTGRES_URL
    db_from_env =
        dj_database_url.config(
            default=POSTGRES_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)
```

Una vez que hayáis conseguido que los tests se ejecuten de forma satisfactoria ejecutad el comando `coverage`.

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

mirad la cobertura del fichero `models/models.py` y si no es del 100 % añadid los test que hagan falta para alcanzar este valor. (La solicitud de 100 % de cobertura aplica exclusivamente al código que creéis vosotros. Ignorad el código creado por *Django* o el suministrado por vuestros profesores.)

2.5. Probando los Modelos

Para poder verificar en correcto funcionamiento de la aplicación web se necesita almacenar datos de prueba en la misma.

Poblar la Base de Datos Crea un “script” llamado `populate.py` que genere objetos de los diferentes modelos y los persista en la base de datos del proyecto. Usa como guía el fichero llamado `populate.py` que se entrega junto a la documentación de esta practica (véase repositorio). Este “script” tiene una estructura que le permite ser invocado usando la línea de comandos `python3 ./manage.py populate` (o `make populate`).

El fichero debe situarse en el directorio `management/commands` (el “path” está dado desde el directorio que contiene la aplicación *models*). Si el directorio no existe creadlo.

Se sugiere el uso del modulo `Faker` para la generación de los datos (véase <https://zetcode.com/python/faker/>).

2.6. Servicios: creación de cuestionarios

Continuaremos la implementación de la aplicación web creando los servicios que permitirán crear/borrar/modificar los cuestionarios. Usad vistas basadas en clases (*class based views*), esto es, modelad nuestras vistas como clases que hereden de clases tipo `CreateView`, `UpdateView`, etc. Comenzad creando una aplicación llamada `services` e implementad en ella las nuevas vistas.

A continuación se listan las vistas a implementar. Para cada vista primero se muestra el alias (`name` en el fichero `urls.py`), el URL al que estará conectado, el test que puede usarse para verificar la vista y finalmente la parte relevante del diccionario `context` devuelto así como una somera descripción del comportamiento esperado y de la “template” a usar. En el URL se ha suprimido la parte común a todas las vistas, esto es, `/services`. Obviamente cada vista debe llamar a una “template” y mostrar el resultado al usuario, en esta práctica no valoraremos la estética de estas “templates” pero: (1) debéis implementar al menos el mínimo necesario para probar las vistas y (2) todas las “templates” deben heredar de una “template” base. Os recomendamos que uséis algún “framework CSS” como puedan ser “Bootstrap” o “Bulma”. Bulma es más sencillo pero Bootstrap es más completo.

`home` | `"` (cadena vacía) | `test01_home` | Si el usuario está conectado se devuelve un listado con SUS últimos cinco cuestionarios ordenados por el atributo `updated_at`, en caso contrario no se devuelve nada. Si el usuario no está conectado en la “template” se mostrarán las opciones de “login” y “signup” (Fig. 1). Si el usuario está conectado aparte del listado con los 5 cuestionarios devueltos se ofrecerán las opciones de: “log-out”, creación de un nuevo cuestionario y mostrar todos los cuestionarios del usuario (Fig. 2). Igualmente debe ser posible seleccionar uno de los cuestionarios y acceder a una página donde se muestren en detalle.

`questionnaire-detail` | `questionnaire/<int:pk>` | `test02_questionnaireDetail` | Devuelve el cuestionario con clave primaria igual a `pk` si el cuestionario ha sido creado por el usuario conectado, si el cuestionario no ha sido creado por el usuario conectado devuelve un mensaje de error, finalmente si no hay ningún usuario conectado

se mostrará la página de login. La “template” debe mostrar el título del cuestionario así como cada una de las preguntas que lo integran (Fig. 3). Desde la página devuelta por esta vista debe ser posible borrar/editar/crear preguntas (**questions**) así como ejecutar un cuestionario (crear un **game**).

questionnaire-list | **questionnairelist/** | **test04_questionnaireList** | Devuelve un listado con todos los cuestionarios pertenecientes al usuario conectado. Si no hay ningún usuario conectado se mostrará la página de login. La “template” debe mostrar el listado de cuestionarios devuelto (Fig. 4) y ofrecer la posibilidad de eliminar cualquier cuestionario.

questionnaire-remove | **questionnaireremove/<int:pk>** | **test03_questionnaireRemove** | Borra el cuestionario con id=pk si pertenece al usuario conectado, si el cuestionario no ha sido creado por el usuario conectado devuelve un mensaje de error, finalmente si no hay ningún usuario conectado se mostrará la página de login. Se debe solicitar confirmación antes de proceder a borrar. Tras borrar el cuestionario se mostrará la página **questionnaire-list**.

questionnaire-update | **questionnaireupdate/<int:pk>** | **test05_questionnaireUpdate** | Modifica el cuestionario con id=pk si pertenece al usuario conectado, si el cuestionario no ha sido creado por el usuario conectado devuelve un mensaje de error, finalmente si no hay ningún usuario conectado se mostrará la página de login. Esta vista modifica los atributos propios de **questionnaire** pero no sus claves extranjeras (por ejemplo **question**) que serán modificadas usando otra vista. La Fig. 5 muestra un ejemplo de formulario usado para crear un cuestionario. Tras modificar el cuestionario se mostrará la página **questionnaire-detail** conteniendo el cuestionario recién modificado.

questionnaire-create | **questionnairecreate/** | **test06_questionnaireCreate** | Crea y devuelve un cuestionario nuevo perteneciente al usuario conectado. Si no hay ningún usuario conectado se mostrará la página de login. La “template” debe mostrar el cuestionario creado y ofrecerá la posibilidad de añadir/quitar/modificar preguntas. La Fig. 5 muestra un ejemplo de formulario usado para crear un cuestionario. Tras crear el cuestionario se mostrará la página **questionnaire-detail**

conteniendo el cuestionario recién creado.

`question-detail` | `question/<int:pk>` | `test12_questionDetail` | Devuelve la pregunta con clave primaria igual a `pk` si la pregunta ha sido creada por el usuario conectado, si la pregunta no ha sido creada por el usuario conectado devuelve un mensaje de error, si no hay ningún usuario conectado se mostrará la página de login. La “template” debe mostrar la pregunta y sus respuestas. Igualmente la “template” debe ofrecer la posibilidad de borrar/editar/añadir respuestas a la pregunta (Fig. 7). Si el numero de respuestas es cuatro, la opción de añadir una respuesta no debe ofrecerse.

`question-remove` | `questionremove/<int:pk>` | `test13_questionRemove` | Borra la pregunta con `id=pk` si ha sido creada por el usuario conectado, si el usuario conectado no ha creado la pregunta se devuelve un mensaje de error, si no hay ningún usuario conectado se devuelve la página de login. Se debe solicitar confirmación antes de proceder a borrar. Tras borrar la pregunta se mostrará la página `questionnaire-detail` con `id=pk`.

`question-update` | `questionupdate/<int:pk>` | `test15_questionUpdate` | Modifica la pregunta con `id=pk` si pertenece al usuario conectado, si la pregunta no pertenece al usuario conectado se devuelve un mensaje de error, si no hay usuario conectado se mostrará la página de login. La template debe mostrar la pregunta y sus respuestas tras la modificación. Esta vista modifica los atributos propios de `question` pero no sus claves extranjeras (por ejemplo `answer`) que serán modificadas usando otra vista. La Fig. 6 muestra un ejemplo de formulario usado para modificar una pregunta. Tras modificar la pregunta se mostrará la página `question-detail` con `id=pk`.

`question-create` | `questioncreate<int:questionnaireid` | `test16_questionCreate` | Crea y devuelve una pregunta asociada al cuestionario con `id=questionnaireid`. Si el cuestionario no ha sido creado por el usuario conectado se devuelve un mensaje de error, si no hay ningún usuario conectado se devuelve la página de login. La Fig. 6 muestra un ejemplo de formulario usado para crear una pregunta.

Tras crear la pregunta se mostrará la página `question-detail` conteniendo la pregunta recién creada.

- `answer-create` | `answercreate/<int:questionid>` | `test26_answerCreate` | Crea y devuelve una respuesta asociada a la pregunta con `id=questionid`. Si la pregunta no ha sido creada por el usuario conectado se devuelve un mensaje de error, si no hay ningún usuario conectado se devuelve la página de login. La Fig. 8 muestra un ejemplo de formulario usado para crear una respuesta. Tras crear la respuesta se mostrará la página `question-detail` con `id=answer.question.id`.
- `answer-remove` | `answerremove/<int:pk>` | `test23_answerRemove` | Borra la respuesta con `id=pk` si ha sido creada por el usuario conectado. Si la respuesta no ha sido creada por el usuario conectado se devuelve un mensaje de error, si no hay ningún usuario conectado se devuelve la página de login. Tras borrar la respuesta se mostrará la página `question-detail` con `id=answer.question.id`.
- `answer-update` | `answerupdate/<int:pk>` | `test25_answerUpdate` | Modifica la respuesta con `id=pk` si pertenece al usuario conectado. Si la respuesta no ha sido creada por el usuario conectado se devuelve un mensaje de error, si no hay ningún usuario conectado se devuelve la página de login. La Fig. 8 muestra un ejemplo de formulario usado para modificar una respuesta. Tras modificar la respuesta se mostrará la página `question-detail` con `id=answer.question.id`.
