
Proyecto de procesamiento de lenguaje natural (5.0v)

Definición de TFG para procesamiento de voz en Glifing S.L.

Jesús Manuel Nieto Carracedo

20 de abril de 2024



Índice

1. Descripción general del proyecto	3
2. Arquitectura del sistema	3
3. Modelo de datos	4
3.1 Usuarios, roles, pauta	4
3.1.1 Almacenamiento de resultados.	5
3.2 Sesiones	6
3.3 Activities	7
3.4 Items	7
3.5 Status	8
3.6 Task	8
3.6.1 Descripción del fichero de instalación	8
3.6.2 Instalación de una tarea.	10
3.6.3 Ejecución de código de instalación (Ayuda)	11
4. Componentes	13
4.1 Ejemplo de una vista tipo índice(index)	14
4.2 Ejemplo de una vista tipo cabecera-líneas	15
5. Vistas	15
5.1 Usuarios	15
5.1.1 Index principal de usuarios	15
5.1.2 Editor de usuario	16
5.1.3 Editor de pauta	16
5.1.4 Ejecución de sesiones	17
5.1.5 Vista de player	17
5.2 Sesiones	20
5.2.1 Index principal de sesiones	20
5.2.2 Editor de sesiones	20
5.3 Actividades	20
5.3.1 Index principal de actividades	20
5.3.2 Editor de actividades	20
5.4 Items	20
5.4.1 Index principal de items	21
5.4.2 Editor de items	21

5.5 Status	21
5.5.1 Index principal de status	21
5.5.2 Editor de estados	21
5.6 Task	21
5.6.1 Index principal de tareas	21
5.6.2 Editor de tareas	21
6 Inter-conexión de capas: Api	22
6.1 Operaciones básicas de la Api	23
6.2 Operaciones especiales de la Api	23
7. Recomendaciones y buenas prácticas	23
7.1 Repositorios	23
7.1.1 Buenas Prácticas con Git, uso de .gitignore, proceso de instalación	23
7.1.2 ¿ Cuando y como implementar esto?	25
7.2 Variables, paths, strings, etc...	25
7.2.1 Identificar las cadenas	25
7.2.2 Idiomas	26

1. Descripción general del proyecto

El objetivo general del proyecto es crear una plataforma que nos permita crear y editar un banco de pruebas de sesiones, con actividades de lectura, para obtener una serie de estadísticos y métricas que puedan ser analizados por plataformas externas.

Dado el tiempo asignado para la resolución de este proyecto, y sus fines formativos, nos centraremos en las partes de valor, es decir, en las capas de la aplicación que gestionan los datos relativos a la lectura, por ese motivo, no se pondrá excesivo foco en cuestiones como la seguridad, concurrencia en la edición, etc.. por ello se simplificarán las partes comunes y superfluas, que comúnmente suelen resolverse a través de rutinas, métodos, o funciones estandarizadas en los distintos **frameworks** utilizados.

Haremos uso de un **modelo de desarrollo incremental**, y para ello usaremos las diferentes tecnologías que se indican a continuación.

2. Arquitectura del sistema

Se generará un sistema basado en 3 grandes capas, basadas en el modelo **MVC**:

- **Persistencia:** Estará basada en una base de datos documental.
 - Se implementará con MongoDB
- **Controlador:** Para intercambiar información entre persistencia y las vistas del sistema, usaremos una **api-rest**.
 - Se implementará a través del framework backend django-rest-franmework el cual usará como lenguaje de servidor **python 3.x**.
- **Vistas:** Las vistas, tanto para **backend** como para **frontend** implementarán una programación orientada a componentes, implementando el modelo **MVVM**:
 - Se implementará a través del framework frontend angular, donde usaremos **html5** y **typescript**.
 - Para la maquetación usaremos un framework CSS llamado bootstrap
- **Sistemas:** La pila estará compuesta por:
 - **Servidor:** Usaremos linux en una versión basada en **debian**, preferiblemente **Ubuntu**.
 - **Servicio:** Usaremos como servidor **Apache2**, el cual usará el módulo **wsgi** para poder ejecutar código **python**

- **Virtualización:** Usaremos virtualización basada en docker que tendrá el siguiente despliegue:
- **Código:** Se desplegará en el todo el código fuente, pero nada de persistencia, este modelo de arquitectura persigue ser altamente escalable tanto horizontal, como verticalmente.
- **Documentación:** La misma se hará a través de estos sistemas:
 - **Código fuente:** A través de **git**, como plataforma de intercambio de código fuente usaremos github.com
 - **Api:** A través de swagger
 - **Proyecto-cliente:** A través de trello

3. Modelo de datos

3.1 Usuarios, roles, pauta

Los usuarios podrán disponer de unos roles, y pueden tener los 3 tipos, o solo alguno de ellos, además en caso de tener el rol de entrenado, tendremos que recoger algún campo adicional. Solo podrá tener el usuario un rol activo al tiempo, esto se hace para facilitar la carga de vistas. Es decir, si un usuario tiene el rol de editor y entrenado, y accede al sistema, tendrá que poder determinar el sistema que rol está usando en ese momento para poder mostrar su vista adecuada.

- Tipos y acciones, será una jerarquía donde el primero podrá hacer todo lo que hacen los siguientes más sus acciones:
 - **1. Administrador:**
 - * Administrar totalmente al resto de usuarios.
 - * Acceder y revisar configuraciones en la plataforma.
 - * Acceso a resultados de usuarios.
 - **2. Editor:**
 - * Administrar totalmente los paquetes, sesiones y actividades de la plataforma.
 - * Asignar carga de trabajo a los entrenados
 - **3. Entrenado:**
 - * Ejecutar las cargas de trabajo que tenga asignadas, en el orden indicado en una lista.
- Información que necesitamos recoger de cada uno de nuestros usuarios:
 - **Comunes:**
 - * **identificador:** id único. (Usar formato MongoDB)

