UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Escuela Técnica Superior de Ingeniería de Telecomunicación

Explotación de eventos generados por Moodle y su uso para personalización de la experiencia de aprendizaje

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA TELEMÁTICA

Autor: Alberto Ros Galian

Director: Mathieu Kessler

Codirector: Daniel Pérez Berenguer

Cartagena, 29 de febrero de 2024

Agradecimientos

En primer lugar, me gustaría agradecer a los directores de este proyecto, Mathieu Kessler y Daniel Pérez Berenguer. He estado en contacto constante durante la realización de este proyecto y me han ayudado a resolver absolutamente todas las dudas que he ido teniendo todos estos meses.

En segundo lugar, a la Universidad Politécnica de Cartagena durante todos estos años y a todos los profesores que me han ido formando. Muchos de ellos me han despertado una curiosidad inmensa por ciertas ramas que antes no me llamaban la atención y que en un futuro me interesaría poder profundizar en ellas.

Y por supuesto, a mi familia por apoyarme siempre y a mis amigos. Muchos de ellos conocidos en esta universidad y en gran parte la razón por la que me llevo un muy buen recuerdo de mi etapa universitaria.

Muchas gracias.

Índice

1. Introducción

El uso obligado de plataformas digitales en el ámbito de la enseñanza por culpa de la pandemia COVID-19 ha acelerado la transformación digital en la educación, permitiendo a profesores utilizar una variedad de herramientas para recopilar y analizar datos sobre el rendimiento de los estudiantes, lo que permite tomar decisiones más acordes e informadas sobre como enseñar y ajustar el trabajo para así satisfacer las necesidades individuales de los estudiantes. Además, la digitalización ha permitido a los profesores utilizar herramientas de analítica de aprendizaje.

Los estudios realizados por el INTEF (2017) destacan la importancia de las analíticas de aprendizaje como herramientas clave para mejorar los procesos educativos. Estas implican la medida, recopilación, análisis e informe de datos sobre los estudiantes y sus contextos, con el fin de comprender y optimizar el aprendizaje y los entornos en que tiene lugar, con el fin de mejorarlo. Algunos de los beneficios que tienen las analíticas de aprendizaje pueden ayudar a los profesores a mejorar la motivación, prevenir el abandono temprano y mejorar el entendimiento global. Además, permiten a profesores personalizar el aprendizaje para satisfacer las necesidades individuales de los estudiantes.

En esta línea, la UNESCO (s. f.) ha enfatizado la relevancia del aprendizaje personalizado, sugiriendo que la personalización es fundamental para atender las necesidades individuales de los estudiantes. La personalización del aprendizaje se basa en la idea de que cada estudiante es único y tiene necesidades y preferencias de aprendizaje individuales. Esta se ha vuelto cada vez más popular en los últimos años por diversos factores. En primer lugar, la tecnología ha hecho posible que los profesores personalicen el aprendizaje de manera más efectiva, y, en segundo lugar, se ha convertido en una prioridad para muchos padres, al buscar un enfoque educativo más centrado en el estudiante.

Existen muchas herramientas que permiten o se basan en la explotación de analíticas de aprendizaje, como, por ejemplo, EDpuzzle, que permite a los docentes crear y editar videos educativos, enriqueciendo así la experiencia de aprendizaje (IDdocente, s. f.), o Classcraft (INTEF, s. f.), que permite a profesores dirigir un juego de rol donde los alumnos encarnan diferentes personajes, aumentando la motivación, alentando al trabajo en equipo y favoreciendo un mejor comportamiento en las aulas. Este último se trata de un ejemplo de ludificación (gamificación), donde el juego transforma la manera en la que los alumnos viven la enseñanza.

El uso de sistemas de gestión de aprendizaje (IdeasPropiasEditorial, s. f.), como Moodle, cuya estructura modular permite a los profesores y centros adaptar sus cursos según las necesidades específicas, es fundamental para facilitar la educación en línea. Es la plataforma de aprendizaje usada por la UPCT, y es la que se va a usar en este trabajo. Este sistema permite la incorporación de plugins (Moodle, 2023). Además, explota los datos que genera para proporcionar información relevante a profesores y alumnos, como, por ejemplo, el vencimiento de las próximas tareas o las últimas modificaciones en los cursos. Entre los datos que genera están los eventos (Moodle, 2023), que son los datos que vamos a explotar en este proyecto y que permiten extraer información útil que facilita a los profesores el seguimiento y monitorización de sus alumnos.

Este trabajo es una prueba de concepto (Asana, s. f.), cuyo objetivo es diseñar la arquitectura, implementar los distintos componentes y demostrar la viabilidad de la solución.

En resumen, este proyecto consiste en la recopilación de dichos eventos, su análisis para la obtención de datos relevantes tanto para profesores y alumnos, y la representación de dichos datos en la página de Moodle ya sea mediante alertas para el profesor respecto a sus alumnos, mensajes de información para dichos alumnos, etc.

2. Estructura y flujo del proyecto

Con el objetivo de proporcionar una visión global del proyecto, en este capítulo vamos a ver, sin entrar en mucho detalle, los pasos que se han seguido en la realización de este, desde la instalación local de Moodle en un contenedor de Docker, hasta la visualización de los datos obtenidos de eventos en bloques de Moodle.

2.1. Estructura y tecnologías del proyecto

El objetivo principal de este proyecto es obtener, a partir de la recopilación y análisis de los eventos generados por la plataforma Moodle, datos que sirvan a profesores y alumnos para monitorizar el aprendizaje:

Primer paso: Instalar la plataforma Moodle de manera local en un contenedor de Docker. Aunque nosotros lo instalamos localmente al ser más práctico para el desarrollo y las pruebas, es importante saber que la idea de este proyecto es usarlo en un sistema de producción como el del aula virtual de la UPCT. En esta instalación de Moodle, crearemos varios usuarios con roles de administrador, profesores y alumnos. Además, generaremos de manera planificada cursos, datos de entregas de tareas, etc.

Junto con la instalación que hemos elegido de Moodle viene incluida la instalación de la base de datos relacional MariaDB, que nos ayudará a no perder los datos que generemos en la plataforma.

Segundo paso: Desarrollar, en PHP, un plugin de tipo local para Moodle que recopile los eventos generados por este y los envíe a un microservicio en Azure preparado por los directores de este trabajo. Este microservicio a su vez empaquetará dichos eventos en un solo archivo avro y los cargará en un contenedor de Azure preparado por nosotros.

Tercer paso: A través de Python, descargar los archivos, desempaquetarlos y analizarlos. Una vez analizados y obtenidos los datos que vamos a presentar tanto a profesores como alumnos, dichos datos los empaquetamos de nuevo en archivos avro y los cargamos en un contenedor de Azure distinto al anterior, que servirá únicamente para almacenar los datos analizados.

Cuarto paso: Consiste en desarrollar, de nuevo en Python, una función en Azure que contenga dos endpoints. Un endpoint estará diseñado para los profesores y el otro para los alumnos. A estos endpoints les mandaremos solicitudes HTTP y nos devolverán, si hemos hecho la solicitud de manera exitosa, las respuestas HTTP que contendrá los datos deseados.

Quinto paso: Una vez comprobado que estos endpoints funcionan con éxito a través de Postman, desarrollar de nuevo en PHP dos plugins, esta vez de tipo bloque, cuyo contenido serán los datos analizados que contenían las respuestas HTTP de los endpoints.

Desarrollados estos dos plugins y conseguido la visualización de los datos en Moodle, podemos dar por concluido este proyecto.

Figura 1: *Logos de todas las tecnológicas mencionadas*

2.2. Gráfica visual del proyecto

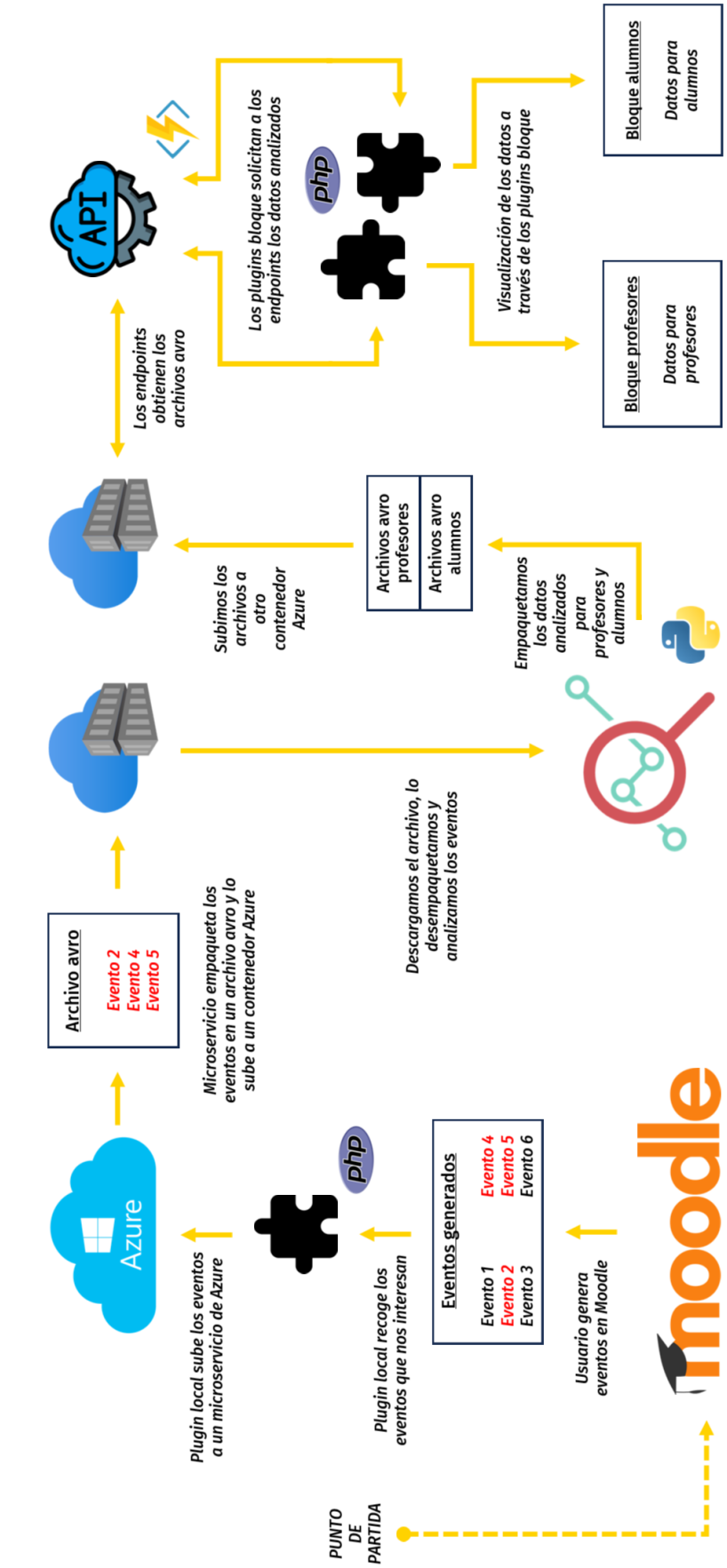

Figura 2: *Gráfica visual del proyecto*

3. Desarrollo del proyecto

Habiendo visto la visión general de las diferentes partes del proyecto, este capítulo va a consistir en ver en profundidad cada una de las partes de este.

Iré detallando de manera cronológica el desarrollo de cada una de las partes, acompañando con su código correspondiente, y en caso de verlo oportuno, una explicación de las tecnologías que se usan.

Los bloques de código se han copiado y pegado directamente de Visual Studio Code con el tema de color 'Light', se les ha aplicado un borde externo y un ligero sombreado gris claro para diferenciar las secciones de código y el resto, además de añadir un pequeño resumen del código justo a continuación.

En el anexo indicamos el repositorio de GitHub donde este se encuentra todo el código.

3.1. Instalación de Moodle en Docker

El primer paso consiste simplemente en instalar una imagen que contenga Moodle en Docker. Para la instalación de Moodle para este proyecto se han escogido la imagen preparada por la compañía Bitnami (Bitnami, 2023), que viene también con una imagen de MariaDB, una base de datos relacional que nos va a servir para persistir los datos y así no perderlos cada vez que iniciemos Moodle. Moodle y MariaDB conformarán lo que en Docker se denomina un contenedor.