- `game-create` | `gamecreate/<int:questionnaireid>` | `test36_gameCreate` | Crea y devuelve un juego asociado a el cuestionario con `id=questionnaireid`. Si el cuestionario no ha sido creado por el usuario conectado se devuelve un mensaje de error, si no hay ningún usuario conectado se devuelve la página de login. No es necesario implementarlo ahora mismo pero usando Ajax la “template” debería conectarse periódicamente al servidor (vista `game-update-participant`) y mostrar la lista de participantes a medida que se vayan incorporando. La vista debe guardar en una variable de sesión el identificador de juego (`game.id`) el cual se usará en el futuro.

Algunos consejos sobre como implementar las vistas

Como se comentó al principio de la sección, las vistas solicitadas deben basarse en clases. Para implementar los requerimientos solicitados te puede hacer falta sobrescribir alguno de las vistas definidos por defecto para estas clases como puedan ser: `get_queryset`, `get_object`, `form_valid`, etc. `get_queryset` puede usarse para restringir los objetos a los que las vistas puede acceder a aquellos producidos por el usuario validado, `get_object` permite manipular el objeto (e.g. `questionnaire`) antes de pasárselo a la “template”, `form_valid` permite examinar y modificar el contenido del formulario que se va a usar para crear un objeto antes de crearlo.

3. Resumen del trabajo a realizar durante la primera mitad de la práctica

Crear un proyecto *Django* llamado *kahootclone* que incluya las aplicaciones *models* y *services* satisfaciendo los siguientes requerimientos:

- El proyecto contendrá una página de administración (interfaz *Django* en la dirección `http://hostname:8001/admin/`) que permita introducir y borrar datos. Tanto el nombre de usuario como la clave del usuario de administración deben ser *alumnodb*.
- Los datos deben persistirse en una base de datos *PostgreSQL* almacenada en `https://neon.tech`.
- La aplicación *models* contendrá el modelo de datos descrito en la sub sección 2.3 incluido el script `populate.py`.
- El código creado debe satisfacer los test definidos en `models.tests_authentication` y `models.test_models`.
- El “coverage” de los ficheros `models.py` debe ser del 100 %. La solicitud de 100 % de cobertura aplica exclusivamente al código que creéis vosotros. Ignorad el código creado por *Django* o el suministrado por vuestros profesores.

- La aplicación *services* contendrá al menos todos los servicios relacionados con el cuestionario (aquellos cuyo alias empieza por *questionnaire*) y el necesario para crear la página de inicio (alias *home*) (véase sección 2.6).
- El código creado debe satisfacer al menos los primeros seis test definidos en `services.test_services.py`.
- Todo el código Python que escribáis debe satisfacer los requerimientos de estilo marcados por la utilidad *flake8*. Este requerimiento no se extiende al código generado automáticamente por *Django* o al código suministrado por vuestros profesores.
- Las vistas (`views.py`) deben implementarse usando clases.

4. Servicios: uso de los cuestionarios

En la sección anterior implementamos la infraestructura necesaria para crear cuestionarios. Ahora vamos a crear parte de los servicios encargados del uso de los cuestionarios por parte de los participantes. Mientras que para la creación de los cuestionarios se necesita un único navegador, para jugar es necesario mantener abiertos dos navegadores. El primero, común a todos los participantes, muestra las preguntas y las puntuaciones mientras que el segundo se usa para suministrar las respuestas. Pasamos a describir las vistas necesarios para mostrar las preguntas y las puntuaciones. Básicamente nos hace falta una vista para cada una de las siguientes acciones:

1. crear el juego (esta vista se creó en el apartado anterior) y esperar a que los participantes se unan al mismo.
2. mostrar una página que avise a los participantes de que el juego va a empezar en breve.
3. mostrar de forma ordenada y consecutiva las preguntas del cuestionario seleccionado.
4. mostrar, tras cada pregunta, la puntuación.

5. al finalizar el cuestionario mostrar un pódium con las mejores puntuaciones.

Los cuatro últimos pasos pueden juntarse e implementarse en una única vista.

Para comenzar un juego sugerimos que añadáis un botón a la página web que muestra los detalles del cuestionario. Al seleccionar este botón se creará un juego (`game`) con `game.state=WAITING` y se mostrará una página que contenga el identificador (`publicId`) del juego que será usado por los participantes para unirse al mismo (Fig. 9). Esta página se refrescará periódicamente actualizando el listado de los alias de los participantes que se hayan conectado (Fig. 10). Además de los alias de los participantes, la página contendrá un botón que permita comenzar el juego mostrando primero una página que avise a los participantes de que el juego va a comenzar (Fig. 11) y después muestre las preguntas (y las respuestas) de forma sucesiva (Fig. 12). Tras mostrar una pregunta se esperan `question.answerTime` segundos y se mostrará de forma automática la respuesta correcta y la puntuación (Fig. 13). La pantalla que muestre la puntuación contendrá un botón que permita pasar a la pregunta siguiente o muestre el pódium con la puntuación de los participantes en caso de no existir más preguntas (Fig. 14). Obviamente no se puede responder a una pregunta una vez que se ha mostrado la respuesta correcta.

Listado de vistas a implementar

`game-create` | `gamecreate/<int:questionnaireid>` | Esta vista ya se describió en la sección anterior y se debe expandir de forma que llame cada 2 segundos a `game-updateparticipant` y actualice el listado de participantes en el juego. En <https://stackoverflow.com/questions/32702758/using-ajax-in-django-to-display-time-of-day-every-second> podéis ver un ejemplo de como se usa Ajax para llamar periódicamente a un URL.

`game-updateparticipant` | `gameUpdateParticipant/<int:publicid>` | `test01_gameUpdateParticipant` | Devuelve el listado de participantes que se hayan unido al juego y que será usado para actualizar el listado de participantes (Fig. 10). Esta vista no recargará la totalidad de la página sino sólo el área destinada a mostrar el listado de participantes. El identificador de juego se obtendrá a partir de una variable de sesión.

game-count-down | gamecountdown/ | test02_gameCountdown | Esta vista gestiona el resto de los requerimientos. Consulta el valor de `game.state`, lo actualiza adecuadamente, crea las variables necesarias y devuelve en cada caso una “template” diferente. De esta forma (a) si el estado inicial de `game.state` es `WAITING` muestra primero un count-down (Fig. 11) seguido de una pregunta (Fig. 12) y actualiza el estado a `ANSWER`, (b) si el estado es `QUESTION` muestra una pregunta (con sus respuestas) (Fig. 12) y actualiza el estado a `ANSWER`, (c) si el estado es `ANSWER` muestra la puntuación (Fig. 13), actualiza la variable `game.questionNo` y actualiza el estado a `LEADERBOARD` si es la última pregunta o a `QUESTION`. (d) Si el estado es `LEADERBOARD` se muestra un podio (Fig. 14). Para implementar esta vista os será útil redefinen los métodos `get_context_data` y `get_template_names` en `gamecountdown`. IMPORTANTE: el test `test02_gameCountdown` asume la existencia de una variable de sesión donde se guarda el estado del juego y comprueba como se va modificando. Este test es muy dependiente de la implementación así podéis modificarlo y crear algo funcionalmente equivalente que se adapte a vuestra implementación.

Desde el punto de vista de los participantes la aplicación gana bastante si se añade sonido a las páginas usadas para jugar. Una forma sencilla de hacerlo es añadiendo en las templates código similar a:

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

Por defecto los navegadores tienen deshabilitada la reproducción de sonido sin que el usuario la consienta expresamente. En <https://support.mozilla.org/en-US/kb/block-autoplay> podéis ver como habilitar esta característica para un servidor en particular.

4.1. *Render.com*

Finalmente, despliega el proyecto en *Render.com* y puebla la base de datos usando el script `populate`. Como se comentó en la práctica 1, en producción (`settings.DEBUG=False`), *Django* no está diseñado para servir los ficheros estáticos sino que asume que algún servidor web se encargará de hacerlo y su única responsabilidad es crear las URLs que apunten a estos datos. A continuación describimos como servir los ficheros estáticos en los distintos escenarios posibles:

Ejecución local y variable `DEBUG` en `settings.py` igual a `TRUE` *Django* servirá los ficheros estáticos sin necesidad de hacer ningún ajuste

Ejecución local y `DEBUG=False` La manera más sencilla de servir el contenido estático será iniciando el servidor con la bandera `-insecure`. Esto es:

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

Ejecución en *Render.com* y `DEBUG=False` Los ficheros estáticos deben formar parte del repositorio de git que se sube a *Render.com* (no se pueden generar después de hacer el `git push` aunque se ejecute después `./manage.py collectstatic`). Además si quieres servir ficheros estáticos binarios (por ejemplo música) has de incorporar el modulo `whitenoise` tal y como se describe en la práctica 1.

4.2. Testing y coverage

Para verificar la correcta implementación de los diferentes modelos y servicios, se proporcionan diversos ficheros de test, los cuales contienen una batería de test (no necesariamente completa) que debe satisfacer tu código. Estos test deben entenderse como requerimientos adicionales al proyecto.

Una vez que hayáis conseguido que los test se ejecuten de forma satisfactoria ejecutad el comando `coverage` para las aplicaciones (`models` y `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
```