- * **username**: nombre de usuario, llevará una máscara de entrada que solo permita el formato nombre.apellido, en el caso de repetirse (existir algún otro en el sistema), se tendrá que añadir un número secuencial. El campo será una cadena.
 - * **username_glifing**: será el nombre de usuario que correlaciona con su usuario Glifing, este campo será del mismo tipo que el **username**, pero puede encontrarse vacío, si el usuario no tiene correlación alguna.
 - * **first_name**: nombre, cadena.
 - * **last_name**: apellidos, cadena.
 - * **email**: correo electrónico, se deberá verificar que se ha introducido un email correcto, cadena.
 - * **password**: clave de acceso, usaremos un algoritmo de cifrado que provea el framework Django.
 - * **avatar**: será una imagen asociada al entrenado. Se almacenará en la base de datos.
- **Entrenado:**
- * **cursos**: será una lista de tipo “curso” para los cuales queremos dar acceso a entrenamiento.
- **curso**: - **id**: identificador del curso. - **entrenamientos**: lista de sesiones de entrenamiento de tipo “entrenamiento”.
- **entrenamientos**:
- * **identificador**: id único. (Usar formato MongoDB)
 - * **date**: fecha en la cual se tiene que realizar la sesión.
 - * **session_id**: identificador **mongodb** de la colección de sesiones.
 - * **results**: Lista de resultados, de tipo “resultado”, de la realización de la sesión de entrenamiento.
- **results**: Se genera una lista de resultados de cada actividad. - **identificador**: id único. (Usar formato MongoDB) - **activity_id**: se usará el identificador de la actividad. - **datetime**: fecha y hora de cuando se han enviado los audios. - **task**: se almacena el diccionario **json** obtenido al ejecutar la tarea asociada a la actividad.

3.1.1 Almacenamiento de resultados.

Para almacenar los resultados, generaremos las siguientes acciones:

- **Almacenamiento de audios**: En nuestro directorio de persistencia de resultados, crearemos una carpeta llamada **results**, en el primer nivel de la jerarquía, tendremos el **identificador de usuario de mongodb**, y dentro de esta carpeta, el **identificador de entrenamiento**, dentro de

esta carpeta, se almacenarán los audios con un nombre que será el identificador del ítem al que hacen referencia, de la actividad realizada.

```
1 results/  
2 <IDENTIFICADOR_USUARIO>/  
3   <IDENTIFICADOR_ENTRENAMIENTO>/  
4     <IDENTIFICADOR_RESULT1>/  
5       <identificador_item1>.wav  
6       <identificador_item2>.wav  
7       <identificador_item3>.wav  
8       <identificador_item4>.wav  
9     <IDENTIFICADOR_RESULT2>/  
10      <identificador_item1>.wav  
11      <identificador_item2>.wav  
12      <identificador_item3>.wav  
13      <identificador_item4>.wav
```

- **Almacenamiento de resultados:** Queda explicado en el apartado anterior.

3.2 Sesiones

Los editores y administradores, tienen que poder editar **sesiones** de trabajo, para ello es necesario disponer de una colección de documentos llamados sesiones. Las sesiones serán el primer nivel de agrupamiento, una sesión deberá tener una lista de actividades.

Las listas de **actividades**, son los distintos ejercicios que tienen que realizar los entrenados. Un ejemplo puede ser una práctica de universidad, la cual contendrá sub-apartados, o un examen que contendrá diferentes preguntas, pues bien a esto lo denominaremos actividades. Las actividades a su vez contendrán elementos que usaremos para configurar la actividad, por tanto dentro de una actividad, habrá una lista de elementos que llamaremos **ítems**.

Aunque los ítems, podrían ser diferentes elementos multimedia [“texto”, “audio”, “vídeo”, “imagen”], el proyecto solo soportará texto, para simplificarlo, por tanto no será un requerimiento mínimo del proyecto, implementar soporte nada más que para texto, ya que esto implicaría tener que generar un módulo para la gestión de inventario de ficheros multimedia asociados.

- Información que necesitamos recoger para las sesiones:
 - **identificador:** id único. (Usar formato MongoDB)
 - **name:** nombre de la sesión, tipo cadena.
 - **description:** descripción de la sesión, tipo cadena.
 - **status_id:** debe reflejar un estado de la colección de estados, es decisión de diseño crear una colección llamada estatus, o bien indicarlo en este campo. Buscar la definición en el apartado correspondiente. A tener en cuenta, una sesión en estado **En pruebas** no se podrá

utilizar desde la vista de **player** para garantizar que un entrenado juegue una versión no estable. Y del mismo modo una sesión **activa** tiene que marcarse como “en pruebas** para que pueda ser editable.

- **activities**: Lista de tipo actividades.
- **createdUser_id**: Almacenar el id del usuario que creó la sesión.
- **modifiedUser_id**: Almacenar el id del usuario que modificó la sesión.
- **created**: Almacenar la fecha y hora en la cual se creó la sesión.
- **modified**: Almacenar la fecha y hora en la cual se modificó la sesión.

3.3 Activities

Los documentos de tipo **activities** se encuentran en una lista de actividades en cada sesión. Guardarán la información de configuración que deberá cargar el player, para presentar y coordinar el entrenamiento de cada entrenado.

Vamos a distinguir **2 tipos de actividades**: **-1.- Texto completo**: Se leerá un texto completo. El texto se almacenará como un solo item, esto facilita almacenar el formato. Para simplificar el desarrollo, solo se podrá almacenar un item por actividad para actividades de texto completo. **-2.- Palabras sueltas**: Se mostrarán palabras, sílabas, no palabras, etc... para leer de forma individual, cada item será una palabra para leer.

- Información necesaria que necesitamos recoger para las actividades:
 - **identificador**: id único. (Usar formato MongoDB)
 - **name**: nombre de la actividad, tipo cadena.
 - **description**: descripción **corta** de la sesión, tipo cadena.
 - **instruction**: las instrucciones de que se requiere hacer en la tarea, la diferencia con el campo anterior está en que la descripción, será un resumen corto, y la instrucción se podrá editar como texto con formato, por ello será de tipo cadena.
 - **orderInSession**: el orden en el cual se ejecutará dentro de la sesión.
 - **task_id** : el id de la colección de **task** tareas de análisis sobre los audios de resultado. Aunque se detalla en sección a parte, indica que método, codificado en un fichero **python** a parte que se ejecutará en el sistema sobre los resultados de la actividad.

3.4 Items

Los **items** son los elementos de configuración que irán en una lista dentro de las distintas actividades.

Para simplificar el proyecto, solo vamos a tener items de texto, sería deseable poder añadir elementos multimedia, tales como fotos, audio, o vídeo, pero quedan fuera de las necesidades de este TFG.