3.1.1. Imágenes, contenedores y volúmenes en Docker

En Docker, una imagen es un paquete de software que contiene todo lo necesario para ejecutar una aplicación, desde el código, las bibliotecas y las dependencias. Estas imágenes se utilizan como plantillas para crear contenedores. Un contenedor es una unidad de software que contiene todo lo necesario para ejecutar una aplicación, incluyendo el código, las bibliotecas y las dependencias. La principal diferencia entre imagen y contenedor es que un contenedor es una instancia en ejecución de una imagen, es decir, los contenedores se pueden crear a partir de imágenes, pero no se pueden crear imágenes a partir de contenedores. Y por su parte, los volúmenes son una forma de almacenar datos de manera persistente en Docker, de manera que no se pierdan datos entre las ejecuciones de los contenedores.

Una de las facetas que más caracterizan a Docker es que estos contenedores son portátiles, lo cual quiere decir que se pueden ejecutar en cualquier sistema que tenga Docker instalado.

3.1.2. Archivo docker-compose

Para instalar Moodle y MariaDB de manera simultánea, usamos Docker Compose, un comando básico de Docker que nos permite instalar/iniciar, de acuerdo con el contenido de un archivo YML llamado '*docker-compose.yml*', múltiples imágenes con su configuración correspondiente. El contenido de dicho archivo para el proyecto es el siguiente:

```
services:
```

```
mariadb:
  image: docker.io/bitnami/mariadb:11.0
  environment:
    - ALLOW_EMPTY_PASSWORD=yes
    - MARIADB_USER=bn_moodle
    - MARIADB_DATABASE=bitnami_moodle
    - MARIADB_CHARACTER_SET=utf8mb4
    - MARIADB_COLLATE=utf8mb4_unicode_ci
  ports:
    - '3306:3306'
  volumes:
    - 'mariadb_data:/bitnami/mariadb'
```

Parte del código del archivo 'docker-compose.yml'

Esta primera parte del archivo define el servicio '*mariadb*', donde se usa la imagen definida en '*image*'. Los puertos que va a usar en la maquina local se definen en '*ports*', y el volumen que se utilizará para almacenar los datos se define en '*volumes*', que tiene la sintaxis siguiente: '*nombre_del_volumen: ruta_en_el_contenedor*'.

Por último, en '*enviroment*' se definen las variables de entorno de MariaDB:

- **ALLOW_EMPTY_PASSWORD:** Se utiliza para permitir ingresar a la base de datos con una contraseña vacía. Esto es muy útil para entornos de desarrollo, pero para entornos de producción no se debe usar en ninguna circunstancia.
- **MARIADB_USER:** Se utiliza para establecer el nombre de usuario de la base de datos de MariaDB. En nuestro caso usamos la que viene por defecto con Bitnami, '*bn_moodle*'.
- **MARIADB_DATABASE:** Se utiliza para establecer el nombre de la base de datos. Como en el caso anterior, usamos la que viene por defecto de Bitnami, '*bitnami_moodle*'.
- **MARIADB_CHARACTER_SET:** Se utiliza para establecer el conjunto de caracteres de la base de datos. Usamos '*utf8mb4*', que resumidamente, es un conjunto que admite caracteres Unicode, incluyendo emojis y otros tipos de caracteres.
- **MARIADB_COLLATE:** Se utiliza para establecer la intercalación de la base de datos. De manera muy resumida, establece la manera en la que se comparan y ordenan cadenas de caracteres. Un ejemplo de ello es si se distingue entre mayúsculas y minúsculas, acentos y demás. En nuestro caso, '*utf8mb4_unicode_ci*' es útil cuando se trabaja con cadenas de caracteres en varios idiomas.

```
moodle:
  image: docker.io/bitnami/moodle:4.2
  ports:
    - '80:8080'
    - '443:8443'
  environment:
    - MOODLE_DATABASE_HOST=mariadb
    - MOODLE_DATABASE_PORT_NUMBER=3306
```

```
      - MOODLE_DATABASE_USER=bn_moodle
      - MOODLE_DATABASE_NAME=bitnami_moodle
      - ALLOW_EMPTY_PASSWORD=yes
      - MOODLE_USERNAME=alber
      - MOODLE_PASSWORD=alber
    volumes:
      - 'moodle_data:/bitnami/moodle'
      - 'moodledata_data:/bitnami/moodledata'
    depends_on:
      - mariadb
```

Parte del código del archivo 'docker-compose.yml'

Esta parte define la imagen de Moodle. Además de las variables de entorno, lo único a comentar de esta sección de código es que, como podemos intuir de '*depends_on*', Moodle depende de la base de datos MariaDB, y dicha línea de código garantiza que la base de datos siempre iniciará antes que Moodle para evitar errores a la hora de iniciar las imágenes.

Las variables de entorno de Moodle son:

- **MOODLE_DATABASE_HOST:** Se utiliza para especificar el host de la base de datos que usará Moodle para almacenar los datos, en nuestro caso, MariaDB.
- **MOODLE_DATABASE_PORT_NUMBER:** Se utiliza para especificar el puerto de MariaDB. Usamos el mismo que hemos establecido en las variables de entorno de MariaDB, el 3306.
- **MOODLE_DATABASE_USER:** El mismo que '*MARIADB_USER*'. Usamos el mismo para simplificar la configuración.
- **MOODLE_DATABASE_NAME:** El mismo que '*MARIADB_DATABASE*'. Especificamos la base de datos que usará Moodle.
- **MOODLE_USERNAME:** Se utiliza para especificar el nombre de usuario que utilizaremos para iniciar sesión en Moodle. Este tendrá rol de admin.
- **MOODLE_PASSWORD:** Se utiliza para especificar la contraseña que usaremos para iniciar sesión en Moodle. Para simplificar, será la misma que el nombre de usuario.

```
volumes:
  mariadb_data:
    driver: local
  moodle_data:
    driver: local
  moodledata_data:
    driver: local
```

Parte del código del archivo 'docker-compose.yml'

Por último, la sección '*volumes*' define que los volúmenes definidos de cada imagen se almacenarán de manera local, es decir, en el sistema de archivos del host.

El archivo '*docker-compose.yml*' es prácticamente el mismo que usa Bitnami, pero con ligeras modificaciones para nuestro proyecto. El comando para instalar/iniciar el contenedor que contiene tanto Moodle y MariaDB es:

```
>docker-compose up
```

Si no hemos instalado antes el contenedor, lo instalará y lo iniciará. Si ya estaba previamente instalado, lo que hará será simplemente iniciarlo.

Una vez instalado, la aplicación de Docker Desktop se verá así:

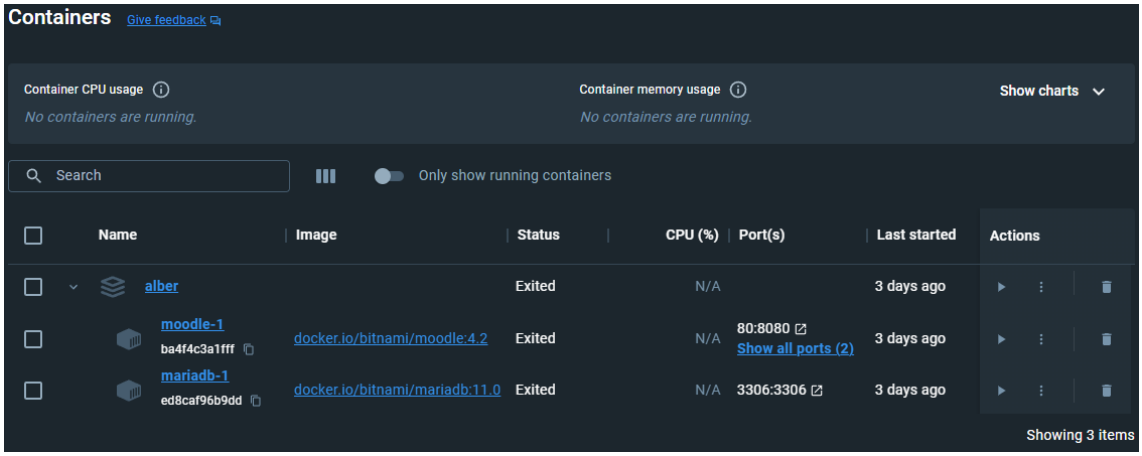

Figura 3: *Página de contenedores de la aplicación Docker Desktop*

Para comprobar que todo se ha instalado correctamente ingresamos en la dirección de localhost:8080 en cualquier navegador. Si nos sale la siguiente página, la instalación ha sido un éxito:

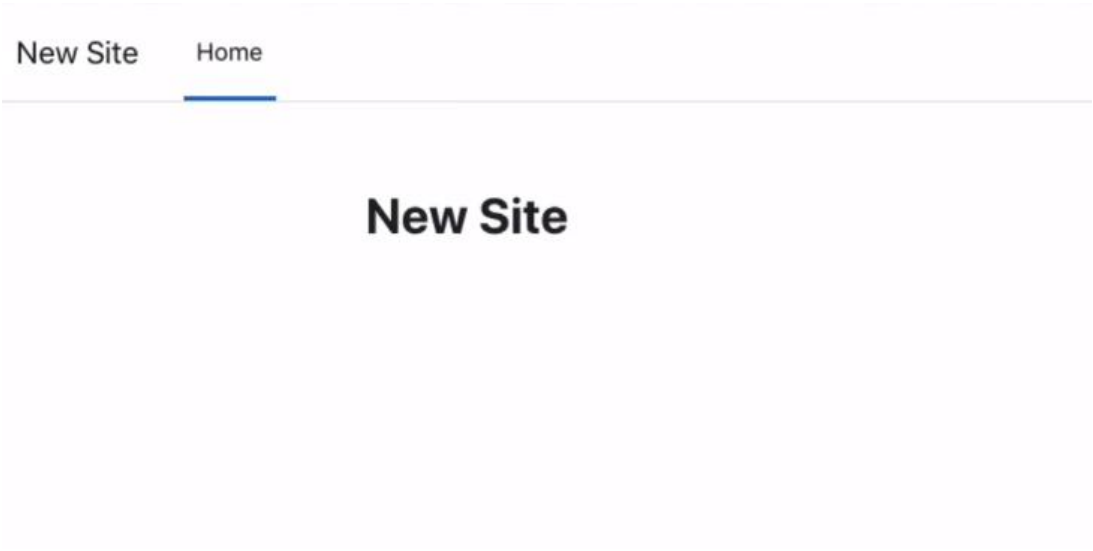

Figura 4: *Página principal de Moodle recién instalado*

Para interrumpir la ejecución del contenedor, es decir, tanto de la imagen de Moodle como la de MariaDB, podemos usar el siguiente comando:

```
>docker-compose stop
```

Con Moodle en este estado, podemos continuar con la creación del plugin local.

3.2. Desarrollo del plugin event listener

El siguiente paso sería desarrollar un plugin muy sencillo para Moodle. El objetivo con este plugin es obtener los eventos generados por Moodle (no todos, los que nos interesen) y subirlos a un microservicio de Azure preparado por los tutores del TFG, y de ahí los suben a un contenedor también en Azure que tendremos que crear más adelante.

Antes de explicar el desarrollo del plugin, es importante conocer qué es un plugin, los diferentes tipos que existen en Moodle y qué son los eventos.

3.2.1. Plugins

Moodle tiene lo que se denominan '*plugins*', componentes de software que se usan para agregar características y funcionalidades adicionales a Moodle. Dichos plugins permiten a usuarios y organizaciones extender y personalizar Moodle más allá de lo hecho por los desarrolladores de Moodle.

Hay muchos tipos de plugins (MoodleDev, s. f.). Algunos ejemplos de plugins incluyen bloques, filtros, informes, actividades, temas y mucho más. La funcionalidad de nuestro plugin no entra dentro de ninguna categoría estándar de Moodle, por lo que el tipo de plugin que nos interesa es del tipo '*local*'.

3.2.2. Eventos

En Moodle, los eventos son piezas de información que describen la mayoría de las acciones que ocurren. Estos se crean principalmente a causa de acciones de los usuarios. También pueden ser creados por acciones de la administración, aunque estos últimos no nos interesan.

Cuando ocurre una acción que tiene asociada un evento, este se crea utilizando la API de eventos, y esta API tiene una lista existente de eventos de todo tipo, desde iniciar sesión, la visita de algún recurso dentro de un curso hasta la subida de algún documento a una tarea. La lista que contiene los eventos existentes en la documentación de Moodle está obsoleta, para saber con exactitud los eventos que contiene nuestra versión de Moodle local, debemos entrar en '*Site administration* > *Reports* > *Events list*'.