mirad la cobertura de los ficheros `models.py` y `views.py`. Si no es del 100 % añadid a los ficheros de test los test que hagan falta para alcanzar este valor. (La solicitud de 100 % de cobertura aplica exclusivamente al código que creéis vosotros. Ignorad el código creado por *Django* o el suministrado por vuestros profesores.)

5. Trabajo a presentar al finalizar la práctica

- Aseguraros que vuestro código satisface todos los test proporcionados. **A menos que se especifique lo contrario, no es admisible que se modifique el código de los test.**
- Implementar todos los test que consideres necesarios para cubrir la funcionalidad desarrollada. Estos test se deben implementar en un fichero llamado `test_additional.py`.
- Incluir en la raíz del proyecto un fichero llamado `coverage.txt` que contenga el resultado de ejecutar el comando `coverage` para todos los test.
- Desplegad y probad la aplicación en *Render.com* en modo de producción (`DEBUG=False`, `SECRET_KEY` y `DATABASE_URL` en una variable de entorno).
- Subir a *Moodle* el fichero obtenido al ejecutar el comando `zip -r ../assign4_final.zip .git` desde la raíz del proyecto. Recordad que hay que añadir y “comitir” los ficheros a git antes de ejecutar el comando. Si queréis comprobar que el contenido del fichero zip es correcto lo podéis hacer ejecutando la orden: `cd ..; unzip assign4_final.zip; git clone . tmpDir; ls tmpDir`.
- Verificar que la variable `ALLOWED_HOSTS` del fichero `settings.py` incluido en la entrega contiene tu dirección de despliegue en *Render.com* (si no aparece corregiremos la práctica como si el proyecto no estuviera desplegado en *Render.com*). Asimismo, comprobar en *Render.com* que tanto el nombre de usuario como la clave del usuario de administración son *alumnodb*.

6. Criterios de evaluación

Nota: A la hora de evaluar esta práctica **NO** se considerará la estética de la misma (la cual será evaluada en la próxima práctica).

Para aprobar con 5 puntos es necesario satisfacer en su totalidad los siguientes criterios:

- Todos los ficheros necesarios para ejecutar la aplicación se han entregado a tiempo.
- El código se ha guardado en un repositorio de git y este repositorio es privado.
- El fichero subido a *Moodle* contiene un repositorio de git.
- El script `populate.py` existe y es funcional.
- La aplicación se puede ejecutar localmente.
- Al ejecutar en local los test el número de fallos no es superior a cuatro y el código que los satisface es funcional.
- No se ha modificado el código de los test.
- La aplicación funciona contra una base de datos creada en `https://neon.tech` e implementada usando *PostgreSQL*.
- La aplicación de administración de base de datos se ha desplegado y está accesible en el servido de *Django* local usando el nombre de usuario/clave *alumnodb*.
- Usando localmente la aplicación de administración es posible crear o borrar objetos pertenecientes a todos los modelos solicitados
- IMPORTANTE: nos hará falta el URI de `https://neon.tech` para evaluar esta práctica por favor incluídlo en un fichero llamado “env” que se encuentre en la raíz del proyecto.

Cumpliendo los siguientes criterios se puede optar a una nota máxima de 6.9:

- Los criterios enunciados en el párrafo anterior se satisfacen en su totalidad.
- La aplicación está desplegada en *Render.com*. En el fichero `settings.py` se encuentra añadida la dirección de *Render.com* a la variable `ALLOWED_HOSTS`. Además de estar desplegada, la aplicación funciona correctamente en *Render.com*.
- El código subido a *Moodle* es idéntico al código desplegado en *Render.com*.
- La aplicación de administración de base de datos se ha desplegado y está accesible en *Render.com* usando el nombre de usuario/clave *alumnodb*.
- Usando en *Render.com* la aplicación de administración es posible crear o borrar objetos pertenecientes a todos los modelos solicitados.
- Todas las “templates” heredan de `base.html`.
- IMPORTANTE: Si añades a `ALLOWED_HOSTS` la dirección en la cual se despliega la aplicación en *Render.com* usando una variable de entorno, añade el valor de esta variable al fichero “env”

Cumpliendo los siguientes criterios se puede optar a una nota máxima de 7.9:

- Los criterios enunciados en el párrafo anterior se satisfacen en su totalidad.
- *Render.com* está desplegado en modo de producción. `DEBUG=False` y `SECRET_KEY` no está almacenada en `settings.py`.
- Todas las vistas (métodos/clases implementados en `views.py`) heredan de clases tipo.
- Al ejecutarse los test el número de fallos no es superior a dos y el código que los satisface es funcional.
- El código es legible, eficiente, está bien estructurado y comentado.
- Se utilizan las herramientas que proporciona el framework.
- Sirva como ejemplo de los puntos anteriores:

- Todos los formularios que involucren un “modelo” se han creado de forma que hereden directa o indirectamente de la clase `forms.Form`.
 - Las búsquedas las realiza la base de datos no se acceden a todos los elementos de una tabla y se busca en las funciones definidas en `views.py`.
 - Los errores se procesan adecuadamente y se devuelven mensajes de error comprensibles.
 - El código presenta un estilo consistente y las funciones están comentadas incluyendo su autor. Nota: el autor de una función debe ser único.
 - Se es coherente con los criterios de estilos marcados por *Flake8*. *Flake8* no devuelve ningún error al ejecutarse sobre las líneas de código programadas por el estudiante.
- Resulta imposible suplantar a un usuario (o participante) sin conocer su nombre de usuario y clave (o `game.publicId`). Por ejemplo: (a) no se puede modificar un cuestionario/pregunta/respuesta sin hacer login previamente accediendo directamente al URL correspondiente, (b) No se puede crear respuestas `guess` sin conocer el `game.publicId`, etc.

Cumpliendo los siguientes criterios se puede optar a una nota máxima de 8.9:

- Los criterios enunciados en el párrafo anterior se satisfacen en su totalidad.
- Todos los test y todas las pruebas ejecutadas dan resultados satisfactorios.
- Si reducimos el tamaño de la ventana del navegador o utilizamos el zoom, todos los elementos de la página siguen resultando accesibles y no se pierde funcionalidad.

Para optar a la nota máxima se debe cumplir lo siguientes criterios:

- Los criterios enunciados en el párrafo anterior se satisfacen en su totalidad.
- La cobertura para los ficheros que contienen los modelos, las vistas y los formularios es superior al 99 %.

- Se ha implementado el sonido en las páginas que ven los participantes.

Nota: Entrega fuera de plazo → substraer un punto por cada día (o fracción) de retraso en la entrega.

Nota: El código usado en la corrección de la práctica será el entregado en *Moodle*. Bajo ningún concepto se usará el código existente en *Render.com*, *Github* o cualquier otro repositorio.