Sería deseable que los items de texto, puedan almacenar formato, como colores, negrita, etc... para ello se ha adjuntado librería javascript que facilita esto. Pero de la misma forma que lo anterior, será criterio deseable.

Los items serán una lista dentro de la actividad de cada sesión.

- Información necesaria que necesitamos recoger para los items:
 - **identificador**: id único. (Usar formato MongoDB)
 - **orderInActivity**: Indica el orden en el que se muestran dentro de la actividad.
 - **value**: El valor del item, en actividades de lectura será un texto para leer, y en actividades de tipo palabra, será una palabra.

3.5 Status

Colección de estados posibles para un documento, pudiendo ser:

- **Activo**: El documento que contenga este estado no se podrá utilizar para entrenar, o de forma definitiva. Un documento “activo” solo se podrá usar en modo lectura, nunca en modo escritura, habrá que cambiar primero el estado **En pruebas**. Usaremos este sistema para garantizar que no hay un usuario entrenando por ejemplo en una sesión, cuando alguien está editando la misma.
- **En pruebas**: El documento solo se podrá editar, estará en modo lectura/escritura.

3.6 Task

Las tareas son los ficheros importables para realizar las operaciones sobre los resultados. La idea por tanto de estas tareas, consiste en que las mismas puedan funcionar como programas externos a este desarrollo.

3.6.1 Descripción del fichero de instalación

Se importarán a través de un **fichero .zip**, que contendrá una estructura fija con varios ficheros:

- **info.json**: Contendrá un diccionario **json**, donde tenemos que guardar los siguiente campos:
 - Autor: Nombre del autor del desarrollo.
 - Company: Empresa que ha desarrollado el procedimiento.
 - Name: Nombre de la tarea, este nombre deberá aparecer en el desplegable de la actividad cuando se selecciona el tipo de operación a realizar sobre los datos de audio.

- **Description:** Descripción corta de la tarea, por ejemplo, si esta tarea solo va a medir la velocidad de lectura, y la media de palabras por minuto, es aquí donde indicará esta información. Sería deseable, que esta información esté escrita en **markdown**
- **Parameters:** Descripción de los parámetros de entrada, tipos, que la tarea espera recibir. Por ejemplo, si espera recibir un solo audio, indicará que espera un fichero de audio. Para simplificar el desarrollo, en lugar de poder indicar 0 ó N parámetros, siempre se enviará una lista, para que el procedimiento que lea los parámetros, lea una lista vacía, en caso de 0 parámetros, o una lista de 2 valores, en caso de esperar 2 valores. De esta forma evitaremos complicar la inter-conexión entre nuestra plataforma y los desarrolladores de tareas.
- **Response:** La respuesta siempre se enviará en formato de **cadena tipo json**, el diccionario deberá tener esta estructura:
 - * **status:** Diccionario del estado de la ejecución de la tarea.
 - **code:** Código de estado, 0 indica ejecución correcta, con -1 o cualquier otro valor, indicará error de ejecución de tarea.
 - **msg:** Mensaje con la descripción del problema, o bien vacío si la ejecución fue exitosa.
 - * **results:** Será el diccionario con los resultados, que se almacenará en la pauta del entrenamiento una vez realizada la actividad. Este diccionario dependerá de la documentación entregada, tendrá un formato variable, de ahí la necesidad de almacenar un json, en lugar de hacer uso de tablas relacionales.
- **task_code.py:** Contendrá el código fuente de la tarea, que tiene que adaptar su entrada y salida a lo descrito anteriormente.
- **requirements.txt:** Cada tarea dispondrá de **su propio entorno virtual** para ejecutarse. Dado que usaremos el gestor de paquetes **pip**, cuando importemos el fichero de instalación, leeremos y cargaremos las librerías que nos demande el programador en la versión **python** adecuada.
- **libraries.zip:** En caso de que la tarea requiera ficheros, librerías, etc... se debería incorporar un **fichero.zip**, el cual será descomprimido en la carpeta donde instalaremos la tarea. Por ejemplo, podríamos requerir modelos de lenguaje pre-entrenados, y descargables para usar en local, como por ejemplo los que encontramos en <https://huggingface.co/>
- **doc.pdf:** La documentación de la versión de la tarea. En sucesivas versiones, se debería ver la opción de cambiar el fichero .pdf por un fichero de tipo **markdown**. (Aunque no es objeto de este TFG, sería deseable).
- **doc.html:** Será una réplica de la documentación, maquetada en .html. para que se pueda almacenar como texto en el la colección de tareas, y podamos mostrarla como ayuda, para los editores de actividades.

Para el presente desarrollo, se podrán realizar algunas tareas de ejemplo, no es necesario que sean excesivamente complejas, ya que no es el objeto del proyecto, por ejemplo en una tarea podríamos

programar una o varias tareas como las siguientes (sobre todo estadísticos): - Media de palabras por minuto. - Total de palabras leídas. - Desviación típica - Distribución de palabras leídas por cada minuto... - Tiempo total - Palabras leídas correctamente - Frecuencia de palabras leídas correctamente - Total de palabras leídas erróneamente - Frecuencia de errores por minuto - Etc ...

Ejemplos de tareas más complejas, podrían implicar el uso de modelos de lenguaje para tareas de **procesamiento de lenguaje natural**, y que serían deseables, pero quedan fuera de este TFG, como: - Tokenización de textos - Devolución de fonemas leídos - Análisis de sentimientos - Estructura gramatical - Comprensión de textos - Etc...

Por convenio, los ficheros de instalación tendrán extensión .zip, pero podríamos eliminar la extensión o llamarla como queramos, al final se hará el tratamiento de fichero .zip.

Como ejemplo de este sistema, tenemos los ficheros que usa **Microsoft** para sus programas de ofimática, podríamos renombrar un .docx a .zip y si lo abrimos, veremos como se organizan sus distintos ficheros, aunque, externamente aparentan ser un fichero propietario único. Si queremos inventarnos una extensión de 3 a 4 caracteres, para poder añadir un icono en nuestro sistema, podríamos hacerlo, pero por simplificar, con un .zip es suficiente.

3.6.2 Instalación de una tarea.

En nuestra aplicación de servidor, deberemos almacenar en una ubicación de persistencia (disco) una carpeta que contenga las tareas. En esta carpeta, se creará una carpeta por tarea, una solución de diseño es que asignemos el id de la tarea en la colección de documentos de tareas, así la clave del documento, nos dará la carpeta de la base de datos.