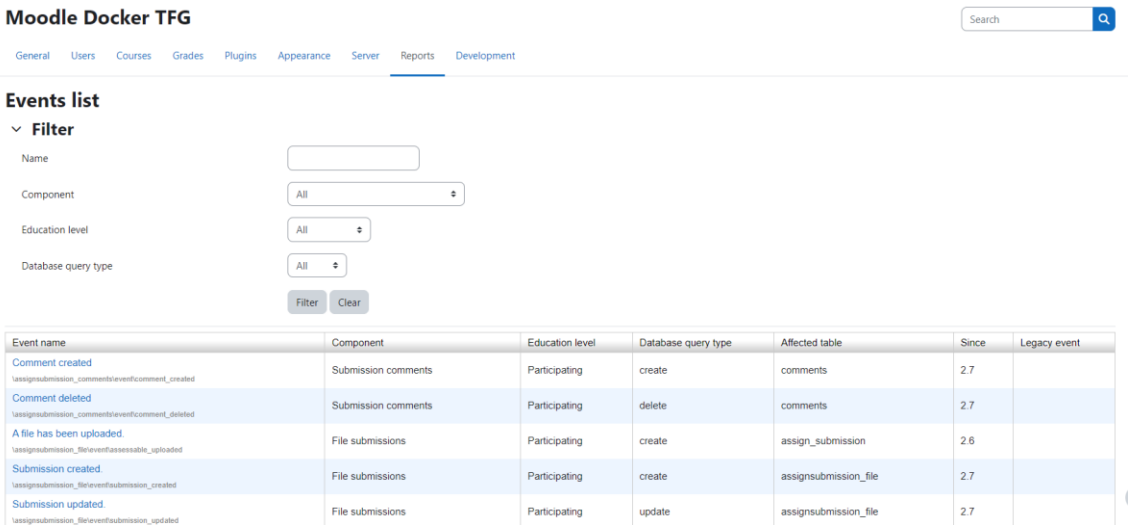

Figura 5: *Lista de eventos de la versión de Moodle local*

¿Y qué información contienen los eventos? Como ejemplo, este es un evento de inicio de sesión en formato JSON:


```
{
    “eventname”: “\\core\\event\\user_loggedin”,
    “component”: “core”,
    “action”: “loggedin”,
    “target”: “user”,
    “objecttable”: “user”,
    “objectid”: “2”,
    “crud”: “r”,
    “edulevel”: 0,
    “contextid”: 1,
    “contextlevel”: 10,
    “contextinstanceid”: 0,
    “userid”: “3”,
    “courseid”: 0,
    “relateduserid”: null,
    “anonymous”: 0,
    “other”: {“username”:  “alber”, “extrauserinfo”: []},
    “timecreated”: 1696591689,
    “domain”: tfgalberto@tfg.es
}
```


Evento de inicio de sesión en formato JSON

Todos los eventos tienen los mismos campos, solo que en algunos la información dentro de ‘*other*’ es distinta.

Los campos más relevantes serían:

- **eventname**: el nombre del evento.
- **userid**: el ID del usuario que provocó el evento.
- **courseid**: el curso donde se creó dicho evento (en este caso es 0 al no estar relacionado con ningún curso).
- **other**: información adicional del evento específico.
- **timecreated**: fecha/hora en la que se creó el evento en formato timestamp Unix.

El parámetro ‘*domain*’ se lo añadimos por código para diferenciar que estos eventos han sido generados para la realización de este TFG.

3.2.3. Plugin local ‘send_events’

Una vez definido lo que son los plugins y los eventos, vamos a ver qué hace exactamente nuestro plugin y cómo funciona.

Parte de su funcionamiento consiste en ‘escuchar’ qué eventos se crean para hacer algo, por lo que se le denomina comúnmente como ‘event listener’ (LDTalent, 2021), y su función principal es, una vez definidos los eventos existentes que nos interesan, enviarlos al microservicio en Azure para almacenarlos y poder acceder a ellos más adelante cuando queramos analizarlos. De ahí el nombre del plugin, ‘*send_events*’.

Para que Moodle lo reconozca como plugin de tipo local debemos colocarlo en el siguiente directorio: '*\alber_moodle_data_data\local\send_events*'.

La estructura de archivos de nuestro plugin es la siguiente:

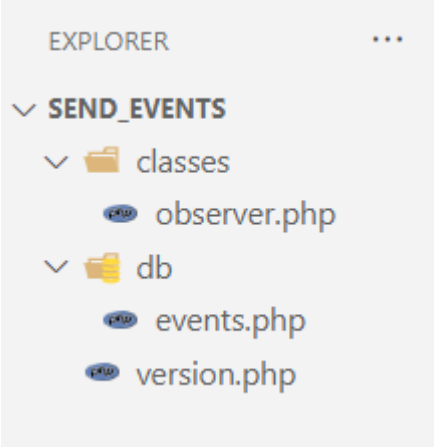

Figura 6: *Estructura de archivos del plugin 'send_events'*

- **version.php**: Este archivo contiene la información de la versión del plugin:

```
defined('MOODLE_INTERNAL') || die();
$plugin->version = 2023112408;
$plugin->requires = 2023041800;
$plugin->component = 'local_send_events';
```

Código de version.php

La línea de código que podemos ver al principio de cada función, '*defined('MOODLE_INTERNAL') || die()*', es una medida de seguridad que garantiza que los archivos de Moodle se ejecuten correctamente y no sean vulnerables a ataques de seguridad.

El plugin tiene varios atributos. El parámetro '*version*', que indica la versión del plugin y que tendremos que ir cambiando cada vez que queramos actualizar el plugin con nuevas funcionalidades, el parámetro '*requires*', que indica la versión mínima de Moodle donde se podrá ejecutar, y el parámetro 'component', que indica el nombre del plugin.

- **events.php**: En este archivo especificamos los eventos que nos interesan:

```
defined('MOODLE_INTERNAL') || die();
$observers = array(
    //Cuando se hace login
    array(
        'eventname' => '\core\event\user_loggedin',
        'callback' => 'local_send_events_observer::user_loggedin',
    ),
    //Cuando se sube una tarea
    array(
        'eventname' => '\mod_assign\event\submission_created',
        'callback' =>
'local_send_events_observer::mod_assign_submission_created',
    ),
```

```
    //Cuando se ve un documento
    array(
        'eventname' => '\mod_resource\event\course_module_viewed',
        'callback' =>
'local_send_events_observer::mod_resource_course_module_viewed',
    ),
    //Cuando se ha visitado un "external tool"
    array(
        'eventname' => '\mod_lti\event\course_module_viewed',
        'callback' =>
'local_send_events_observer::mod_lti_course_module_viewed',
    ),
    //Cuando se crea un módulo (crear tarea, subir documento, subir
external tool, etc)
    array(
        'eventname' => '\core\event\course_module_created',
        'callback' =>
'local_send_events_observer::course_module_created',
    ),
);
```

Código de events.php

Antes de continuar con la explicación del código, es importante recordar lo que se comentó en la introducción, y es que esto se trata de una prueba de concepto, por lo tanto, aunque estemos recogiendo todos estos tipos de eventos, en este proyecto nos vamos a limitar al análisis de los eventos pertenecientes a los de inicio de sesión. Podríamos haber considerado más, como los que se ven en el código, pero al final se decidió analizar únicamente los mencionados.

En el array '*$observers*' guardamos arrays con la información sobre los eventos que nos interesa 'escuchar' y las funciones que se deben llamar cuando se produce el evento. Dentro de los arrays de los eventos están sus propiedades. La primera, '*eventname'*, describe el nombre de dicho evento, y la segunda, '*callback'*, para especificar la función que se debe llamar cuando se produce el evento.

Por poner un ejemplo, cuando se produce el evento de inicio de sesión, '*\core\event\user_loggedin*', se llama a la función '*user_loggedin*' contenida en la clase '*local_send_events_observer*' (que está en el archivo '*observer.php*').

- **observer.php**: Este archivo contiene las funciones para todos los eventos especificados en '*events.php*'. Aquí es donde escribimos lo que sea que queramos que pase cuando un evento ocurra. En nuestro caso, enviarlos a un microservicio en Azure. Como todas las funciones correspondientes a los eventos hacen exactamente lo mismo, voy a mostrar ciertas partes del código para que no sea redundante:

```
public static function user_loggedin(\core\event\user_loggedin $event)
    {
        $event_data = $event->get_data();
        $event_data['domain'] = 'tfgalberto@tfg.es';
```

```
        self::send_event_to_api($event_data);
    }
```

Código que ejecutan todas las funciones de los eventos en observer.php (función de inicio de sesión).

Como ejemplo muestro la función de inicio de sesión. El bloque de código que contiene lo ejecutan todas las funciones que corresponden a los eventos. Cuando se produce un evento, la información de este se guarda en la variable '*$event*'. Entonces, para conseguir dicha información, usamos el método '*get_data()*' y lo guardamos en una variable llamada '*$event_data*'. A este array le asignamos la propiedad '*domain*' para, cuando enviemos los eventos al microservicio, podamos distinguir los eventos generados por este trabajo de otros.

Por último, se envía la información del evento a una función cuyo propósito es enviar los eventos al microservicio. Como dicha función reside dentro de la clase '*local_send_events_observer*', que es la clase donde están todas las funciones de los eventos, se usa '*self*', refiriéndose a que la función se encuentra dentro de la misma clase.

Como he dicho antes, para intentar evitar lo máximo posible la redundancia en el código que muestro, sustituiré el código del resto de funciones por una línea que diga '*[Mismo código que la función user_loggedin]*':

```
class local_send_events_observer
{
    public static function user_loggedin(\core\event\user_loggedin
$event)
    {
        $event_data = $event->get_data();
        $event_data['domain'] = 'tfgalberto@tfg.es';
        self::send_event_to_api($event_data);
    }
    public static function
mod_assign_submission_created(\mod_assign\event\submission_created
$event)
    {
        [Mismo código que la función user_loggedin]
    }
    public static function
mod_resource_course_module_viewed(\mod_resource\event\course_module_viewe
d $event)
    {
        [Mismo código que la función user_loggedin]
    }
    public static function
mod_lti_course_module_viewed(\mod_lti\event\course_module_viewed $event)
    {
        [Mismo código que la función user_loggedin]
    }
    public static function
course_module_created(\core\event\course_module_created $event)
    {
```

```
        [Mismo código que la función user_loggedin]
    }
```

Funciones de los eventos especificados en events.php

Las funciones de los eventos toman como argumento la variable '*$event*', que almacenará el evento que se haya producido.

A continuación, voy a mostrar por partes el código de la función '*send_event_to_api*':

```
public static function send_event_to_api($data)
```

No merece mucha explicación, simplemente tener en cuenta que toma como argumento los datos del evento y los guarda en una variable llamada '*$data*'.

```
// Solicitud HTTP Post a Azure
$ch = curl_init();
curl_setopt($ch, CURLOPT_URL, "{URL de la API donde enviamos eventos}");
curl_setopt($ch, CURLOPT_POST, 1);
curl_setopt($ch, CURLOPT_POSTFIELDS, json_encode($data));
curl_setopt($ch, CURLOPT_HTTPHEADER, array(
    'Content-Type: application/json',
    'Authorization: Bearer {Token autorización de la API Azure}
));
curl_exec($ch);
```

Parte del código de 'send_event_to_api'.

Este código se encarga de crear una solicitud HTTP de tipo POST al microservicio en Azure utilizando la biblioteca cURL. Con el método '*curl_setopt()*' configuramos los diferentes parámetros de la solicitud:

- **CURLOPT_URL:** Establece la URL del servicio web donde enviamos la solicitud.
- **CURLOPT_POST:** Al asignarle '1', indicamos que la solicitud será de tipo POST. Este tipo se usa para enviar datos a un servidor para que lo procese.
- **CURLOPT_POSTFIELDS:** Establece los datos que enviaremos en la solicitud. En este caso los codificamos como JSON usando la función '*json_encode()*'.
- **CURLOPT_HTTPHEADER:** Cabecera de la solicitud. Las solicitudes HTTP contienen una cabecera con diferentes campos, entre ellos '*Content-Type*', donde indicamos que el tipo de información que estamos enviando es JSON, y la clave de autorización en el campo '*Authorization*'.

Realizamos la solicitud con '*curl_exec()*'.

Se ha quitado del código tanto la URL del microservicio como el token de autenticación por motivos de seguridad.

```
// Verificar si hubo algún error en la formación de la solicitud
if (curl_errno($ch))
{
    var_dump('Error en la solicitud');
    die();
```

```
}
// Obtener código de la respuesta HTTP
$codigo_respuesta = curl_getinfo($ch, CURLINFO_HTTP_CODE);

// Cerramos la sesión
curl_close($ch);

if ($codigo_respuesta == 201)
{
    return true;
}
else
{
    var_dump("Codigo de respuesta: $codigo_respuesta");
    die();
}
```

Parte del código de 'send_event_to_api'.