A. Imágenes

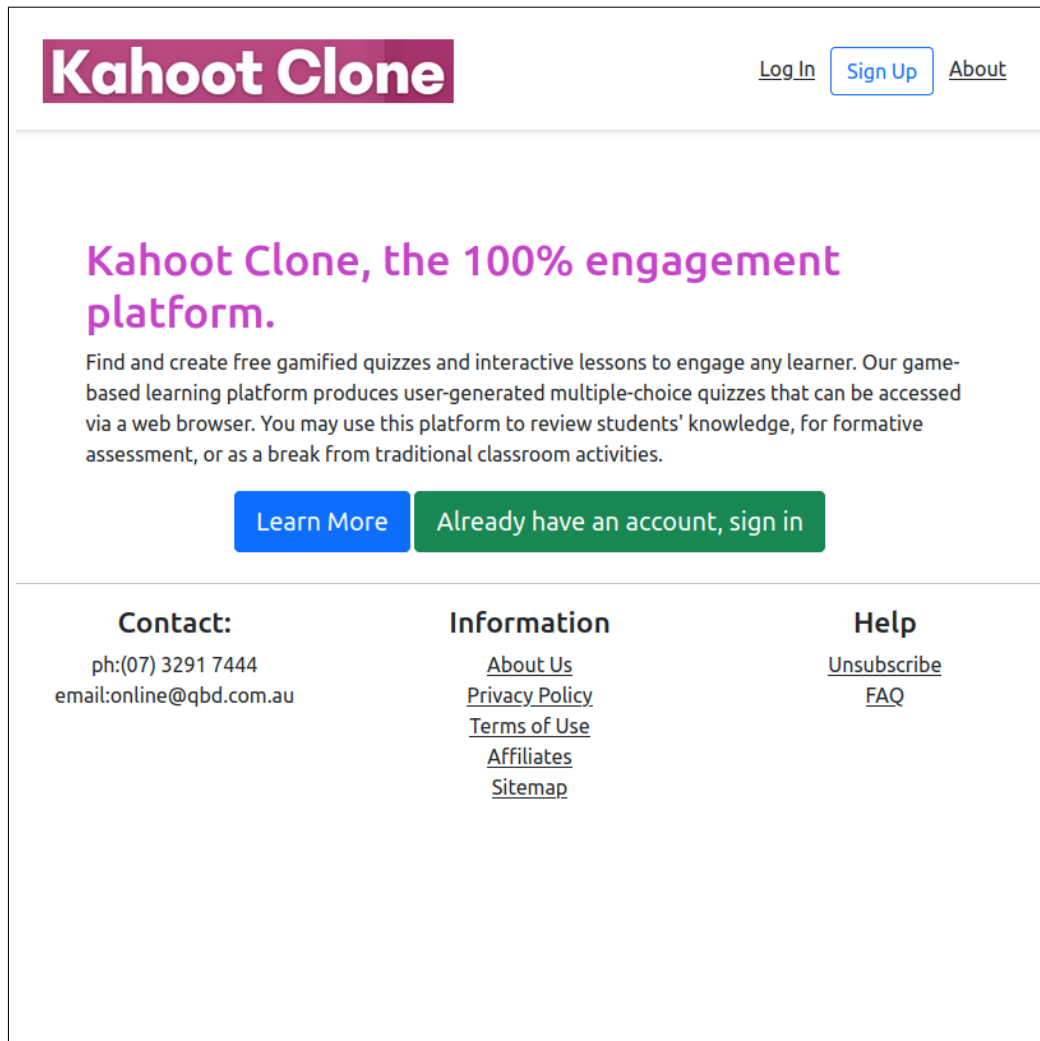


Figura 1: Ejemplo de “homepage”, el usuario no está conectado.

Kahoot Clone

[Log Out](#) [About](#)

Kahoot Clone, the 100% engagement platform.

Find and create free gamified quizzes and interactive lessons to engage any learner. Our game-based learning platform produces user-generated multiple-choice quizzes that can be accessed via a web browser. You may use this platform to review students' knowledge, for formative assessment, or as a break from traditional classroom activities.

[Add New Questionnaire](#) [List All Your Questionnaires](#)

Your last questionnaires

- [None power admit red car dream better](#)
- [Everything need collection activity degree information share](#)
- [Compare total record only fly](#)
- [Source pull wish pay soon](#)
- [Next difference police](#)

Contact:
ph:(07) 3291 7444
email:online@qbd.com.au

Information
[About Us](#)
[Privacy Policy](#)
[Terms of Use](#)
[Affiliates](#)
[Sitemap](#)

Help
[Unsubscribe](#)
[FAQ](#)

Figura 2: Ejemplo de “homepage”, el usuario está conectado.

Kahoot Clone

[Log Out](#) [About](#)

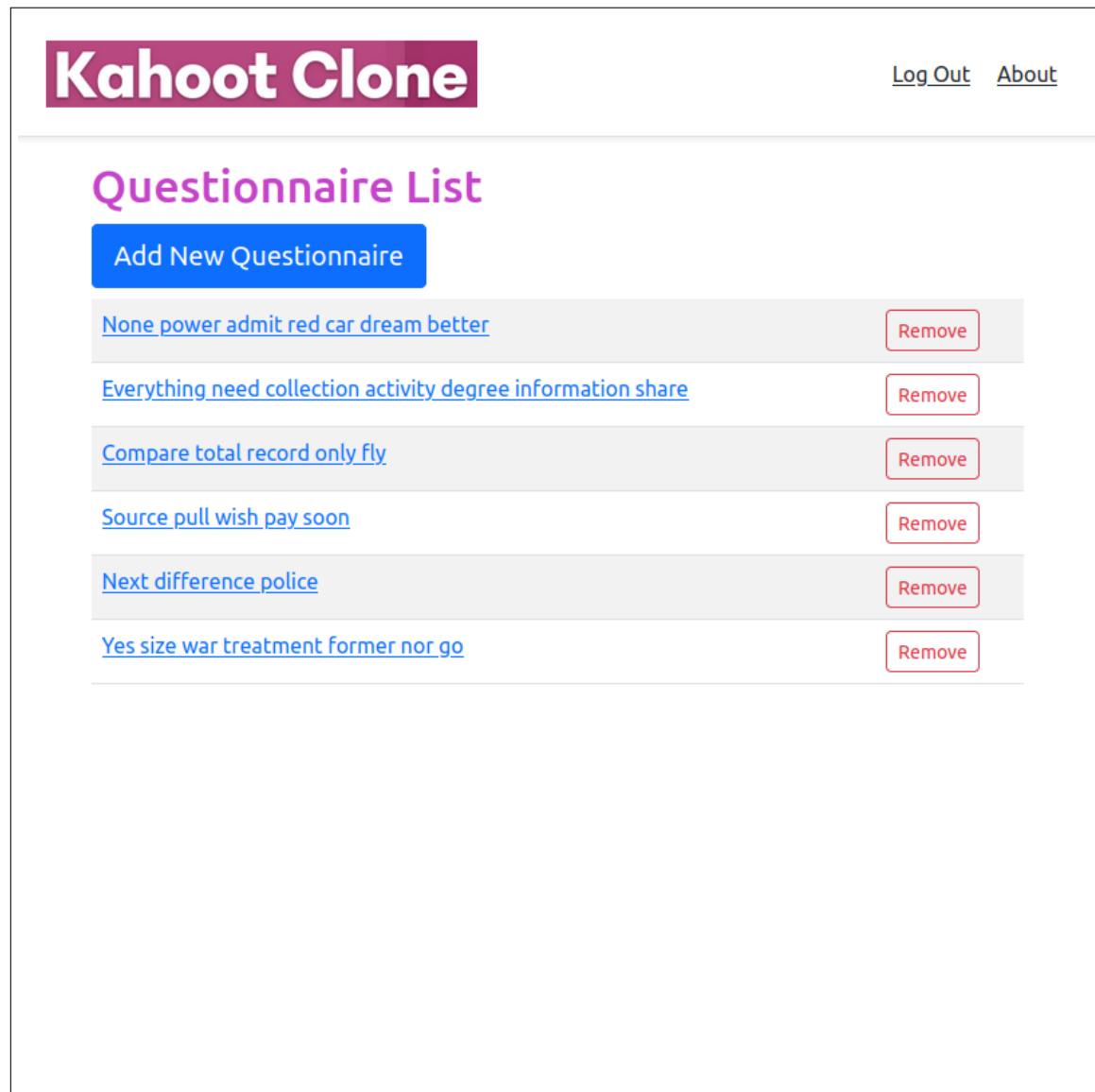
Questionnaire Detail

None power admit red car dream better: [Edit title](#) [Play Game](#)

question	No answers	
7 * 3 =	4	Remove
3 * 1 =	4	Remove
1 * 0 =	4	Remove
10 * 4 =	4	Remove
2 * 3 =	4	Remove
6 * 1 =	4	Remove

[Add New Question](#) [Back to questionnaire list](#)