Dentro de la carpeta, descomprimiremos el zip, para que quede con el siguiente formato (en caso de tener problemas para localizar ficheros, se puede reorganizar):

```
1 <TASK_ID>/
2   info.json
3   libs/
4       model1/
5       library2/
6       env/
7       requeriments.txt
8   docs/
9       doc.md
10      doc.pdf
11      doc.html
12   src/
13      task_code.py
```

Especificación: - **libs**: En esta carpeta se ubicará el entorno virtual, así como las librerías, modelos de

lenguaje, y resto de ficheros de importación. - **env**: Entorno virtual que se instalará con **pip**. - **docs**: Carpeta de documentación del proyecto - **src**: Carpeta para código fuente.

Se creará una colección que se llamará **tasks**, la cual contendrá el documento **json** que viene en el fichero **.zip**

3.6.3 Ejecución de código de instalación (Ayuda)

Subprocess

Usaremos esta librería para poder ejecutar comandos de consola, directamente en nuestro **script python**.

El código siguiente nos muestra como enviar una lista de parámetros, a un **script** el cual construye una cadena de texto en formato **json**, que contiene los parámetros, en este caso dos cadenas, y tendrá esta forma:

```
1 {
2     "parametro1": "Hola",
3     "parametro2": " ",
4     "parametro3": "Mundo"
5 }
```

task.py

```
1 import sys
2 import json
3
4 def main():
5     # Leer los argumentos pasados al script
6     if len(sys.argv) != 2:
7         print("Se necesita exactamente un argumento en formato JSON.")
8         sys.exit(1)
9
10    # Convertir el argumento de JSON a una lista de parámetros
11    parametros = json.loads(sys.argv[1])
12
13    if len(parametros) != 3:
14        print("Se necesitan exactamente tres parámetros.")
15        sys.exit(1)
16
17    # Crear el diccionario con los parámetros
18    resultado = {
19        "parametro1": parametros[0],
20        "parametro2": parametros[1],
21        "parametro3": parametros[2]
22    }
23
```

```
24     # Convertir el diccionario a una cadena JSON y mostrarlo
25     print(json.dumps(resultado))
26
27 if __name__ == "__main__":
28     main()
```

Intérprete de tareas

```
1  import subprocess
2  import json
3
4  def ejecutar_task():
5      # Definir los parámetros
6      parametros = ["Hola", " ", "Mundo"]
7
8      # Convertir los parámetros a una cadena JSON
9      parametros_json = json.dumps(parametros)
10
11     # Ejecutar el script task.py y pasar los parámetros en formato JSON
12     try:
13         result = subprocess.run(['python', 'task.py', parametros_json],
14                                 capture_output=True, text=True, check=True)
15         # Mostrar la salida de task.py
16         print("Salida de task.py:")
17         print(result.stdout)
18     except subprocess.CalledProcessError as e:
19         print(f"El comando falló con el error: {e}")
20         print(e.output)
21
22 if __name__ == "__main__":
23     ejecutar_task()
```

Unzip

Para ejecutar comandos de consola del sistema, como **unzip** en **linux debian**, podremos hacerlo como se muestra en el siguiente ejemplo:

```
1
2  import subprocess
3
4  # Definir el archivo ZIP y el directorio de destino
5  archivo_zip = 'archivo.zip'
6  directorio_destino = 'mi_carpeta_destino'
7
8  # Ejecutar el comando unzip
9  import subprocess
10
11 # Definir el archivo ZIP y el directorio de destino
12 archivo_zip = 'archivo.zip'
13 directorio_destino = 'mi_carpeta_destino'
14
```

```
15 # Ejecutar el comando unzip
16 try:
17     result = subprocess.run(['unzip', archivo_zip, '-d',
18                             directorio_destino], capture_output=True, text=True, check=True)
19     print("Comando ejecutado con éxito:")
20     print(result.stdout)
21 except subprocess.CalledProcessError as e:
22     print(f"El comando falló con el error: {e}")
23     print(e.output)
```

Explicación de los parámetros usados: - **unzip**: El comando a ejecutar. - **-d**: Especifica el directorio de destino donde se descomprimirá el archivo. - **capture_output=True**: Captura la salida estándar y el error estándar. - **text=True**: Devuelve la salida y el error como cadenas de texto en lugar de bytes. - **check=True**: Lanza una excepción si el comando devuelve un código de salida diferente de cero.

Pip

Para crear un fichero **requirements.txt**:

```
1 $ pip freeze > requirements.txt
```

Para instalar el entorno, realizaremos la operación contraria:

```
1 $ pip install -r requirements.txt
```

4. Componentes

Dado que se está usando un **patrón de diseño basado en componentes**, se tendrá que diseñar cada vista, con especial cuidado en no repetir el código de visualización. Por ejemplo, dado que tendremos que hacer una vista para administrar sesiones, y otra para administrar usuarios, otra seguramente para poder administrar la pauta de entrenados y sesiones, parece lógico generar un componente que permita administrar una lista de colecciones, donde cada elemento tenga unas operaciones a realizar por elemento y una cabecera para las agregaciones de los distintos elementos de cada colección.

Del mismo modo, crear componentes de filtro que se adapte a la naturaleza de la colección administrada, parece mejor idea que crear un diseño de filtro diferente para cada colección.

Dado que vamos a tener que crear muchos elementos de tipo “cabecera-líneas”, (usuarios, sesiones, actividades, pauta, etc...), en esta primera versión de la especificación será importante definir bien, que componentes deberemos crear, que podamos personalizar en sus entradas y salidas, y nos sirvan para manejar distintos tipos de entidades.

Al crear los componentes, tener en cuenta que se pueden, y se deben utilizar librerías ya desarrolladas por terceros, como por ejemplo: - **jHtmlArea: WYSIWYG Html Editor jQuery Plugin**, para crear un

editor de texto, para la entrada de datos. Se puede descargar la librería, y enlace el código del proyecto, para ver como se puede personalizar. - **bootstrap: Plantilla Bootstrap**, plantilla **bootstrap** donde se puede ver controles de tabla, tipografías, paneles, gráficos, formularios de ejemplo, etc... Es muy buena idea reutilizar o bien esta, o bien cualquier otra, pero la misma para todo el proyecto, y de esta “paleta” de muestra, construyas los distintos componentes de la aplicación.