Este bloque de código se encarga de verificar los errores que puedan ocurrir. Usamos '*curl_errno()*' para obtener el número de error de la solicitud. Si este es '0' no se produjo ningún error. Si en algún momento de la solicitud ha habido un error, ya sea porque no se ha podido realizar la solicitud, o el código de respuesta sea distinto a 201 (que indica que la petición se ha realizado con éxito), usamos el método '*var_dump()*' para imprimir en pantalla el error. Esto me sirve para saber si ha habido algún fallo o no, ya que, si consigo navegar por Moodle sin ningún tipo de interrupción, significará que todo se están enviando correctamente.

El código de respuesta mencionado se obtiene consultando el parámetro '*CURLINFO_HTTP_CODE*' de la solicitud con el método '*curl_getinfo()*'.

3.3. Análisis de los eventos

Antes de intentar enviar algún evento al microservicio, hay que crear un contenedor en Azure para poder almacenarlos. Gracias a tener un email de la UPCT, podemos crear una cuenta de Azure para estudiantes, que tiene acceso a más herramientas que una cuenta estándar de manera gratuita. Una vez creada, debemos crear una cuenta de almacenamiento a la que llamaremos '*tfgmoodle*'. Esta cuenta de almacenamiento reside en la nube y nos proporciona una solución para poder almacenar y acceder a datos no estructurados, como archivos, tablas, colas o blobs. Una de sus ventajas es que los datos en una cuenta de almacenamiento son duraderos y altamente disponibles, seguros y escalables (Microsoft, 2023).

Dentro de esta cuenta de almacenamiento, crearemos un contenedor llamado '*capture*'.

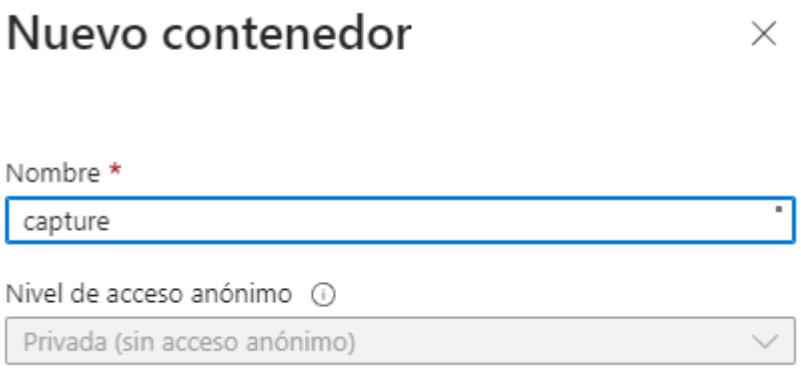

Figura 7: *Creación del contenedor 'capture'*

Al crearlo, la manera de que los tutores puedan subir desde el microservicio los eventos al contenedor es compartiéndoles una cadena de conexión perteneciente a la cuenta de almacenamiento, la cual les permitirá conectarse al contenedor y subir los eventos.

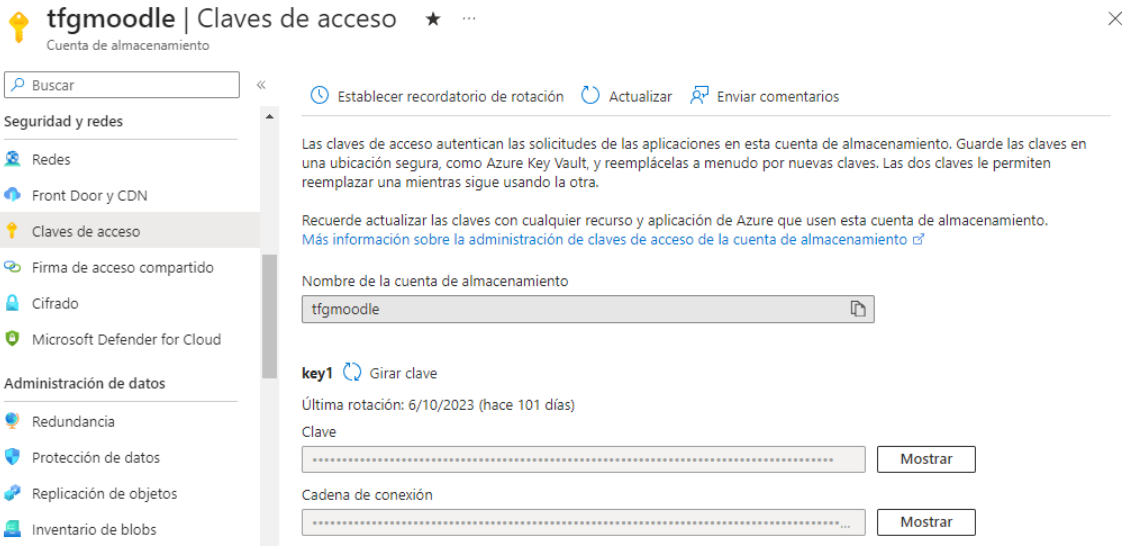

Figura 8: *Panel de Azure donde obtenemos la cadena de conexión de la cuenta de almacenamiento*

Una vez compartido con los tutores la cadena de conexión, lo que harán será subir los eventos del microservicio al contenedor en archivos con formato '*avro*'. Avro es un sistema de serialización de datos que se utiliza para almacenar y transmitir datos en diferentes lenguajes de programación, y es el que usamos para almacenar los eventos. Más adelante veremos cómo se forman con más detenimiento.

Con esto preparado, podemos empezar a enviar los eventos al microservicio. Conseguimos mandar bastantes de ellos para su posterior análisis y esperamos a que los tutores suban los archivos avro al contenedor:

Figura 9: *Contenedor 'capture' en Azure donde almacenamos los eventos en formato avro*

El siguiente paso es descargarlos en Python para poder almacenarlos en DataFrames y así analizarlos. Esta parte en concreto es sencilla porque el código responsable para la descarga de los archivos avro nos lo proporciona los directores del TFG, y se tratan principalmente de dos archivos:

- **container.py**: Este archivo lo usamos para listar todos los avro dentro del contenedor '*capture*'.
- **events.py**: Este archivo tiene la función necesaria para descargar dichos archivos avro.

```
from dotenv import load_dotenv
import os

load_dotenv()

alberto_storage_connection_str =
os.environ["ALBERTO_STORAGE_CONNECTION_STR"]

from square_analytics.container import Container
from square_analytics.events import Events

# Listamos los blobs del contenedor capture
capture_container = Container("capture", alberto_storage_connection_str)
print(capture_container.list_blobs())

# Descargamos los eventos y pasamos a DataFrame
events = Events(capture_container)
eventsdf = events.dataframe
```

Código para listar y descargar los eventos del contenedor 'capture'

Las variables de entorno están almacenadas en un archivo del directorio actual llamado '*.env*', encargado de almacenar dichas variables. Entonces, lo primero que debemos hacer antes de nada es cargar las variables de entorno con el método '*load_dotenv()*'. Una vez cargadas, usamos '*os.environ*', un diccionario que contiene todas las variables de entorno cargadas. La cadena de conexión está almacenada en la variable de entorno de nombre

'*ALBERTO_STORAGE_CONNECTION*_STR'. Almacenamos la cadena en la variable '*alberto_storage_connection_str*'.

A continuación, importamos las funciones, imprimimos una lista de los blobs que hay en el contenedor y descargamos los eventos, convirtiéndolos a dataframe y almacenándolos en la variable '*eventsdf*'.

En el primer argumento de la función '*Container*' debemos poner el nombre del contenedor y en el segundo la variable que contiene la cadena de conexión. Así mismo, a la función '*Events*' debemos pasarle el contenedor.

```
['upctevents/devevents/0/2023/10/09/17/43/05.avro', 'upctevents/devevents/0/2023/10/10/16/13/05.avro', 'upctevents/devevents/0/2023/11/09/07/
Downloaded a non empty blob: upctevents/devevents/0/2023/10/09/17/43/05.avro
Downloaded a non empty blob: upctevents/devevents/0/2023/10/10/16/13/05.avro
Downloaded a non empty blob: upctevents/devevents/0/2023/11/09/07/28/05.avro
Downloaded a non empty blob: upctevents/devevents/0/2023/11/21/09/13/05.avro
Number of downloaded events: 130
```

	eventname	component	action	target	objecttable	objectid	crud	edulevel	contextid
0	\core\event\user_loggedin	core	loggedin	user	user	2	r	0	1
1	\core\event\user_loggedin	core	loggedin	user	user	2	r	0	1
2	\core\event\user_loggedin	core	loggedin	user	user	2	r	0	1
3	\core\event\user_loggedin	core	loggedin	user	user	2	r	0	1
4	\core\event\user_loggedin	core	loggedin	user	user	2	r	0	1
...	...	...	...	...	...	...	...	...	...
125	\core\event\user_loggedin	core	loggedin	user	user	2	r	0	1
126	\core\event\user_loggedin	core	loggedin	user	user	2	r	0	1
127	\core\event\user_loggedin	core	loggedin	user	user	8	r	0	1
128	\assignsubmission_file\event\submission_created	assignsubmission_file	created	submission	assignsubmission_file	13	c	2	75
129	\core\event\user_loggedin	core	loggedin	user	user	2	r	0	1

130 rows × 18 columns

Figura 10: *Extracto del listado de los blobs y el dataframe donde almacenamos los eventos*

3.3.1. Tablas de la base de datos 'estáticas'

Uno de los grandes objetivos de la explotación de eventos generados por Moodle para analíticas de aprendizaje es evitar lo máximo posible las llamadas a las bases de datos de Moodle (Examulator, 2022), es por ello por lo que las tablas de bases de datos con las que nos podemos permitir trabajar son con aquellas que apenas cambien a lo largo de un curso académico, porque de lo contrario, no tendría sentido usar los eventos para sacar información. Por ello, en vez de realizar las llamadas a dichas bases de datos las haríamos a copias de esas bases de datos, de forma que no afecte de ninguna manera al rendimiento de Moodle.

Ejemplos de tablas de bases de datos que no cambian mucho a lo largo de un curso académico son:

- **mdl_user**: Lista de usuarios registrados.
- **mdl_course**: Lista de cursos.
- **mdl_context**: Información sobre los contextos en Moodle.
- **mdl_role_assignments:** Roles de los usuarios en contextos específicos.

Para realizar los análisis en este trabajo no es necesario complicarnos más de lo necesario porque no entra dentro de los objetivos de este, es por ello por lo que vamos a jugar alrededor de los eventos pertenecientes a los inicios de sesión, que son la gran mayoría de los generados, y nos permiten usar las tablas de datos antes mencionadas.

username	password	idnumber	firstname	lastname	email
guest	$2y$10$CbISmTU8xM7gPyiU0Q/gz.gm9yZSB.EjPJCqPsZ...		Guest user		root@localhost
alber	$2y$10$Ic1FLC4bDmgUngx1d317IOf/iBC4r/.Ftas3fvL...		Admin	User	user@example.com
n.1696254790	$2y$10$IC5RQwIGVSyHlj3aCOALeeOQ/cYkRvjyIcP3puE...		Alumno	1	f6b16b945f3a595d0f4d0b9270eb0cb5
profesor1	$2y$10$k6nb94FTP7p3dmobD4WydeZVFcLGBYINrtS/qAX...		Tomás	Martínez	tomas.martinez@mail.com
alumno1	$2y$10$8fQQAF1Q6x0v1QQMivEreeiTLCYCum/s88dmAx3...		Alberto	Ros	alberto.ros@mail.com
alumno2	$2y$10$ECMy3WDgiGzp2COtmKipjerRRBoMnRtQMbiG87U...		David	López	david.lopez@mail.com
alumno3	$2y$10$yHtEp0O39KiwS/3sC/aZF.CGtvZjYg0yAMF2vsh...		Fran	Hernández	fran.hernandez@mail.com
alumno4	$2y$10$pBLacGcYcL5yFiCM/VhGIuuUItzPUybI8r8g5hN...		Juan	Redondo	juan.redondo@mail.com

Figura 11: *Extracto de la tabla de usuarios, 'mdl_user'*

La tabla de usuarios nos servirá para sacar información sobre los alumnos, como los nombres, apellidos o sus correos.

id	category	sortorder	fullname	shortname	idnumber	summary
1	0	1	Moodle Docker TFG	MDTFG		Moodle powered by Bitnami
2	1	10002	Introducción a Data Science	IDS		Curso de Introducción a Data Science...