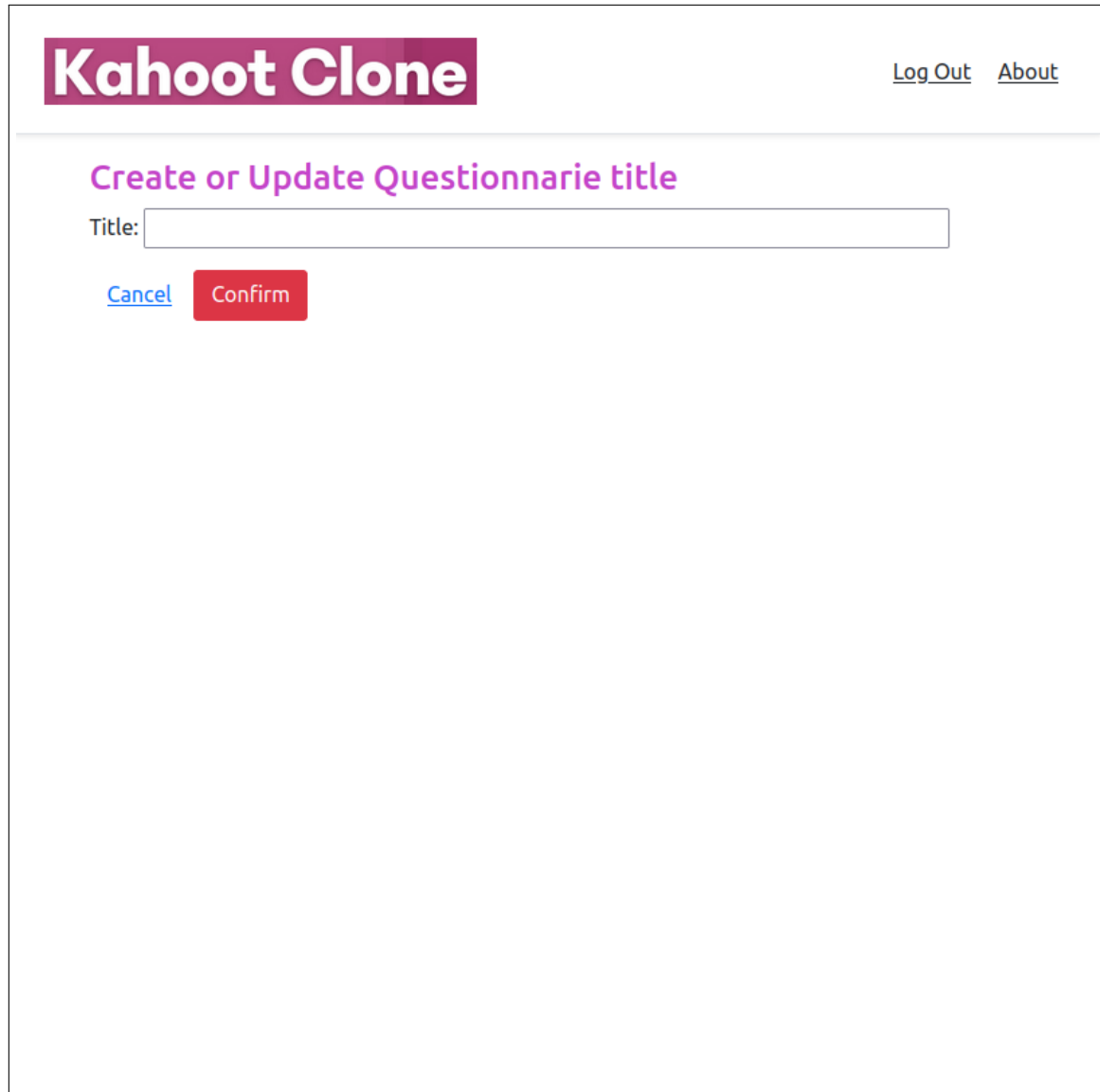
Figura 3: Página mostrando las preguntas pertenecientes a un cuestionario.



The screenshot displays the 'Kahoot Clone' web application interface. At the top, there is a header with the application name 'Kahoot Clone' in a large, bold, purple font on the left, and two links, 'Log Out' and 'About', on the right. Below the header, the main content area is titled 'Questionnaire List' in a purple font. Under this title is a blue button labeled 'Add New Questionnaire'. Below the button is a list of six questionnaires, each represented by a horizontal row. Each row contains a questionnaire title (e.g., 'None power admit red car dream better') and a red 'Remove' button on the right. The rows are separated by thin horizontal lines.

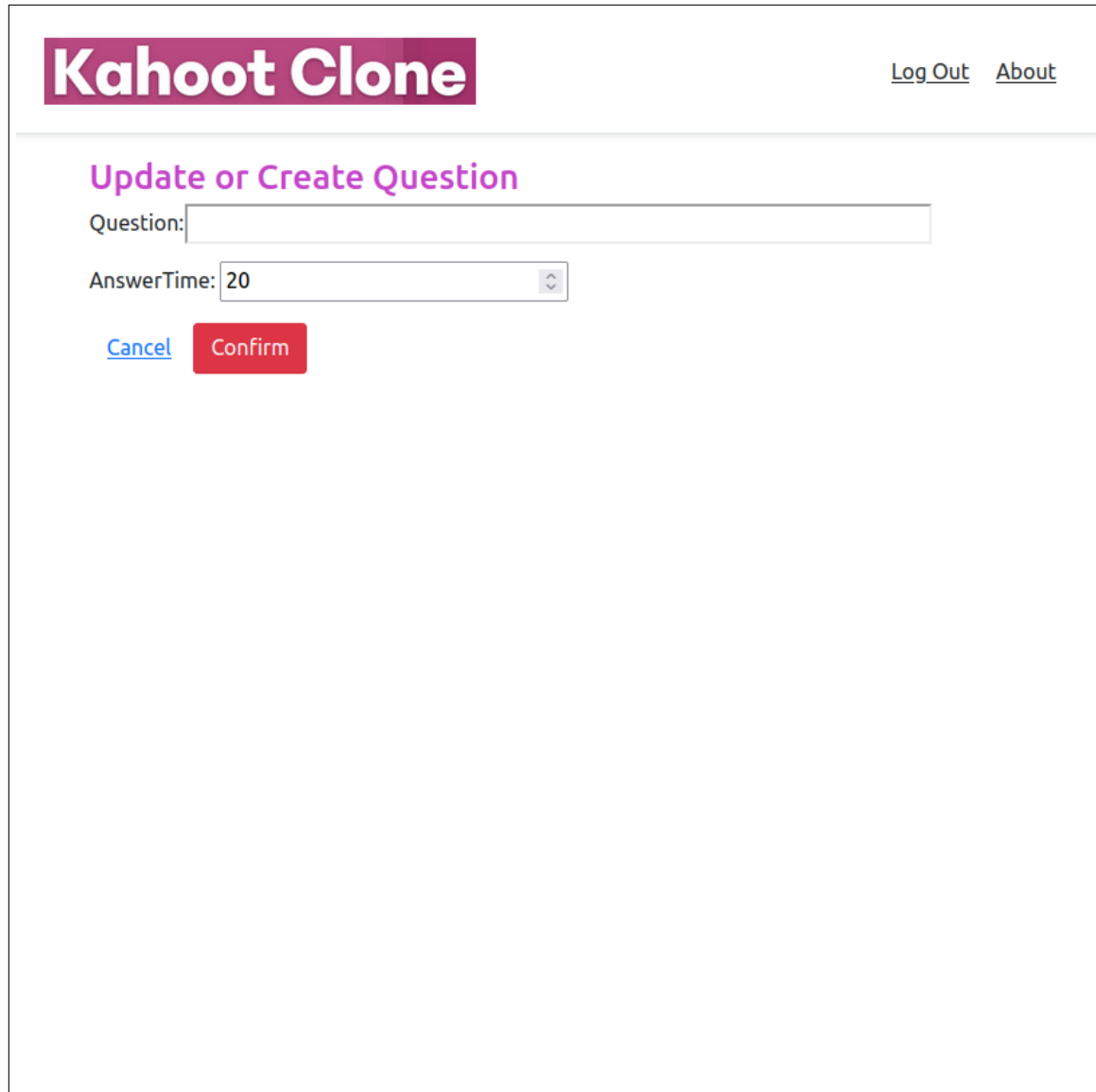
Questionnaire Title	Action
None power admit red car dream better	Remove
Everything need collection activity degree information share	Remove
Compare total record only fly	Remove
Source pull wish pay soon	Remove
Next difference police	Remove
Yes size war treatment former nor go	Remove

Figura 4: Página mostrando un listado con todos los cuestionarios pertenecientes al usuario conectado.