4.1 Ejemplo de una vista tipo índice(index)

Imprimibles

+ Todos

+ Página

- Todo

- Página

Mostrar 10 elementos

	ID	Nombre	Operaciones
<input type="checkbox"/>	1,595	Alfa 01 MAY_AR	<div><div></div><div></div><div></div></div>
<input type="checkbox"/>	1,636	Alfa 01 MAY_ES	<div><div></div><div></div><div></div></div>
<input type="checkbox"/>	1,596	Alfa 02 MAY_AR	<div><div></div><div></div><div></div></div>
<input type="checkbox"/>	1,637	Alfa 02 MAY_ES	<div><div></div><div></div><div></div></div>
<input type="checkbox"/>	1,597	Alfa 03 MAY_AR	<div><div></div><div></div><div></div></div>
<input type="checkbox"/>	1,638	Alfa 03 MAY_ES	<div><div></div><div></div><div></div></div>
<input type="checkbox"/>	1,598	Alfa 04 MAY_AR	<div><div></div><div></div><div></div></div>
<input type="checkbox"/>	1,639	Alfa 04 MAY_ES	<div><div></div><div></div><div></div></div>
<input type="checkbox"/>	1,599	Alfa 05 MAY_AR	<div><div></div><div></div><div></div></div>
<input type="checkbox"/>	1,640	Alfa 05 MAY_ES	<div><div></div><div></div><div></div></div>

Mostrando 1 a 10 de 100 registros

Previo

1

2

3

4

5

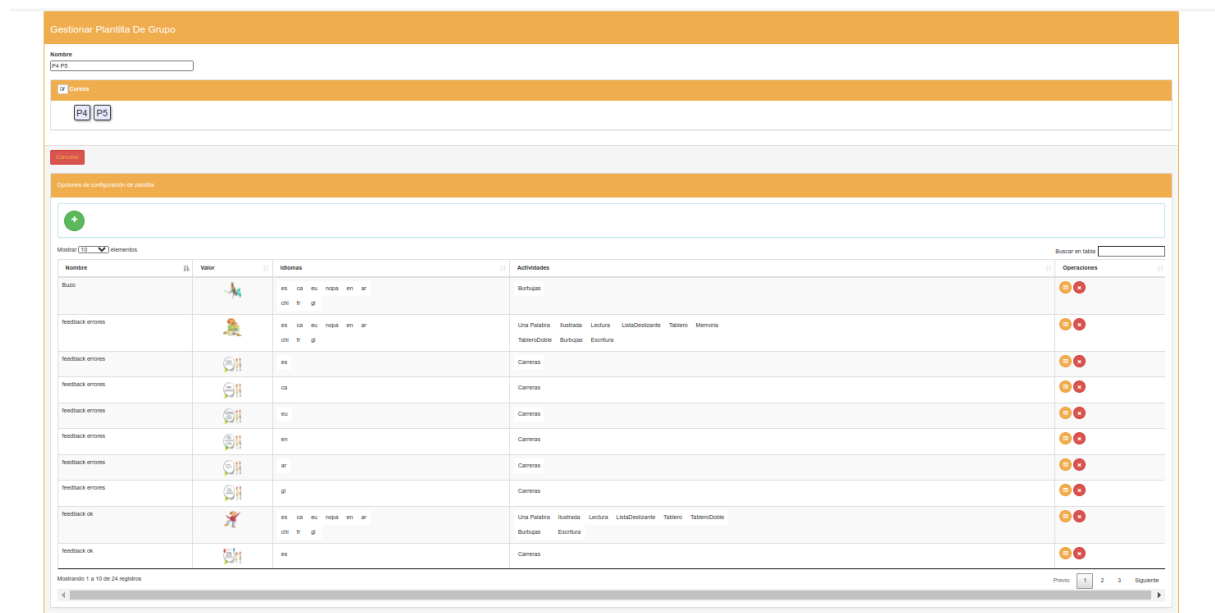
...

10

Siguiente

Figura 1: Ejemplo de una vista tipo índice(index)

4.2 Ejemplo de una vista tipo cabecera-líneas



Nombre	Valor	Metas	Actividades	Operaciones
Bloque		es, ca, eu, nipa, en, ar, u25, n, gl	Burujes	
Feedback errores		es, ca, eu, nipa, en, ar, u25, n, gl	Una Paredra, Rutrada, Lectura, ListaDeCante, Talaro, Memoria, TalaroDoble, Burujes, Escritura	
Feedback errores		es	Camras	
Feedback errores		ca	Camras	
Feedback errores		eu	Camras	
Feedback errores		en	Camras	
Feedback errores		ar	Camras	
Feedback errores		gl	Camras	
Feedback ok		es, ca, eu, nipa, en, ar, u25, n, gl	Una Paredra, Rutrada, Lectura, ListaDeCante, Talaro, Memoria, TalaroDoble, Burujes, Escritura	
Feedback ok		es	Camras	

Figura 2: Ejemplo de una vista tipo cabecera-líneas

5. Vistas

Todas las vistas deberán funcionar, de tal forma que cada vez que se modifique un elemento, se grabará el cambio en la base de datos, es decir, no vamos a crear un botón que tenga que presionar el usuario que esté editando la vista, cuando en el ciclo de vida de la página, se detecte un cambio, se guardará todo el documento de la colección.

5.1 Usuarios

Tal cual se ha desarrollado para el editor de sesiones actual de Glifing.

5.1.1 Index principal de usuarios

Deberá presentar una lista de usuarios dados de alta en el sistema, donde aparezcan sus campos más representativos. Esta lista podrá tener operaciones **C.R.U.D.** tanto a nivel de documento, como poder seleccionar varios documentos a la vez.

- Operaciones por documentos:

- **Editar:** nos deberá permitir editar la información del documento.
 - **Eliminar:** nos deberá preguntar antes si queremos eliminar el documento, para eliminarlo posteriormente en caso de ser confirmado.
 - **Ver:** nos deberá permitir visualizar la información del documento. (Es buena idea crear un solo componente para editar documentos, que actúe en modo solo lectura, o modo edición)
 - **Selección:** marcar o desmarcar el documento seleccionado.
- Operaciones de agregación de documentos:
 - **Añadir:** nos permite crear un documento nuevo.
 - **Eliminar:** nos deberá permitir eliminar varios elementos a la vez, previa confirmación a través de un mensaje que nos indique el número de documentos que vamos a eliminar.
 - **Filtrar:** nos permitirá realizar filtros sobre los documentos.
 - **Paginado:** Tenemos que mostrar los documentos paginados, mostrando un máximo de 10 elementos por página por defecto, pero podremos elegir el paginado de 10,25,50,100 elementos. Además tendremos que disponer de botones que permitan seleccionar los elementos de todas las páginas, o quitar los elementos seleccionados de todas las páginas, así como seleccionar o eliminar la selección de página a página.