Figura 12: *Extracto de la tabla de cursos, 'mdl_course'*

La tabla de cursos nos servirá principalmente para sacar los IDs de los cursos que existen, para así usarlos en la subida de los eventos analizados más adelante.

	id	contextlevel	instanceid	path	depth	locked
0	1	10	0	/1	1	0
1	2	50	1	/1/2	2	0
2	3	40	1	/1/3	2	0
3	4	30	1	/1/4	2	0
4	5	30	2	/1/5	2	0
5	7	80	2	/1/7	2	0

Figura 13: *Extracto de la tabla de contextos, 'mdl_context'*

La tabla '*mdl_context*' puede considerarse algo confusa, pero básicamente contiene información sobre los contextos dentro de Moodle, como, por ejemplo, los cursos.

id	roleid	contextid	userid	timemodified	modifierid	component	itemid	sortorder
1	3	22	2	1696431929	2		0	0
2	3	22	4	1696432073	2		0	0
3	5	22	5	1696432084	2		0	0
4	5	22	6	1697196229	2		0	0
5	5	22	7	1697196229	2		0	0
6	5	22	8	1697196229	2		0	0
7	3	81	2	1704814955	2		0	0
8	5	81	5	1704814981	2		0	0
10	3	81	4	1705601851	2		0	0

Figura 14: *Extracto de la tabla de roles de usuario, 'mdl_role_assignments'*

Esta tabla nos va a ser extremadamente útil cuando obtengamos los usuarios matriculados a un curso en específico. Sin ella, aparte de los estudiantes, obtendríamos todos los demás usuarios matriculados al curso, que pueden ser desde administradores hasta profesores. De esta tabla, podemos deducir que el ID del rol correspondiente a los estudiantes es 5.

Podemos asegurarnos echándole un vistazo a la tabla '*mdl_role*', donde podemos confirmar la deducción anterior:

	id	name	shortname	description	sortorder	archetype
0	1		manager		1	manager
1	2		coursecreator		2	coursecreator
2	3		editingteacher		3	editingteacher
3	4		teacher		4	teacher
4	5		student		5	student
5	6		guest		6	guest
6	7		user		7	user
7	8		frontpage		8	frontpage

Figura 15: *Tabla de la base de datos con los roles, 'mdl_role'*

3.3.2. Obtención de datos de los eventos

Vamos a sacar datos provenientes de los eventos y presentárselos tanto a profesores como a alumnos. Para la obtención de los datos para profesores será necesario hacer llamadas de datos a las bases de datos 'estáticas' antes mencionadas. Para alumnos eso no será necesario.

3.3.2.1. Datos para profesores

Si queremos presentarle a un profesor la lista de alumnos de un curso determinado que llevan por ejemplo una semana sin iniciar sesión, y lo intentamos únicamente a través de la información que nos proporcionan los eventos es imposible, ya que no hay eventos que

nos permitan conocer información sobre alumnos matriculados a un curso concreto. Es por eso por lo que hay que hacer una llamada a las bases de datos estáticas antes mencionadas, y de ahí contrastar con los eventos recogidos.

Lo primero que debemos hacer es obtener, para cada curso, la lista de alumnos matriculados al mismo, y para ello podemos empezar con la conexión a MariaDB:

```
import mariadb

# Conexión con la base de datos MariaDB
conn = mariadb.connect(
    user="root",
    host="localhost",
    database="bitnami_moodle")
```

Código que muestra la conexión a la base de datos

Usamos la biblioteca '*mariadb*' de Python para conectarnos a la base de datos. Su función '*connect()*' nos sirve para establecer la conexión y le tenemos que pasar los parámetros del usuario con el que queremos conectarnos, el host, que será local, y el nombre de la base de datos, cuyo nombre es '*bitnami_moodle*'.

```
import pandas as pd

cur = conn.cursor()
courseid = 2

cur.execute(f"
      SELECT u.id, u.firstname, u.lastname, u.email
      FROM mdl_user u
      JOIN mdl_role_assignments ra ON u.id = ra.userid
      JOIN mdl_context cx ON ra.contextid = cx.id
      AND cx.contextlevel = 50 AND ra.roleid = 5
      JOIN mdl_course c ON c.id = cx.instanceid
      WHERE c.id = {courseid}"
)

results = cur.fetchall()
df_users_course = pd.DataFrame(results, columns=['userid', 'firstname',
'lastname', 'email'])
```

Código que realiza la consulta a la base de datos

Creamos un objeto cursor. Este nos permitirá justo a continuación ejecutar consultas SQL y recuperar resultados de la base de datos.

Con '*cur.execute()*' realizamos la consulta. En el SELECT indicamos que queremos obtener los campos '*id*', '*firstname*', '*lastname*' y '*email*' de la tabla '*mdl_user*'. Hacemos tres JOIN, uno que combina '*mdl_user*' y '*mdl_role_assignments*' para conseguir los usuarios con rol de estudiante, otro JOIN que combina con '*mdl_context*' y '*mdl_role_assignments*' para identificar las asignaciones de roles de estudiantes en los contextos de cursos, y el ultimo JOIN que combina '*mdl_course*' y '*mdl_context*' para

vincular los cursos con los contextos, filtrando los que tienen rol de estudiante con '*ra.roleid=5*' y el contexto de los cursos con '*cx.contextlevel=50*', y así, al hacer el WHERE donde indicamos el ID del curso, sacamos la información deseada.

Guardamos los resultados de la consulta con el método '*fetchall()*' y lo usamos para crear un dataframe llamado '*df_users_course*' que usaremos a continuación. El dataframe con el resultado de la consulta es el siguiente:

	userid	firstname	lastname	email
0	5	Alberto	Ros	alberto.ros@mail.com
1	6	David	López	david.lopez@mail.com
2	7	Fran	Hernández	fran.hernandez@mail.com
3	8	Juan	Redondo	juan.redondo@mail.com

Figura 16: *Usuarios matriculados al curso con ID=2*

Con esta lista, podemos contrastar con la lista de eventos:

```
evento = '\\core\\event\\user_loggedin'
week_ago = int((pd.Timestamp.now(tz='UTC') -
pd.Timedelta(days=7)).timestamp())

df_logins =
      eventsdf.loc[(eventsdf['eventname'] == evento) &
      (eventsdf['timecreated'] > week_ago)]

df_alumnos_sin_logear = df_users_course.loc[
      ~df_users_course['userid'].isin(
      df_logins['userid'].astype('int64')
      )
]
```

Código que muestra la obtención de los datos para los profesores

Como ya vimos antes, el parámetro '*timecreated*' está en formato timestamp Unix, entonces para conseguir el tiempo actual y poder comparar, usamos el método '*Timestamp().now()*'. A este tiempo le restamos exactamente los segundos que tienen una semana. Así obtenemos el timestamp de hace una semana y lo guardamos en la variable '*week_ago*'.

Lo siguiente es filtrar del dataframe de los eventos los que son de inicio de sesión y los que se produjeron hace menos de una semana. Con esto obtenemos los inicios de sesión que se han producido la última semana.

Por último, del dataframe de los alumnos matriculados en el curso, localizamos los que no están en la tabla que iniciaron hace menos de una semana. Los que no coincidan, serán el resultado:

	userid	firstname	lastname	email
0	5	Alberto	Ros	alberto.ros@mail.com
1	6	David	López	david.lopez@mail.com

Figura 17: *Lista de usuarios sin iniciar sesión en una semana*

Este dataframe, '*df_alumnos_sin_logear*', lo usaremos más adelante cuando queramos guardar la información en diccionarios.

3.3.2.2. Datos para alumnos

A alumnos les mostraremos a alumnos información muy poco relevante como la cantidad de veces que ha iniciado sesión, la fecha de su ultimo inicio de sesión y su nombre de usuario. La mayoría de esta información es posible conseguirla únicamente a través de la lista de eventos.

```
event = '\\core\\event\\user_loggedin'
user_loginsdf = eventsdf.loc[(eventsdf['eventname'] == event) &
      (eventsdf['userid'] == user_id)]

n_logins = user_loginsdf.shape[0]

last_login_day = pd.to_datetime(
        user_loginsdf.iloc[-1]['timecreated'], unit='s'
).strftime('%d-%m')

logins = pd.DataFrame(
        {
            "n_logins": [n_logins],
            "last_login_day": [last_login_day]
        }
    )
```

Código que muestra la obtención de los datos de los alumnos

Primero obtenemos la información de inicio de sesión de un usuario concreto y lo guardamos en '*user_loginsdf*'. De aquí sacamos el número de inicios de sesión, que se obtiene con el número de filas que tiene el dataframe. El último día que se inició sesión se consigue del parámetro '*timecreated*' de la última fila del dataframe.

Tanto el número de inicios de sesión como el último día que inició sesión lo guardamos en un dataframe llamado '*logins*', que como en el caso del dataframe de alumnos sin iniciar sesión en una semana, lo usaremos más adelante cuando queramos guardar la información en diccionarios. El nombre de usuario lo conseguimos más tarde cuando vayamos a crear los archivos avro.

La razón de que no haya información más interesante para mostrar a los alumnos es que hasta el momento no hay ningún evento que nos indique, por ejemplo, la creación de una tarea como tal, por eso no hay manera de obtener la lista de alumnos que no han subido una tarea. Lo importante es demostrar que podemos sacar información de los eventos.

En un futuro, cuando Moodle tenga más tipos de eventos, podremos sacar información mucho más relevante tanto para profesores como para alumnos.

Antes de continuar con el siguiente punto, es importante puntualizar que una parte de los datos analizados están destinados a ser recibidos por los profesores, que los visualizarán dentro de los cursos, y los datos analizados destinados a los alumnos no están relacionados con un curso en concreto, por ello, los datos analizados de los profesores los almacenaremos en directorios correspondientes a los cursos, y los datos para alumnos los almacenaremos en directorios individuales para cada alumno.

3.3.3. Diccionarios y archivos avro de cursos y alumnos

El siguiente paso es crear el diccionario en Python, donde introduciremos los datos relevantes que hemos obtenido de los eventos para a continuación, convertir estos diccionarios en datos binarios Avro.

En Python, los diccionarios son estructuras de datos que nos permiten almacenar todo tipo de información, ya sean cadenas de texto, caracteres, números, listas o incluso otros diccionarios (Python, s. f.). La manera de localizar los datos dentro del diccionario es a través de las claves únicas que les asignan, y al no tener un tamaño predefinido, será más o menos grande según las necesidades de la aplicación.

En cambio, Avro (Microsoft, 2023) es un formato de datos que se utiliza para serializar y deserializar datos estructurados. Se usa normalmente en aplicaciones de big data para intercambiar datos entre sistemas, y utiliza un esquema para definir la estructura de los datos, lo que permite su serialización y deserialización de forma eficiente entre diferentes lenguajes de programación.

Antes de introducir los datos en el diccionario, hay que definir mediante código el esquema avro que vamos a seguir, tanto para profesores como para alumnos.

3.3.3.1. Esquema avro de los cursos

El diccionario debe guardar información sobre los alumnos que no han iniciado sesión en una semana, y dicha información tiene que ser relevante para que el profesor los identifique fácilmente. Los datos almacenados en el dataframe antes visto, '*df_alumnos_sin_logear*', contiene el nombre, apellido y email de cada alumno. Esta información es más que suficiente para la identificación de los alumnos.

El esquema avro para los cursos será el siguiente:

```
avro_course_schema = {
    "name": "aggregate_open_repository",
    "namespace": "aggregate_open_repository_namespace",
    "type": "record",
    "fields": [
        {
            "name": " students_not_logged_week",
            "type": {
                "name": " students_not_logged_week_columns",
                "namespace": "columns",
                "type": "array",
```

```
                "default": [],
                "items": {
                    "name": " students_not_logged_week_items",
                    "namespace": "items",
                    "type": "record",
                    "fields": [
                        {"name": "firstname", "type": "string"},
                        {"name": "lastname", "type": "string"},
                        {"name": "email", "type": "string"}
                    ],
                },
            },
        },
    ],
}
```

Esquema avro de los cursos

Queremos crear un registro de alumnos, y que cada alumno tenga sus campos. Lo primero es, en caso de querer tener más de un registro (como veremos en el caso del esquema avro para alumnos), crear, por así decirlo, un registro de registros, al que llamaremos '*aggregate_open_repository*'. Este registro de registros tendrá en su interior otro registro llamado '*students_not_logged_week*'. Este contendrá un array que servirá para almacenar múltiples alumnos, y cada elemento de dicho array serán registros donde almacenaremos los campos de los alumnos.

La mayor diferencia entre un registro y un array es que, en el caso de los registros, cada uno de sus campos tienen un nombre único y un tipo de datos asociado, aparte de no tener filas ni columnas. Un array en cambio se utiliza para coleccionar elementos del mismo tipo de manera ordenada con sus índices.

El registro '*students_not_logged_week*' tiene un nombre lo suficientemente explicativo para que no haya confusiones en caso de haber más de uno, aunque en nuestro caso no haga falta. Los campos de dicho registro son:

- **firstname**: Nombre del alumno. Se obtiene de la llamada a la base de datos de los usuarios, del parámetro '*firstname*'.
- **lastname**: Apellido del alumno. Se obtiene igual que el anterior, esta vez del parámetro '*lastname*'.
- **email**: Correo del alumno. Igual que los anteriores, pero esta vez del parámetro '*email*'.