The screenshot displays the 'Kahoot Clone' web application interface. At the top left, the title 'Kahoot Clone' is shown in a large, bold, white font on a dark red background. To the right of the title, there are two links: 'Log Out' and 'About', both underlined. Below the header, the main content area has a light gray background. It features a heading 'Create or Update Questionnaire title' in a purple font. Underneath this heading is a form with a label 'Title:' followed by a text input field. Below the input field, there are two buttons: a blue 'Cancel' link and a red 'Confirm' button.

Figura 5: Página usada para crear o modificar un cuestionario.



The screenshot shows a web interface for a 'Kahoot Clone'. At the top left is the title 'Kahoot Clone' in a large, bold, white font on a dark red background. To the right of the title are two links: 'Log Out' and 'About', both underlined. Below the title bar is a section titled 'Update or Create Question' in a purple font. This section contains two input fields: 'Question:' followed by a long text input box, and 'AnswerTime:' followed by a dropdown menu currently showing the value '20'. At the bottom of this section are two buttons: a blue 'Cancel' link and a red 'Confirm' button.

Figura 6: Página usada para crear o modificar una pregunta.

Kahoot Clone

[Log Out](#) [About](#)

Question Detail

7 * 3 = (20): [Edit question](#)

answer	correct	
21	True	Remove Edit
1	False	Remove Edit
74	False	Remove Edit
10	False	Remove Edit

[Back to questionnaire](#)

Figura 7: Página mostrando las respuestas pertenecientes a una pregunta.



The image shows a web interface for a 'Kahoot Clone'. At the top left, the title 'Kahoot Clone' is displayed in white text on a dark red rectangular background. To the right of the title, there are two links: 'Log Out' and 'About', both underlined. Below the header, the main content area has a title 'Update or Create Answer' in a purple font. Under this title, there is a label 'Answer:' followed by a long, empty text input field. Below the input field, there is a label 'Correct:' followed by an unchecked checkbox. At the bottom left of the form, there is a blue text link 'Cancel' and a red rectangular button with the word 'Confirm' in white text.

Figura 8: Página usada para crear o modificar una respuesta.

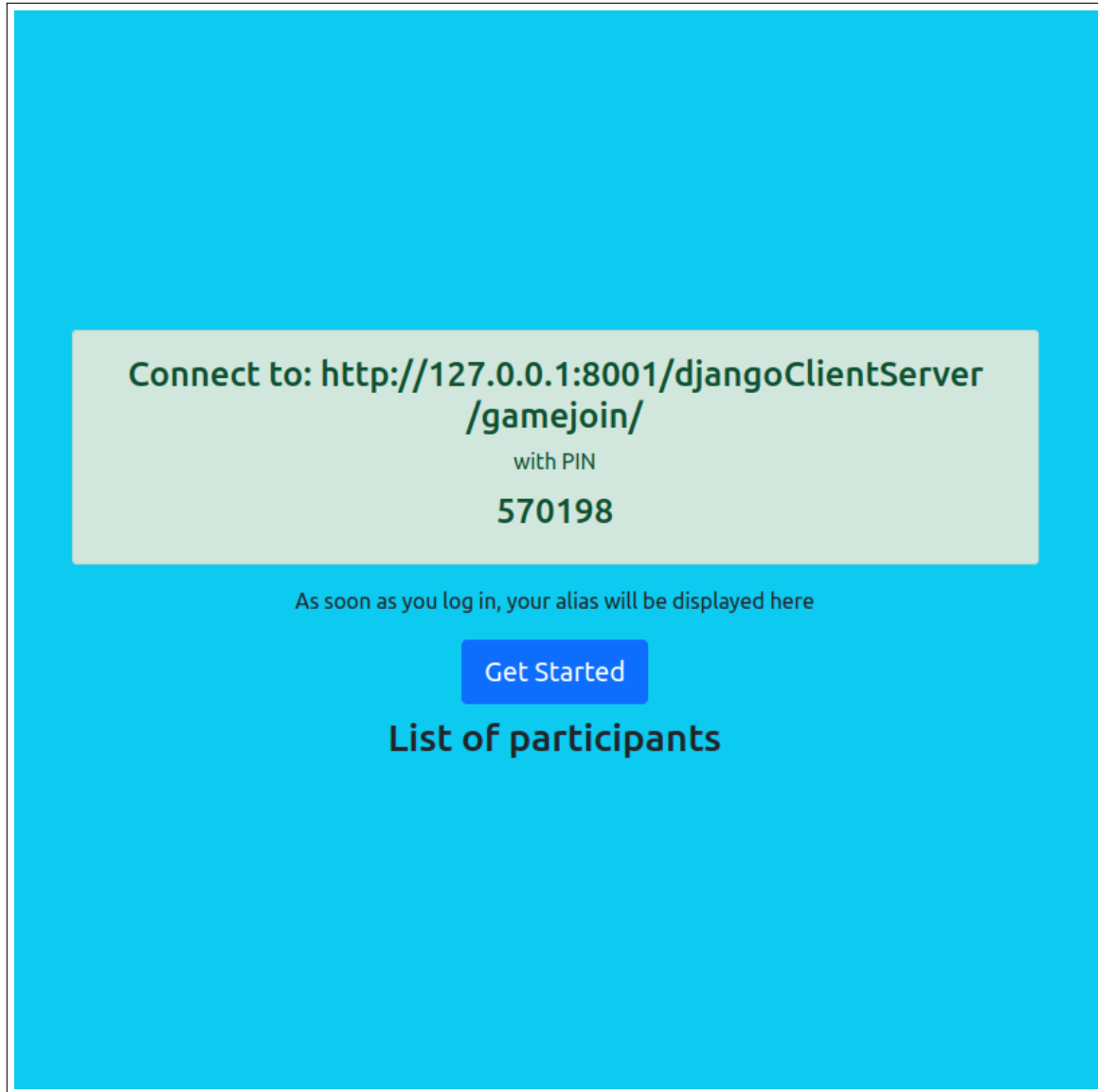


Figura 9: Resultado de ejecutar la vista que crea un juego (game).