Además tendrá que disponer en una de sus columnas, de iconos que identifiquen los roles de los que dispone, se puede por ejemplo escribir las dos primeras letras de cada tipo, dentro de un círculo con un color diferente cuando está activa, y en gris cuando no tiene ese rol. Por ejemplo AD,ED,EN.

5.1.2 Editor de usuario

Para todas las vistas de edición, que tengan solo un documento, sin sub-documentos asociados, se mostrará una capa modal la cual mostrará los controles de los documento editables. Esta capa tendrá un botón para poder cerrar la misma, y cada vez que se cierre la capa de edición, deberá verse reflejado el cambio en la vista de índice.

5.1.3 Editor de pauta

Se generará una vista index, que contenga un listado con todos los usuarios que tienen rol de entrenado. Y una columna que indique el número de sesiones pendientes, otro con el número de sesiones realizadas.

Dado que podría ser un proyecto en sí, vamos a reducir mucho las operaciones que se podrán realizar, serán las siguientes:

- **Filtrar usuarios:** Búsqueda de usuarios por varios campos, nombre de usuario, nombre, apellidos, correo electrónico, etc... al menos intentar que se pueda buscar a un usuario por los campos anteriores. Será operación de agregación.
- **Asignar pauta:** Tanto de forma individual, como para múltiples usuarios. Es deseable que se puedan pautar una o varias sesiones y en fechas distintas. Del mismo modo que en otros apartados, es deseable la pauta múltiple. La individual es la necesaria.
- **Ver detalle de entrenado:** Si seleccionamos a un entrenado, se abrirá una vista que tendrá la cabecera con los datos del entrenado, y un detalle con varias pestañas:
 - **Pauta:** las sesiones que tiene pautadas el entrenado, operaciones:
 - * Eliminar una o varias pautas de este listado.
 - **Historial:** se muestra un listado de las sesiones realizadas, ordenadas de la más actual, hasta la primera, operaciones:
 - * Se podrá descargar el un fichero **.json** de resultados de la sesión.
 - * Se podrá descargar un **.zip** con la carpeta de los audios, en el mismo esquema de carpetas que se ha descrito.

5.1.4 Ejecución de sesiones

Cuando un usuario con rol de entrenado accede al sistema, debe aparecer en su interfaz un botón para poder ejecutar su carga de sesiones. En esta vista, de tipo lista aparecerán las sesiones pendientes (es decir, las sesiones que no tienen resultados).

Se entiende como sesión completada, aquella que tiene todas sus actividades hechas.

La tabla estará ordenada por la sesión más antigua pendiente, con respecto a la fecha actual la primera, y la última, la más cercana a la fecha actual.

Cuando se selecciona una sesión, se ejecutará una vista de player.

5.1.5 Vista de player

Esta parte del proyecto es la única que se hará de forma radicalmente diferente a como se hace en Glifing, por motivos evidentes, no podemos desarrollar actividades complejas a nivel gráfico en tan poco tiempo.

Esta vista lo que hará será cargar la información de la actividad, en una cabecera, y mostrará en un cuerpo, los ítems de la actividad.

Se realizará la sesión de forma secuencial, de la primera a la última de la lista almacenada en la colección de sesiones.

Se irá indicando en cada momento, el número de actividad realizada de tantas actividades realizadas. Para realizar una actividad, en el cuerpo de la actividad se mostrarán primero las instrucciones, y una vez que el usuario presione el botón de iniciar, se mostrarán los items, junto a un botón con micrófono, de color azul, que indica que no se está grabando el audio (o negro).

Para las actividades de texto completo, se mostrará el texto completo en pantalla, si este es largo, se colocará en la capa una barra de desplazamiento lateral, pero nunca horizontal. Sería deseable poder mostrar el texto por páginas, pero esto implica un nivel de desarrollo complejo, y queda fuera de este proyecto. En este tipo de actividades, una vez que se muestra el texto completo el botón pasa a rojo, y empieza a grabar el audio. Cuando el usuario finaliza la lectura, deberá presionar el botón del micrófono, que pasará de rojo a azul (o negro), y pasamos a la siguiente actividad.

Para actividades que muestran palabra a palabra, se simulará el tipo “una palabra” de Glifing. Es decir, se mostrará palabra a palabra en la pantalla. Se muestra la palabra, y se graba el audio, cuando el usuario presiona el botón del micrófono, se muestra la siguiente palabra, y así hasta leer toda la lista. Cuando termina la lista, finaliza la actividad y salta a la siguiente.

Los resultados, es decir, los audios, se envían al **endpoint** correspondiente, cuando finaliza la actividad.

Al terminar todas las actividades, se mostrará un mensaje por pantalla indicando que acabó la sesión, y regresará a la vista de de ejecución de sesiones.

Para grabar audio en un archivo **.wav** desde un micrófono en un documento **HTML**, puedes usar la **API** de **MediaRecorder** disponible en navegadores modernos junto con una librería de **JavaScript** como **Recorder.js** para convertir el audio grabado en formato **.wav**, veamos un ejemplo:

index.html

```
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta name="viewport" content="width=device-width, initial-scale
6     =1.0">
7   <title>Audio Recorder</title>
8 </head>
9 <body>
10  <h1>Audio Recorder</h1>
11  <button id="start">Start Recording</button>
12  <button id="stop" disabled>Stop Recording</button>
13  <audio id="audio" controls></audio>
14  <script src="https://cdnjs.cloudflare.com/ajax/libs/recorderjs
15    /0.1.0/recorder.js"></script>
16  <script>
17    const startButton = document.getElementById('start');
18    const stopButton = document.getElementById('stop');
```

```
17     const audio = document.getElementById('audio');
18     let mediaRecorder;
19     let audioChunks = [];
20
21     startButton.addEventListener('click', async () => {
22         const stream = await navigator.mediaDevices.getUserMedia({
23             audio: true });
24         mediaRecorder = new MediaRecorder(stream);
25         mediaRecorder.ondataavailable = event => {
26             audioChunks.push(event.data);
27         };
28         mediaRecorder.onstop = async () => {
29             const audioBlob = new Blob(audioChunks, { type: 'audio/
30             wav' });
31             const audioUrl = URL.createObjectURL(audioBlob);
32             audio.src = audioUrl;
33             audioChunks = [];
34         };
35         mediaRecorder.start();
36         startButton.disabled = true;
37         stopButton.disabled = false;
38     });
39
40     stopButton.addEventListener('click', () => {
41         mediaRecorder.stop();
42         startButton.disabled = false;
43         stopButton.disabled = true;
44     });
45 </script>
46 </body>
47 </html>
```