Los tres campos serán cadenas de texto. Una vez definido el esquema, toca rellenar el diccionario con la información de los alumnos.

```
# Llamada a la tabla de los cursos para obtener sus IDs
cur.execute("SELECT id FROM mdl_course")
results = cur.fetchall()
coursesdf = pd.DataFrame(results, columns=['id'])
courses_id = coursesdf['id'].unique()
```

```
# Creamos un diccionario por cada curso existente
for course_id in courses_id:
    cursos_dict = {}
    cursos_dict["students_not_logged_week"] =
df_alumnos_sin_logear.to_dict("records")
      ...
```

Código que rellena los diccionarios de los cursos

Primero sacamos los cursos que hay haciendo una llamada a la base de datos de los cursos. En vez de obtener múltiples campos como en casos anteriores, solo obtenemos los IDs. No debería ocurrir que haya más de una fila con el mismo ID, pero con el método '*unique()*' nos aseguramos. Guardamos los IDs en la variable '*courses_id*' para poder hacer el bucle que itere entre los cursos que hay para crear sus diccionarios.

En cada iteración del bucle, creamos un diccionario nuevo y convertimos el dataframe antes visto, '*df_alumnos_sin_logear*', en un diccionario con '*to_dict()*'. A esta función le pasamos el argumento '*records*' para especificar el formato de salida de esta. Lo que devuelve es una lista de diccionarios, cada uno representando una fila de datos en el dataframe.

Es importante puntualizar que el bloque del código que se encarga de convertir los diccionarios en archivos avro y subirlos a Azure se encuentra dentro del bucle mostrado. Más adelante lo veremos.

3.3.3.2. Esquema avro de los alumnos

Para crear el esquema del diccionario de los alumnos vamos a tener un registro y un campo. El registro almacenará información sobre los inicios de sesión, y el campo almacenará el nombre del alumno correspondiente:

```
avro_user_schema = {
    "name": "aggregate_open_repository",
    "namespace": "aggregate_open_repository_namespace",
    "type": "record",
    "fields": [
        {
            "name": "logins",
            "type": {
                "name": "logins_columns",
                "namespace": "columns",
                "type": "array",
                "default": [],
                "items": {
                    "name": "logins_items",
                    "namespace": "items",
                    "type": "record",
                    "fields": [
                        {"name": "n_logins", "type": "int"},
                        {"name": "last_login_day", "type": "string"}
                    ],
                },
```

```
            },
        },
        {"name": "firstname", "type": "string"},
    ],
}
```

Esquema avro de los alumnos

El registro '*logins*' tiene dos campos:

- **n_logins**: El número de inicios de sesión. Se consigue sacando el número de eventos que tenga el alumno de inicio de sesión.
- **last_login_day**: El ultimo día que el alumno inició sesión. Se obtiene del parámetro '*timecreated*' de los eventos.

Además, almacenamos el campo '*firstname*' independiente del registro anterior:

- **firstname**: El nombre del alumno. Se saca con una llamada a la base de datos de los usuarios.

Al igual que con el diccionario de los cursos, metemos los parámetros en el diccionario:

```
# Obtenemos los IDs únicos de los usuarios que han producido eventos
users_loginsdf = eventsdf.loc[(eventsdf['eventname'] == event)]
users_id = users_loginsdf['userid'].unique()

# Rellenamos el diccionario de cada usuario
for user_id in users_id:

    cur.execute(f"SELECT u.firstname
                  FROM mdl_user u
                  WHERE u.id = {user_id}")
    results = cur.fetchall()
    firstname = results[0][0]

    users_dict = {}
    users_dict["logins"] = logins.to_dict("records")
    users_dict["firstname"] = firstname
...
```

Código que rellena el diccionario de los alumnos

Para poder crear los diccionarios de cada alumno, debemos obtener primero sus IDs. Esto lo hacemos buscando los alumnos que crearon eventos de inicio de sesión.

En cada iteración del bucle, primero conseguimos el nombre del usuario para poder almacenarlo en el diccionario nuevo que creamos justo a continuación, y convertimos el dataframe antes visto, '*logins*', en un diccionario. Al registro que contiene el nombre simplemente le asignamos el nombre del usuario sacado de la llamada a la base de datos de usuarios.

Al igual que antes, el código responsable de convertir los diccionarios en archivos avro y subirlos a Azure lo omitimos para enseñarlo a continuación.

3.3.3.3. Subida de archivos avro al contenedor

Previo paso a subir los archivos, tenemos que crear el contenedor en la cuenta de almacenamiento en Azure que estamos usando para este trabajo. Lo único que tenemos que hacer es dirigirnos a la cuenta de almacenamiento que hemos denominado '*tfgmoodle*' y al igual que cuando creamos el contenedor 'capture', creamos un contenedor llamado '*aggregate*'.

Una vez creado, lo siguiente es convertir los diccionarios en archivos avro y subirlos al contenedor. El procedimiento es exactamente el mismo para ambos casos. Creamos el archivo avro con la información del diccionario y lo subimos al contenedor, así por cada curso y alumno que haya.

```
from azure.storage.blob import BlobServiceClient
import fastavro

# Nombre del contenedor
container_name = "aggregate"
# Ruta del archivo .avro (relativa desde este archivo python)
file_path = "aggregate.avro"
# Nombre del archivo .avro que vamos a subir al contenedor
file_name = "aggregate.avro"

# Establecemos conexión con Azure
blob_service_client =
BlobServiceClient.from_connection_string(alberto_storage_connection_str)

for course_id in courses_id:
...
    # Directorio en el que se cargará el archivo .avro en el contenedor
    directory_name = f"courses/{course_id}/"

    blob_client = blob_service_client.get_blob_client(
      container=container_name,
      blob=directory_name + file_name)

    # Creamos el archivo avro en el directorio actual
    with open(f"aggregate.avro", "wb") as f:
        fastavro.writer(f, avro_course_schema, [cursos_dict])

    # Cargar el archivo .avro en el contenedor
    with open(file_path, "rb") as data:
        blob_client.upload_blob(data, overwrite=True)
```

Ejemplo subida archivos avro de los cursos

Primero definimos el nombre del contenedor al que vamos a subir los archivos, el directorio relativo local donde se encuentran los archivos avro creados, y el nombre que tendrán dichos archivos en el contenedor.

A continuación, usamos la biblioteca '*azure.storage.blob*' para crear un objeto de tipo '*BlobServiceClient*'. Este nos servirá para establecer la conexión al contenedor con la cadena de conexión '*alberto_storage_connection_str*'. Esta es la misma variable de entorno que vimos cuando subimos los eventos porque la cuenta de almacenamiento que estamos usando en ambos casos es la misma, '*tfgmoodle*'.

Ingresamos al bucle, y con el esquema definido y el diccionario creado para el curso contenido en '*course_id*', definimos primero el directorio del contenedor en el que subiremos el archivo avro. Este va cambiando en cada iteración, siendo esta la razón de porque lo hemos definido dentro del bucle y no fuera.

Definido el directorio, usamos '*get_blob_client()*' para realizar operaciones con el blob donde vamos a almacenar el archivo. A esta función se le especifica el contenedor y el blob.

Hay que aclarar que en Azure no existe la estructura de carpetas que comúnmente conocemos, sino que se usan blobs. Los blobs son una solución de almacenamiento de objetos de Microsoft para la nube que esta optimizada para el almacenamiento de cantidades masivas de datos no estructurados (Microsoft, 2023). Para simular la estructura de carpetas, se deben usar nombres de blob que incluyan barras diagonales (/). En nuestro caso, a la hora de subir un aggregate.avro hay que especificar que el blob al que hay que subir el archivo es una combinación del directorio más el nombre que tendrá el archivo. Por ejemplo, en el caso de los cursos el blob es: *'courses/{id_curso}/aggregate.avro'*.

Lo siguiente que hacemos es usar la sintaxis '*with open()*' de Python. Esta sintaxis sirve para abrir y cerrar archivos de manera segura. El primer argumento sirve para definir el nombre del archivo con el que queremos trabajar y el segundo, que en este caso es '*wb*', sirve para crear el archivo en el directorio actual.

Dentro de '*with open()*' usamos la función '*writer*' de la biblioteca '*fastavro*' para crear el archivo avro. A dicha función se le debe pasar tanto el esquema avro como el diccionario.

Con el archivo avro creado, usamos de nuevo la sintaxis '*with open()*', pero esta vez para subir el archivo. Con el argumento '*rb*' leemos el archivo y lo subimos con el método '*upload_blob()*' del objeto que hemos mencionado anteriormente. Le pasamos como argumento el contenido del archivo avro y especificamos que queremos sobrescribir los archivos avro que tengan el mismo blob cada vez que subamos los archivos.

Por su parte, el código que genera y sube los archivos de los alumnos apenas cambia respecto al de los cursos. El cambio más significativo es el directorio al que vamos a subir los archivos, que es la línea '*directory_name = f"users/{user_id}/"*'.

En la página de Azure el contenido del contenedor se ve esta manera:

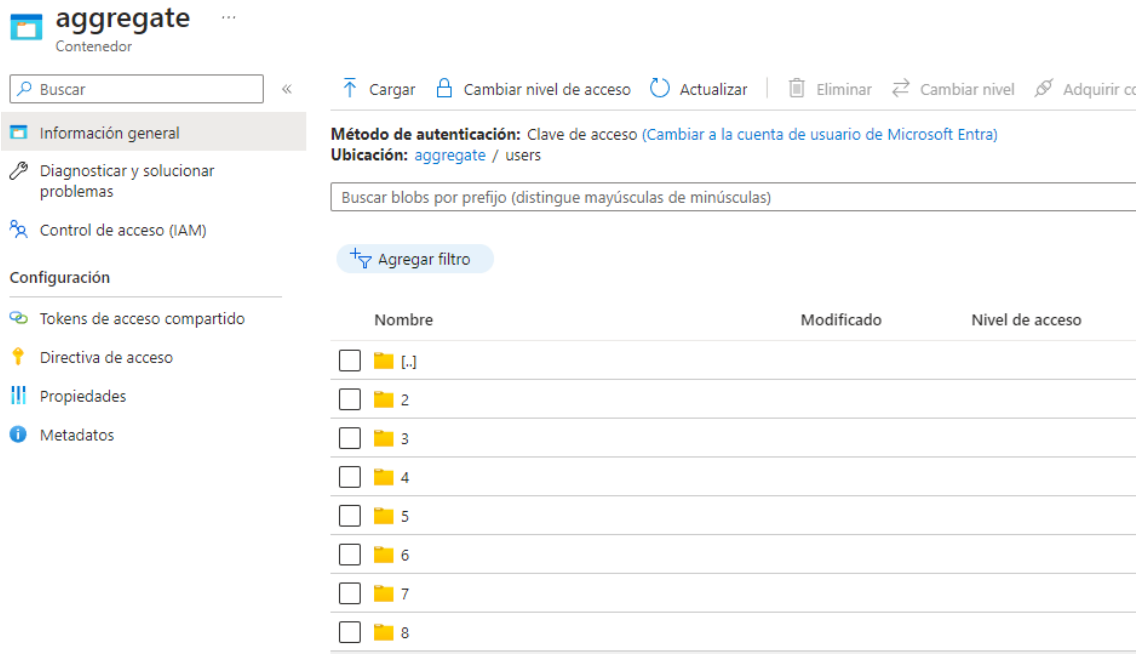

Figura 18: *Contenido de la carpeta users del contenedor 'aggregate'*

3.4. Creación de endpoints en Azure

Para poder obtener la información de los archivos avro, vamos a crear dos 'endpoints', uno será específico para los cursos, y otro para los alumnos.

Un endpoint es una URL que se utiliza para interactuar con un servicio web. En nuestro caso dicho endpoint va a ser de tipo GET. Este devolverá una respuesta HTML que contiene la información que le pidamos. En Azure tenemos la posibilidad de crear dichos endpoints, y para ello primero tenemos que crear una función de Azure que nos permita exponer los endpoints.

3.4.1. Función Azure

Esta función podemos crearla a través de VSCode con la extensión '*Azure Functions*'.