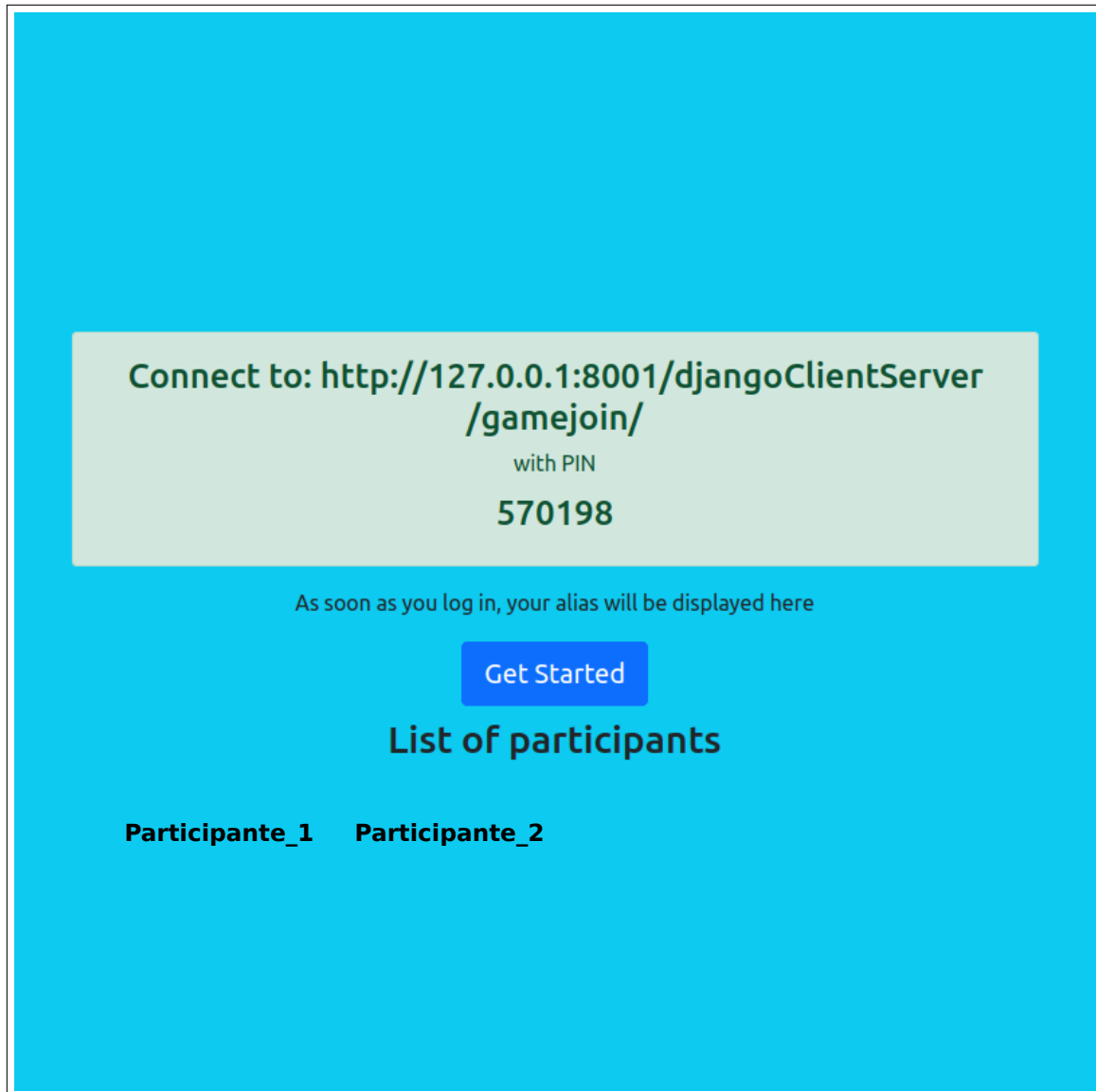


Figura 10: Resultado de ejecutar la vista que actualiza el listado de participantes.

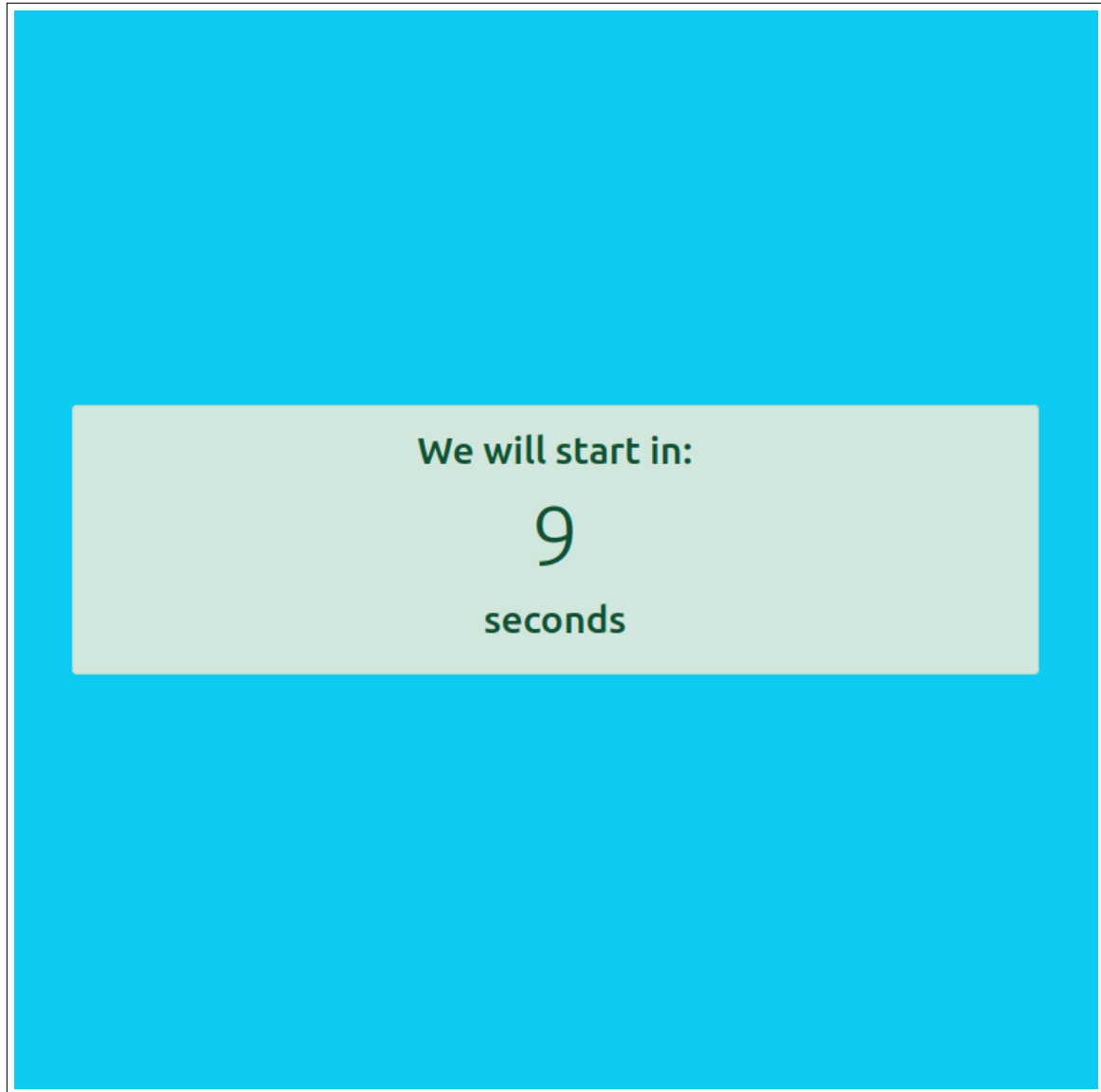


Figura 11: Página que informa a los participantes de que el juego empezará en breve.

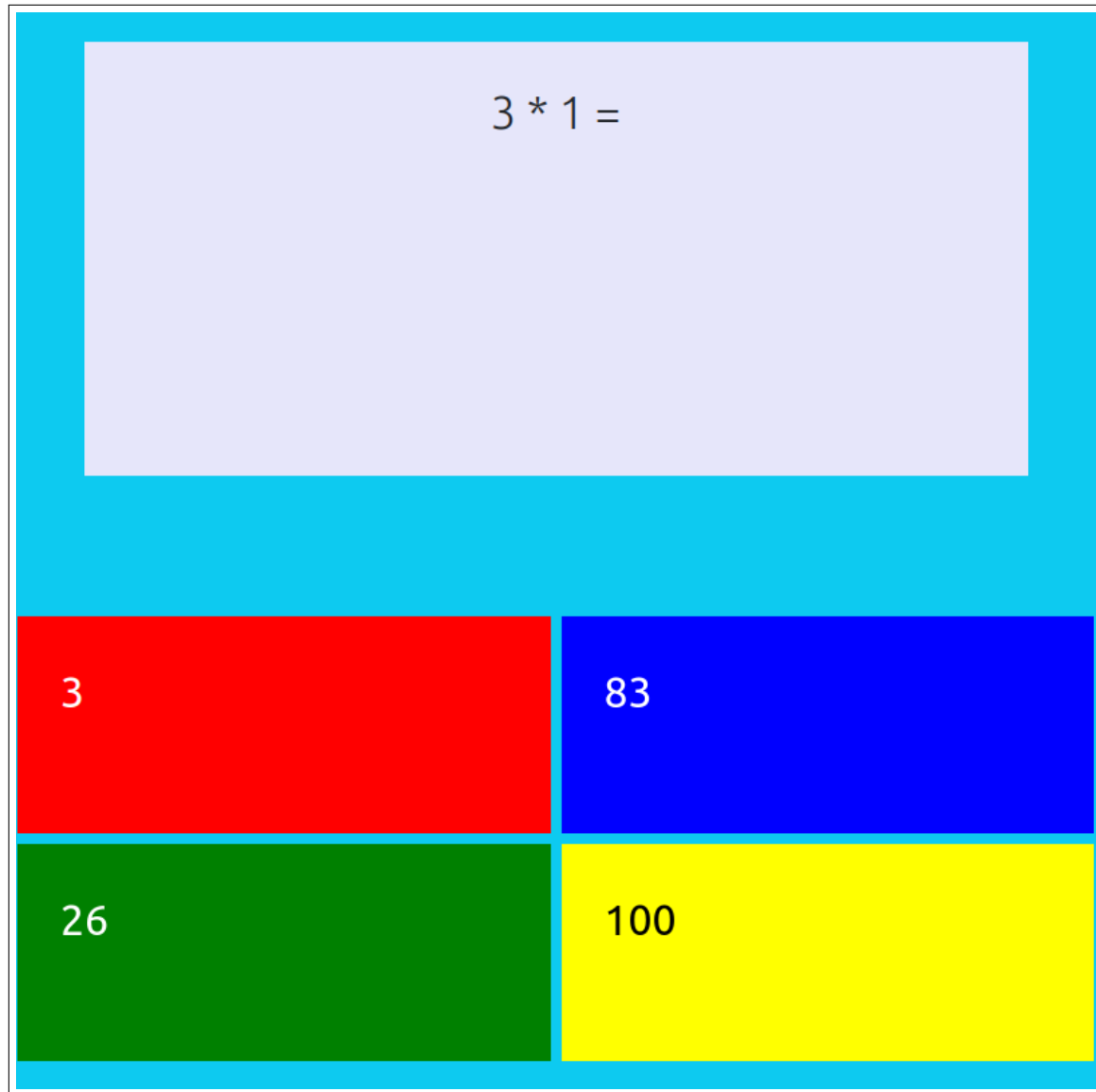


Figura 12: Página mostrando una pregunta y sus respuestas a los participantes.