- 1. HTML Estructura:
 - Tenemos dos botones: uno para iniciar la grabación y otro para detenerla.
 - Un elemento `<audio>` para reproducir la grabación.
- 2. JavaScript:
 - Utiliza la API `navigator.mediaDevices.getUserMedia` para solicitar acceso al micrófono del usuario.
 - Al hacer clic en el botón “Start Recording”, se inicia la grabación utilizando `MediaRecorder`.
 - Los datos de audio se almacenan en `audioChunks`.
 - Cuando se detiene la grabación, los datos se convierten en un **Blob** con **tipo MIME audio/-wav**, y se genera una URL para el archivo de audio que se asigna al elemento `<audio>` para su reproducción.

- 3. Librería Recorder.js:
 - Aunque se carga en el **HTML**, este ejemplo no hace uso explícito de `Recorder.js` ya que la API `MediaRecorder` es suficiente para grabar audio en formato **.wav**. Existen funciones más avanzadas de grabación y procesamiento de audio, podrías integrar Recorder.js o una librería similar para hacerlas funcionar.

5.2 Sesiones

Tal cual se ha desarrollado para el editor de sesiones actual de Glifing.

5.2.1 Index principal de sesiones

Vista de tipo lista, como se ha descrito para la vista de usuarios.

5.2.2 Editor de sesiones

Vista de tipo cabecera y líneas. En la cabecera tendrá el componente de edición de un documento (ficha de sesiones) y en las líneas se editarán una vista index de las actividades para esa sesión.

5.3 Actividades

Tal cual se ha desarrollado para el editor de sesiones actual de Glifing.

5.3.1 Index principal de actividades

Iría integrada en las líneas del detalle del editor de sesiones.

5.3.2 Editor de actividades

Será de tipo cabecera-líneas, donde la cabecera será un componente de tipo ficha de la actividad, y las líneas será una vista index de los items que contiene la actividad.

5.4 Items

Tal cual se ha desarrollado para el editor de sesiones actual de Glifing.

5.4.1 Index principal de items

Ir  integrada en las l neas del detalle del editor de actividades.

5.4.2 Editor de items

Ser  un formulario modal del componente tipo ficha donde aparece el item.

5.5 Status

Seguir  el mismo patr n descrito anteriormente.

5.5.1 Index principal de status

Vista de tipo lista, como se ha descrito para la vista de usuarios.

5.5.2 Editor de estados

Ser  un formulario modal del componente tipo ficha donde aparece el estado.

5.6 Task

Tal cual se ha desarrollado para el editor de tipos de actividades actual de Glifing.

5.6.1 Index principal de tareas

Vista de tipo lista, como se ha descrito para la vista de usuarios. Se tiene que a adir la operaci n de importar tarea, para que automatice la instalaci n, tal y como se explica en el modelo.

5.6.2 Editor de tareas

Dado que las tareas no tienen sub-elementos asociados:

- **Nuevas tarea:** Se har  una operaci n para a adir, que permitir  subir al fichero de instalaci n. Para este proyecto, se instalar n las tareas de una en una. Ser  operaci n de agregaci n (en el marco superior de la lista de tareas)

- **Eliminar tarea:** Se hará tanto por agregación al eliminar varias a la vez, como una a una. Lógicamente se pedirá confirmación. No se permitirá eliminar una tarea, si existen actividades pautadas a un usuario, pendientes de realizar, que tengan una tarea pendiente.
- **Modificar tarea:** Se borrará la información de la tarea anterior de su carpeta de disco, y se instalará la nueva versión.
- **Mostrar tarea:** Ventana modal, que nos mostrará la información del **json** de la tarea, si existen fichero **.html**, **.md** se mostrará en la ventana modal, del mismo modo, si existe fichero **pdf**, se generará el icono para el descargable correspondiente.

6 Inter-conexión de capas: Api

Para la comunicación entre la persistencia del sistema y las vistas, se realizará a través de una **api** del tipo **Api Rest**, la cual será implementada a través de:

- **Lenguaje Servidor:** Se realizará a través de **Python**, sobre el framework **Django-rest**.
- **Documentación:** Se escribirá a través de **Swagger**.

Entendemos por persistencia, toda información que necesite perdurar en el tiempo:

- **Base de datos:** Se implementará a través de un sistema **MongoDB**, cuyas librerías se conecten a la api.
- **Almacén de ficheros:** Todo tipo de ficheros del sistema, si los hubiera, deberán ser almacenados en una ubicación que pueda ser gestionada directamente por la api, de forma que se pueda definir a través del fichero de configuración de la api. De tal forma que, si se decide cambiar la ruta, o emplazamiento, escalar, etc... sea transparente para las vistas. Por ejemplo, una vista que quiere mostrar, o almacenar un fichero, hará la solicitud a un endpoint de la **Api**, el cual devolverá lo solicitado. Si el administrador de la Api, decide cambiar el volumen físico, usar varias ubicaciones para disponer de redundancia, etc... esta parte quedará encapsulada en la implementación de la Api.
- Etc..

El objetivo de implementar la infraestructura basadas en el sistema de capas descrito hasta el momento, persigue independizar al sistema en piezas modulares estándar, que cumplan las siguientes características:

- **“Caja negra”:** Cada pieza de la infraestructura tendrá entradas y salidas claramente diferenciadas.
- **Escalabilidad:** Podremos implementar cada una de nuestras piezas en una máquina física separada, o en una sola máquina para todo el conjunto, esto nos permitirá disponer de:

- **Escalado horizontal:** Podremos distribuir en varios servidores físicos, la ejecución de la lógica de nuestro sistema, balanceando la carga de trabajo, si en un futuro, la demanda de usuarios así lo requiriese.
- **Escalado vertical:** Podríamos aumentar la potencia de la máquina física, si la demanda de usuarios así lo requiriese.