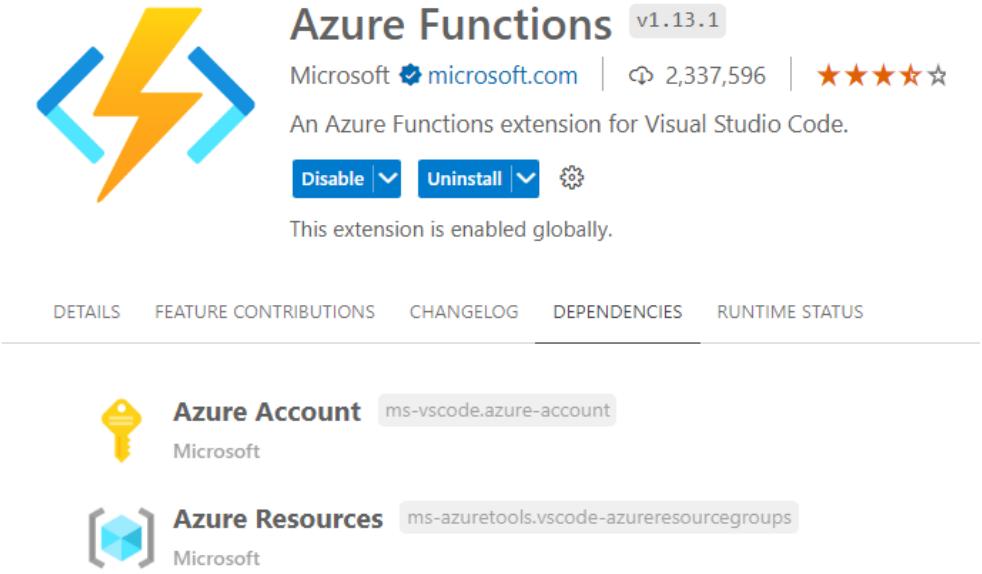

Figura 19: *Extensión 'Azure Functions' en Visual Studio Code.*

Una vez instalada, seleccionamos '*Azure Functions: Crear nueva función*' y esperamos a que se cree. Esto lo único que hace es crear lo que serían los archivos necesarios para su implementación en Azure, ahora tenemos que crear los endpoints.

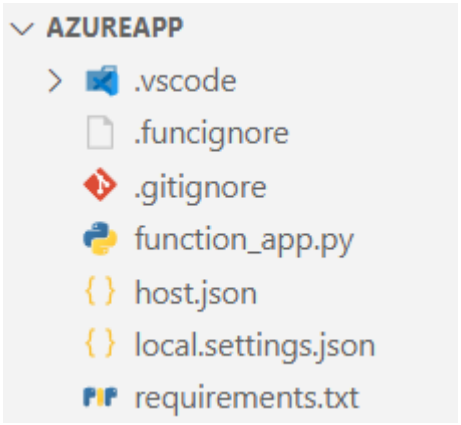

Figura 20: *Archivos generados por VSCode de la función Azure*

De estos archivos, los más relevantes son:

- **function_app.py**: En este archivo irán los endpoints que expondremos.
- **local.settings.json**: Este archivo contiene las variables de entorno de la aplicación.
- **requirements.txt**: Aquí especificamos las versiones de los paquetes que instalaremos en Python.

```
{
  "IsEncrypted": false,
  "Values": {
    "AzureWebJobsStorage": "{Cadena de conexión con contenedor Azure}",
    "FUNCTIONS_WORKER_RUNTIME": "python",
    "AzureWebJobsFeatureFlags": "EnableWorkerIndexing"
  }
```

```
}
```

local_settings.json

La única variable de entorno que vamos a configurar es '*AzureWebJobsStorage*'. Es exactamente la misma cadena de conexión que la que vimos al inicio para conectarnos a la cuenta de almacenamiento '*tfgmoodle*', '*alberto_storage_connection_str*'.

```
azure-functions
azure-storage-blob==12.18.2
fastavro==1.8.4
```

requirements.txt

Con las versiones más recientes tanto del paquete '*azure-storage-blob*' y '*fastavro*' saltaban errores a la hora de implementar la función Azure, por lo que especificamos versiones estables.

Cuando VSCode crea los archivos, '*function_app.py*' tiene por defecto un endpoint de prueba, llamado '*HttpExample*' para comprobar que la aplicación se implementa correctamente en Azure antes de pasar a crear nuestros endpoints personalizados.

Este endpoint toma un parámetro llamado 'name'. Si a la hora de consultar especificamos el valor de dicho parámetro, el endpoint nos devolverá un mensaje personalizado que contenga dicho valor, y si no lo pasamos, nos devuelve otro mensaje distinto.

3.4.2. Respuesta HTML de los endpoints

Antes de ponernos con los endpoints en sí, tenemos que pensar qué formato va a tener la respuesta HTML que devolverán.

Para ello, primero creamos un archivo llamado '*tfg_functions.py*' que contendrá dos funciones, una para el endpoint de los profesores, '*create_course_html*', y otra para el endpoint de los alumnos, '*create_student_html*'. Ambas funciones toman como argumento la información de los archivos avro en formato lista.

Para los cursos, queremos que el profesor vea una lista con la información de los alumnos, por lo que haremos un bucle que irá iterando por los alumnos existentes en la lista y así construiremos la respuesta HTML.

```
def create_course_html(course_info: list):

    html = '<p>¡Bienvenido de nuevo!</p>'
    html += '<p>Los alumnos matriculados en la asignatura sin iniciar
sesión '
    html += 'desde hace más de una semana son:</p>'
    for alumno in course_info:
        for login in alumno['students_not_logged_week']:
            nombre = login['firstname']
            apellido = login['lastname']
            email = login['email']
            html += f'<ul><li>{nombre} {apellido} | {email}</li></ul>'
    html += '<p>Ahí tiene sus correos por si necesita contactarles y
preguntar qué sucede. 😉</p>'
```

```
    return html
```

create_course_html, función que devuelve la respuesta HTML de los cursos

Para el caso de los alumnos no hará falta iterar en un bucle:

```
def create_student_html(student_info: list):

    logins = student_info[0]['logins']
    n_logins = logins[0]['n_logins']
    last_login_day = logins[0]['last_login_day']
    studentname = student_info[0]['firstname']

    html = f'<p>¡Bienvenido de nuevo {studentname}!</p>'

    html += f'<p>Tu último inicio de sesión en el aula virtual fue el
{last_login_day}, '
    html += f'y ese día marca tu inicio de sesión número {n_logins} desde
que '
    html += f'ingresaste en la UPCT. 😇</p>'
    html += f'<p>¡Sigue así, vas muy bien! 💪</p>'

    return html
```

create_student_html, función que devuelve la respuesta HTML de los alumnos

Hemos personalizado las respuestas de manera que parezcan más personales con emojis y demás. Cuando veamos las respuestas que devuelven los endpoints, veremos cómo se vería la información una vez mostrada en Moodle.

3.4.3. Endpoint para los profesores

Este endpoint tiene como objetivo pasar información sobre los inicios de sesión de los alumnos a los profesores, y como se consultan dentro de los cursos, le llamaremos '*courselogins*'. Este endpoint solicitará un parámetro, el ID del curso.

```
import azure.functions as func

app = func.FunctionApp(http_auth_level=func.AuthLevel.FUNCTION)

@app.route(route="courselogins")
def courselogins(req: func.HttpRequest) -> func.HttpResponse:
```

Declaración de la función correspondiente al endpoint para los profesores

Primero creamos una instancia de la clase '*FunctionApp*', que usaremos para proporcionar métodos de administración a nuestra aplicación. A este le especificamos con '*http_auth_level=func.AuthLevel.FUNCTION*' que se requerirá una clave de autorización para acceder a las funciones.

Los niveles de autorización son:

- **FUNCTION:** Requerirá una clave de autorización para poder acceder a la función.

- **ANONYMOUS:** Este nivel no requiere de ninguna clave. Cualquiera podría crear una solicitud HTTP a este endpoint y recibir la respuesta deseada.
- **ADMIN:** Este nivel requiere una clave admin. Es el nivel de autorización más restrictivo y seguro de los tres.

Para crear el endpoint, a la función le colocamos el decorador '*@app.route()*'. En Python, los decoradores (Python Basics, s.f.) son funciones que toman otra función como argumento y devuelven una nueva función que puede modificar el comportamiento de la función original sin cambiar su código fuente.

El decorador '*@app.route*' se usa para asociar una URL a la función. Cuando esta función reciba una solicitud HTTP a la URL especificada en el parámetro '*route*', que en nuestro caso será '*courselogins*', se devuelve la respuesta HTTP.

Para proporcionar información sobre la función, especificamos que la función toma como argumento una solicitud HTTP, '*func.HttpRequest*', y devolverá una respuesta HTTP, '*func.HttpResponse*', aunque no es necesario.

```
# Parametro que requiere el endpoint. ID del curso
id = req.params.get('id')

if not id:
    try:
        req_body = req.get_json()
    except ValueError:
        pass
    else:
        id = req_body.get('id')
```

Obtención del parámetro 'id' que se debe pasar al endpoint

Lo primero que hará la función es intentar obtener el valor del parámetro 'id' de la URL de la solicitud HTTP. Si este valor no está presente, se intentará obtener el valor del cuerpo de la solicitud HTTP. Si ahí tampoco existe el valor, entonces procederemos al siguiente bloque de código:

```
if id:
    try:
        alberto_storage_connection_str =
os.environ["AzureWebJobsStorage"]
        blob_service_client =
BlobServiceClient.from_connection_string(alberto_storage_connection_str)
        container_client =
blob_service_client.get_container_client("aggregate")

        archivo =
container_client.get_blob_client(f"courses/{id}/aggregate.avro").download
_blob().readall()
        reader = fastavro.reader(io.BytesIO(archivo))
        record_list = []
        for record in reader:
```

```
            record_list.append(record)

        html_respuesta = create_course_html(record_list)

        return func.HttpResponse(
            html_respuesta,
            status_code=200
        )
    except Exception as e:
        return func.HttpResponse(
            f"Ocurrió un error\n: {str(e)}",
            status_code=500)
else:
    return func.HttpResponse(
            f"Es necesario introducir el campo 'id' del curso",
            status_code=400
    )
```

Código que devuelve la respuesta HTTP dependiendo del parámetro 'id'

En caso de haber pasado el valor, nos conectaremos como antes a Azure y descargaremos la información del archivo avro del curso específico con '*download_blob().readall()*'.

Si hemos conseguido conectarnos y descargar el archivo avro, lo primero que hacemos es usar la función '*io.BytesIO()*' para crear un flujo de bytes a partir del archivo avro. Este flujo de bytes lo leemos con la función '*reader*' de '*fastavro*'.

Creamos una lista vacia '*record_list*' donde almacenaremos los registros leídos del archivo avro. Justo a continuación, agregamos a la lista todos los registros contenidos en el archivo avro en la lista con un bucle.

Pasamos la lista a la función '*create_course_html*' antes vista y guardamos los resultados en la variable '*html_respuesta*'. Si hasta este punto no ha habido ningún problema, enviaremos dicha respuesta con código 200 (indica que la solicitud se ha hecho con éxito). Si no hemos conseguido conectarnos a Azure, o ha habido algún otro error imprevisto, mandaremos una respuesta HTTP con código 500, indicando que ocurrió un error.

Si no conseguimos extraer el ID ni de la URL ni del cuerpo de la solicitud HTTP, la respuesta que nos devolverá el endpoint nos indicará que introduzcamos el campo 'id' y vendrá con código 400, que indica una mala sintaxis en la solicitud.

Con el endpoint creado, implementamos la función en Azure desde VSCode (Microsoft Azure, 2019) con la instrucción 'Deploy to Function App'. Si no lo hemos hecho anteriormente, debemos crear la 'Function App', lo que nos permite ejecutar la función Azure. Su nombre debe ser único a nivel global, y la llamaremos '*tfgmoodleazureapp*'. La asignamos a la cuenta de almacenamiento '*tfgmoodle*', y si no hay fallos, comprobamos que el endpoint responde a las solicitudes. Para ello usamos Postman, un cliente que nos da la posibilidad de realizar solicitudes HTTP.