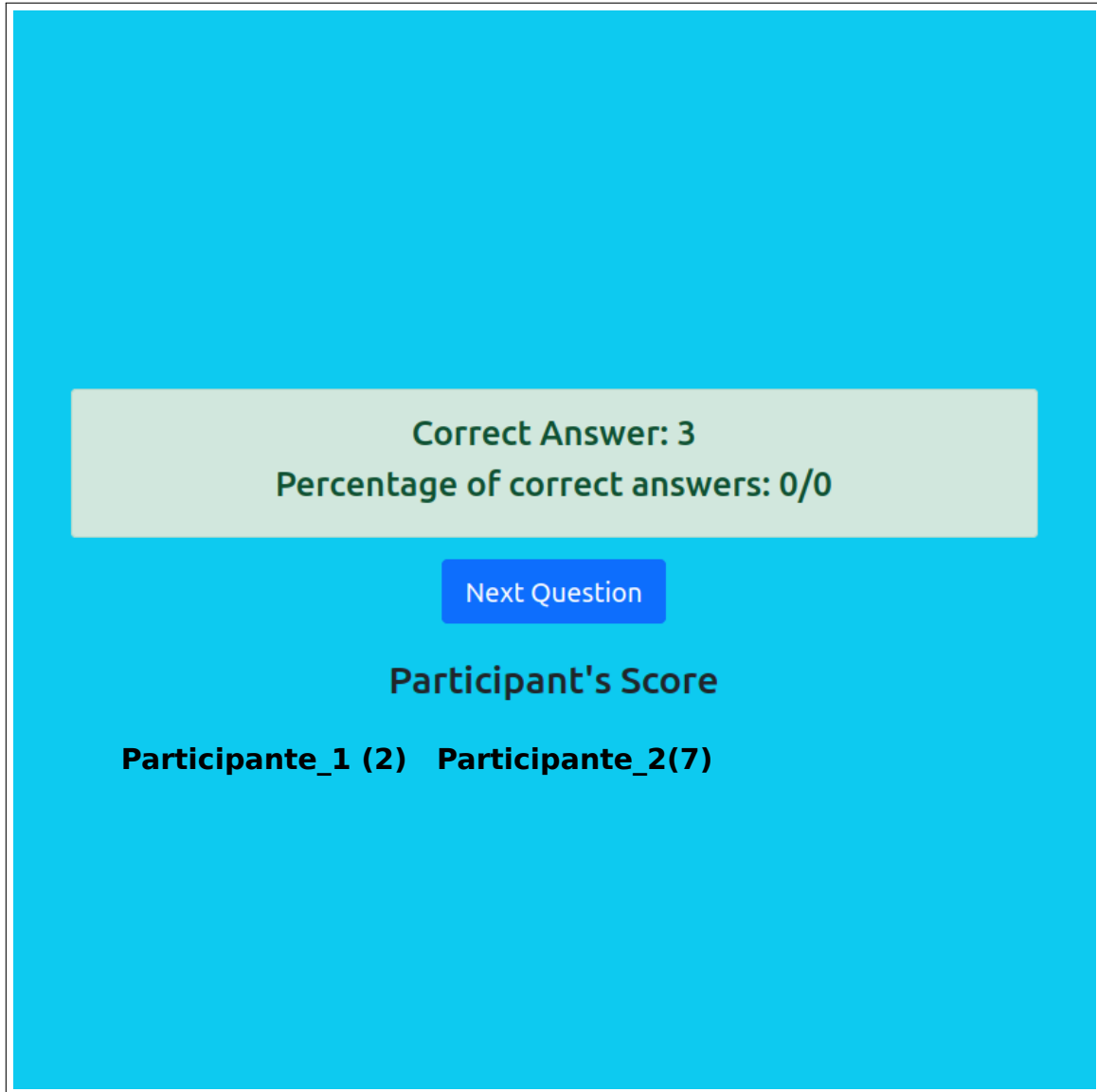


Figura 13: Página mostrando el resultado de una pregunta y la puntuación de los participantes.

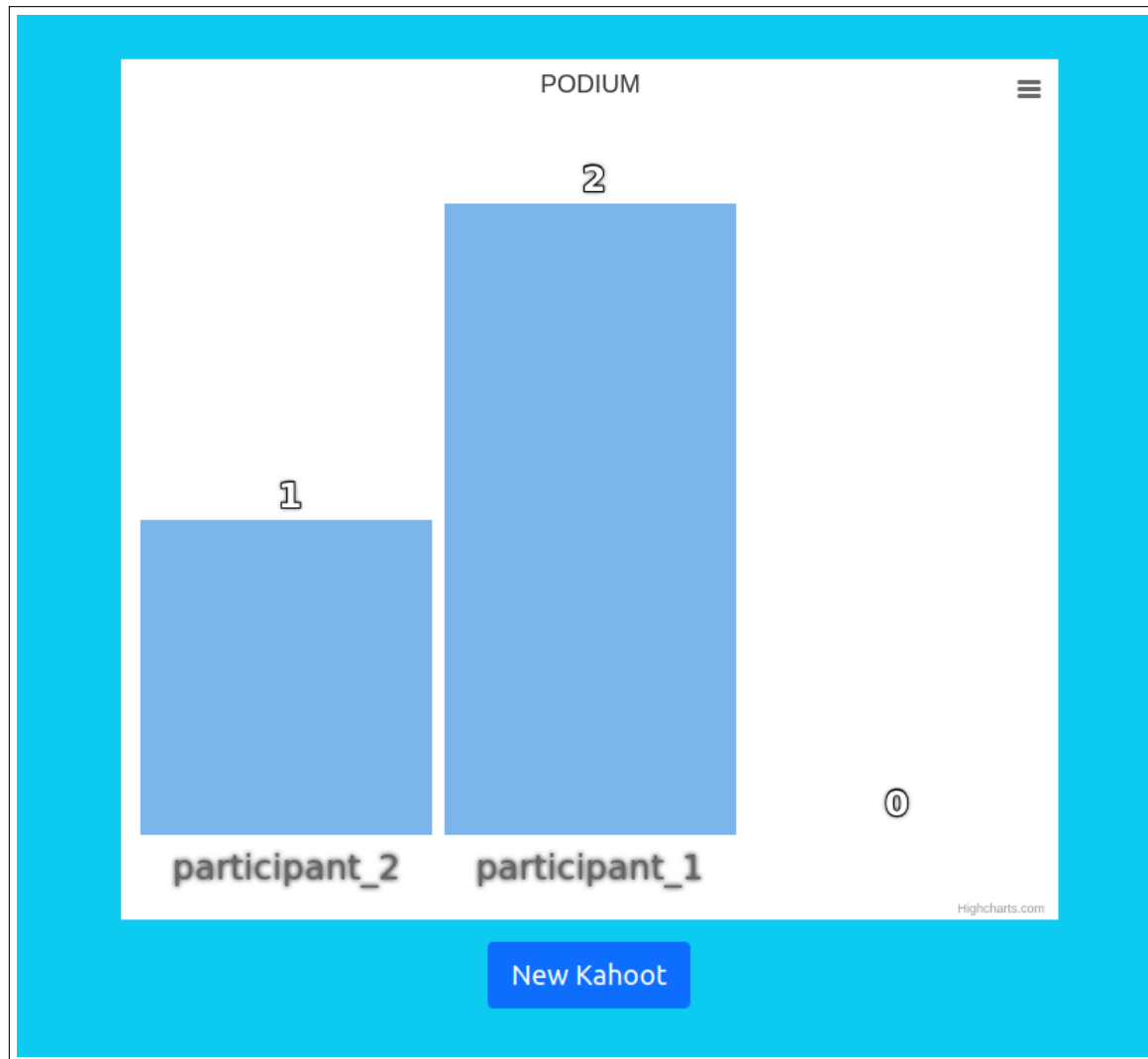


Figura 14: Página mostrando la clasificación final (podio).