6.1 Operaciones básicas de la Api

Todos los modelos definidos en el punto 3, deberán operarse a través de la Api.

6.2 Operaciones especiales de la Api

Las operaciones especiales de la api implican la gestión de resultados:

- Cuando se recibe una lista de audios, de cada actividad realizada, en formato **.wav**, el sistema tendrá que por un lado almacenar los audios como se ha descrito en apartados anteriores, y, se deberá ejecutar la tarea sobre los mismos, para generar el json de resultados.

Las operaciones deseables, suponen que, se puedan exportar, previa autenticación, todos los resultados de un usuario en concreto.

7. Recomendaciones y buenas prácticas

Consejos a cerca de como hacer uso de un repositorio de código de forma más eficiente.

7.1 Repositorios

7.1.1 Buenas Prácticas con Git, uso de **.gitignore**, proceso de instalación

Git dispone de un fichero que se llama **.gitignore** para:

- No almacenar ficheros de configuración local
 - Es buena práctica generar un fichero con el sufijo ****_bkp**** para dejar una muestra de configuración.
 - Sino queremos ensuciar el directorio con ficheros de este estilo, es buena práctica crear un directorio que se llame plantillas de configuración, o **templates**, y guardar ahí estos

ficheros, también se puede añadir a la documentación del proyecto información de que es cada una las plantillas, con algún ejemplo.

- No almacenar ficheros estáticos.
 - Se entiende por ello, **todos los ficheros que son susceptibles a cambios de código**.
 - Por ejemplo
 - * Multimedia
 - * Datos
 - * Librerías externas js
 - * Etc
 - Es buena práctica:
 - * Guardar todos estos ficheros en formato comprimido en una ubicación a parte, alcanzable a través de internet (Una url), para que un script de despliegue, con un comando **wget+unzip** los ubique. De esta forma el repositorio de código no tendrá “basura”, pensar que por cada commit, se duplica el espacio de almacenamiento, o se ralentiza el despliegue.
- No almacenar las librerías estándar del proyecto
 - Tales como:
 - * **Entornos virtuales en python**
 - Para ello se puede seguir la misma estrategia anterior, guardarlas en un .zip, una solución “elegante”, , será **congelar los paquetes de configuración en un fichero**, además este sistema, nos permite posteriormente generar un **sistema de despliegue** basado en scripts. Para ello:
 - * Con **conda** se puede almacenar un fichero con la lista de paquetes y versiones .py del entorno.
 - * A través de **pip**, etc...
 - A veces esto da problemas, debido a cambios externos de los desarrolladores de paquetes, referencias cruzadas, etc... Por eso otra técnica algo menos elegante, pero efectiva, sobre todo para proyectos donde no es muy crítico disponer de las últimas versiones, se puede crear un.zip con estos paquetes de configuración, ligado a una url para el momento del despliegue. Es otra forma de **congelar los paquetes de configuración**.

7.1.2 ¿ Cuando y como implementar esto?

- Cuando los ficheros ocupan un espacio considerable, hay que buscar el balance entre eficacia y servicio, por ejemplo, si se usan imágenes para maquetar una vista, o una carpeta de librerías js, y algunos ficheros de maquetación front, que no ocupan menos de 30MB pues no es muy eficiente lo anterior.
- Una técnica sencilla y barata es crear un gdrive, dropbox, etc... con un usuario y clave, donde quede documentado en la memoria para hacerlo manualmente. Profesionalmente, lo ideal es hacer uso de un sistema de ficheros que permita acceso bajo autorización (a través de una api), en sistemas de computación en la nube, como por ejemplo el servicio **S3** de **AWS**. Pero dado el coste de servicio, no tiene mucho sentido para este proyecto universitario.
- Quedaría muy elegante, hacer un **script bash** que ejecute la instalación y configuración “*auto-mágicamente*”. Por eso buscar un servicio de **computación en la nube** que tenga una api documentada, donde podáis enviar credenciales, a través de una llamada **wget**

7.2 Variables, paths, strings, etc...

En todas las aplicaciones tenemos que colocar cadenas de texto, las cuales pueden ser:

- Rutas físicas a disco
- Urls
- Configuraciones de librerías
- Etc...

7.2.1 Identificar las cadenas

Es decir, aquellas cadenas de texto que son susceptibles de ser usadas bajo el término coloquial “hardcodear una cadena”. Para detectar este tipo de cadenas, tenemos que pensar en, si tuviera que cambiar esa cadena, ¿será necesario crear una versión de código nueva? pues entonces, casi con seguridad, es una variable de configuración.

Por ejemplo ejemplos:

- **Urls a los endpoints:** Si los endpoints cambian sus ubicaciones, por ejemplo, porque queremos balancear carga, o porque queremos hacer uso de un sistema de pruebas, ¿sería sensato tener que re-escribir todas estas direcciones?
- **Rutas a ficheros estáticos:** Del mismo modo que en el caso anterior, si tenemos que cambiar una ruta a la carpeta donde almacenamos logs, multimedia, etc... no tiene sentido que tengamos que crear una nueva versión de código.

- **Cadenas de conexión:** Claves, direcciones a bases de datos, y en definitiva cualquier parámetro susceptible de poder ser editado por el usuario de la aplicación.

Como solución, se pueden generar uno o varios ficheros de texto, de configuración, donde las cadenas carguen como variables en nuestro código.

Hay que pensar que, si queremos realizar un desarrollo exportable a otros desarrolladores, rara vez van a disponer de un sistema de carpetas parecido al nuestro.

7.2.2 Idiomas

Aunque no es objeto de este proyecto, se tiene que tener en cuenta, que la gran amplia mayoría de los frameworks, tienen unos métodos definidos, para que en lugar de escribir las cadenas de texto de las vistas, apliquemos una función que, en mucho resumen, se encargarán de realizar la traducción, si nuestra plataforma soporta varios idiomas.

Cada framework tiene su propio sistema, el cual se reduce a escoger un idioma de referencia, y se generan tablas de correspondencia de dicha cadena, con los idiomas que queremos escribir a disposición del usuario en nuestras vistas. Para que el sistema que renderiza la vista genere la cadena correcta, deberemos escribir las mismas dentro del método correspondiente como parámetro.