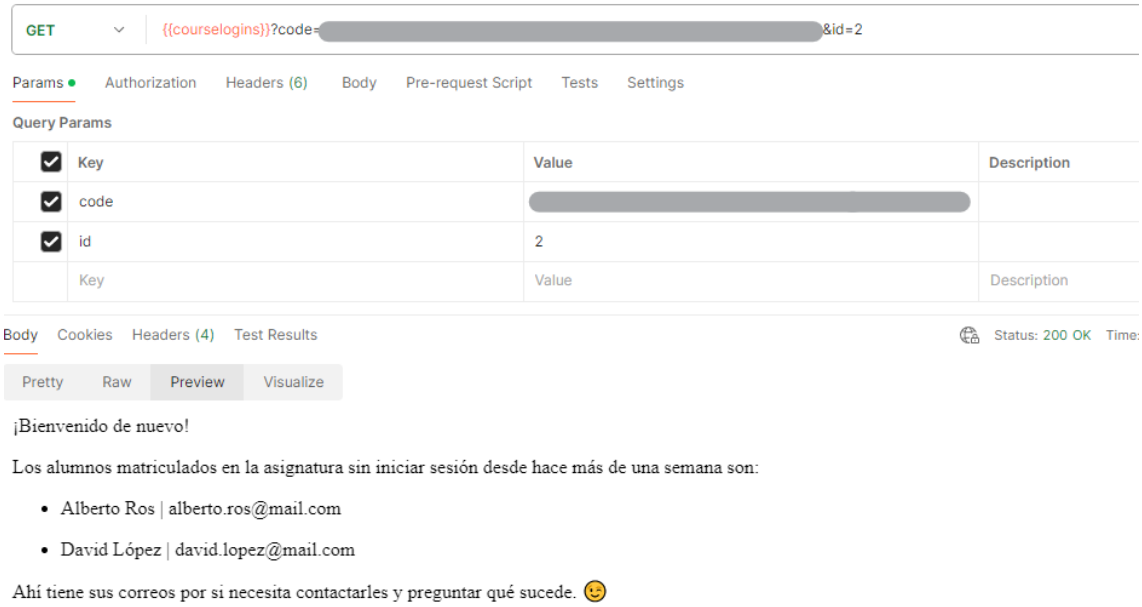

Figura 21: *Respuesta del endpoint para profesores*

Pasándole el ID de curso 2 en el cuerpo de la solicitud HTTP nos devuelve la misma respuesta.

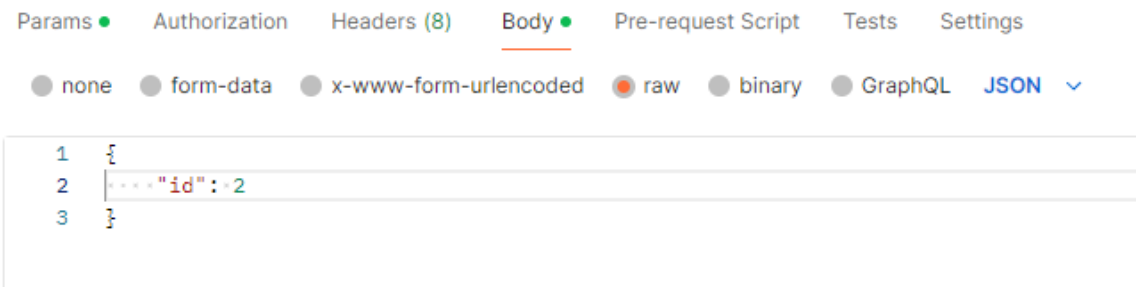

Figura 22: *Envío del 'id' en el cuerpo de la solicitud HTTP en vez de en la URL*

Si no ponemos el parámetro '*id*' en la URL, pero sí en el cuerpo de la solicitud, el resultado es exactamente el mismo que el anterior.

Si hacemos una petición sin el campo 'id', la respuesta será la que hemos visto en el código:

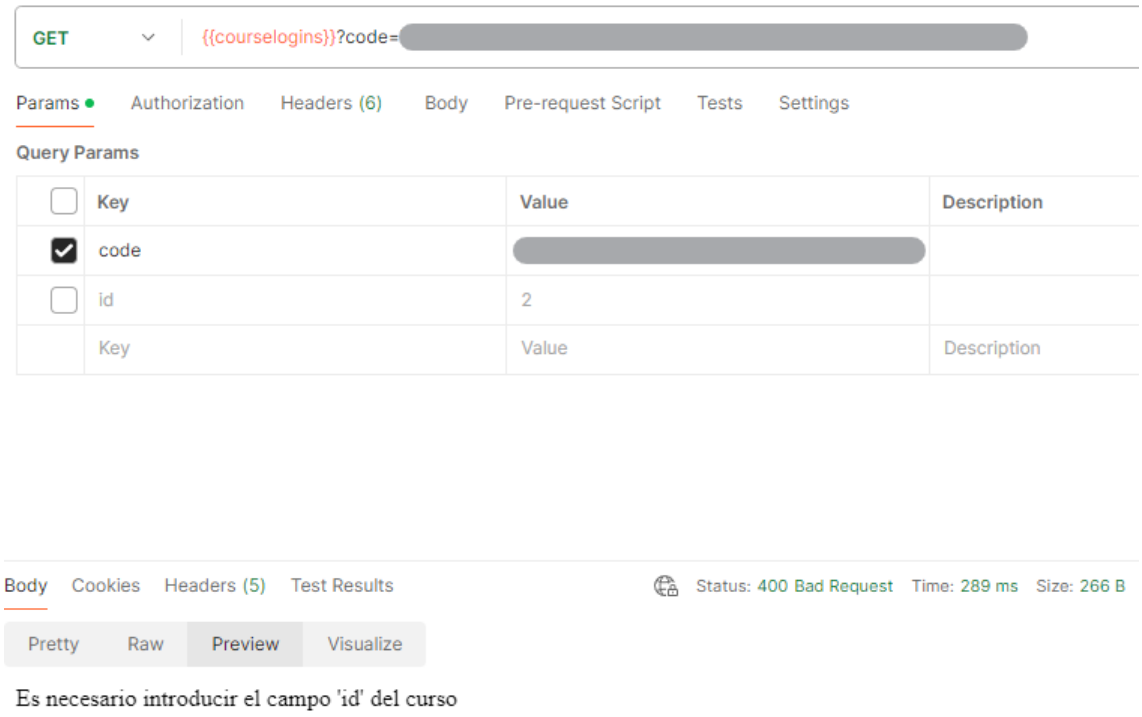

Figura 23: *Respuesta del endpoint para profesores si no pasamos el campo 'id'*

En los raros casos de pasar una clave errónea nos devolverá un código 401 (no autorizado), o pasamos un ID de un curso que no existe nos devolverá un código 500 (error interno del servidor, más específicamente, que el blob no existe).

3.4.4. Endpoint para los alumnos

De manera muy similar al endpoint creado para enviar la información a los profesores, creamos un endpoint al que le vamos a pasar el ID del alumno que tenga iniciada la sesión y este nos va a devolver una respuesta HTML personalizada con los datos de los archivos avro.

No voy a poner la función entera del endpoint porque es muy similar a la de los profesores, pero si los cambios más significativos:

```
@app.route(route="userlogins")
def userlogins(req: func.HttpRequest) -> func.HttpResponse:
...
archivo = container_client.get_blob_client(
      f"users/{id}/aggregate.avro").download_blob().readall()
...
html_respuesta = create_student_html(record_list)
...
else:
     return func.HttpResponse(
          f"Es necesario introducir el campo 'id' del usuario",
          status_code=400
```

Cambios más significativos del endpoint de los alumnos respecto al de los profesores

Como antes, comprobamos que funciona correctamente haciendo una petición HTTP desde Postman:

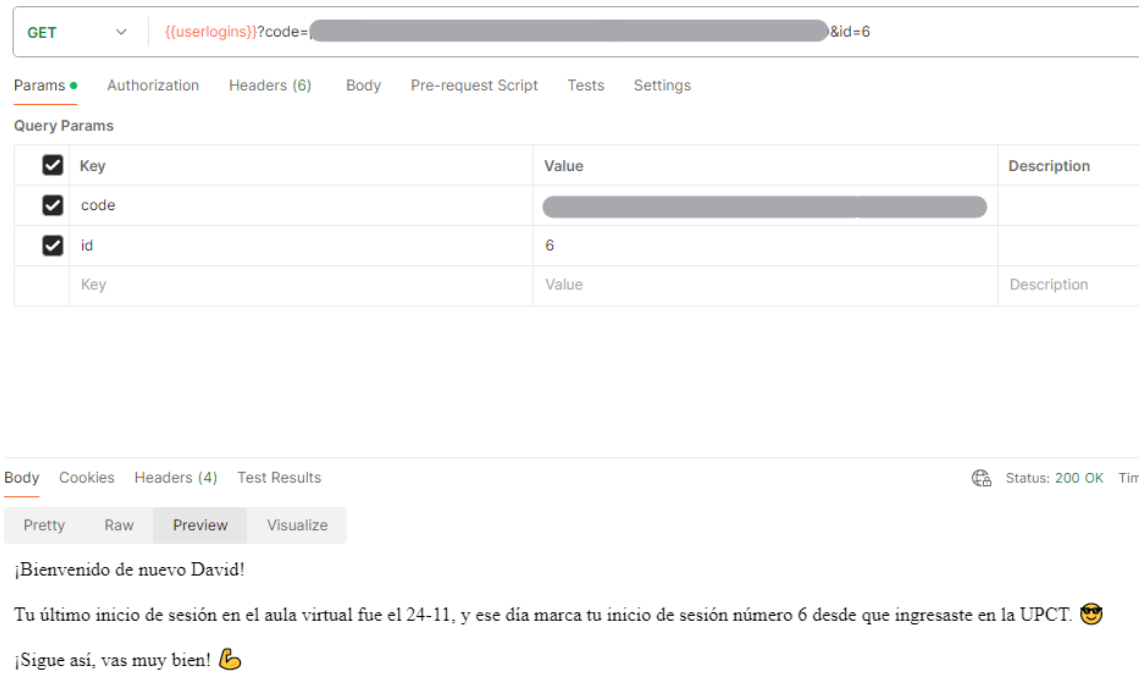

Figura 24: *Respuesta del endpoint para alumnos*

Pasándole un ID de un usuario que existe, vemos que devuelve el HTML con el formato que vimos antes. Y de la misma manera que hemos visto en el endpoint para profesores, si no pasamos el campo 'id', pasamos una clave errónea o pasamos un ID de un alumno que no existe, nos devolverá los códigos que mencionamos en su caso.

3.5. Desarrollo de los plugins de tipo bloque y visualización de los datos en Moodle

El siguiente paso es desarrollar dos plugins de tipo bloque, uno que se mostrará a los profesores, que contendrá la respuesta HTTP que genera la función '*create_course_html*', y el otro irá enfocado a los alumnos, con la respuesta HTTP que genera la función '*create_student_html*'.

Para que Moodle reconozca ambos plugins como plugins de tipo bloque debemos colocarlos en el siguiente directorio: '*\alber_moodle_data_data\blocks*'. Al plugin de los profesores lo llamaremos '*block_teachers_info*' y al de los alumnos '*block_students_info*'.

3.5.1. Bloque de los profesores

A este plugin lo llamaremos '*block_teachers_info*' y su esquema de archivos será:

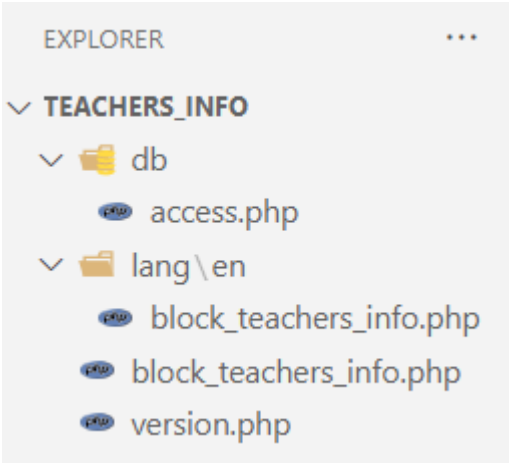

Figura 25: *Estructura de archivos del plugin 'block_teachers_info'*

El archivo '*version.php*' ya lo vimos cuando creamos el plugin local, por lo que no es necesario explicar su funcionamiento.

- **access.php:** Archivo necesario para definir los permisos de acceso necesarios para crear, editar y ver el bloque.

```
defined('MOODLE_INTERNAL') || die();
$capabilities = [
    'block/teachers_info:view' => [
        'captype' => 'read',
        'contextlevel' => CONTEXT_COURSE,
        'archetypes' => [
            'teacher' => CAP_ALLOW,
            'editingteacher' => CAP_ALLOW,
        ],
    ],
];
```

Código de access.php

En la variable '*$capabilities*' debe almacenar los permisos. En nuestro caso queremos que solo los profesores sean capaces de ver el contenido de estos bloques, por lo que le ponemos un permiso que sirva para que solo aquellos con dicho permiso puedan ver el bloque. Este permiso se llama '*block/teachers_info:view*'. Podríamos haberle puesto otro tipo de permiso como '*block/teachers_info:addinstance*', que sirve para aquellos que tengan el permiso para crear el bloque. Ambos casos nos sirven para nuestro propósito.

El parámetro '*captype*' sirve para indicar la capacidad del permiso, y en este caso, los que lo tengan podrán leer el contenido del bloque, nada más.

El parámetro '*contextlevel*' sirve para indicar el nivel de contexto en el que se aplica dicho permiso. Al ponerle como valor '*CONTEXT_COURSE*', le estamos diciendo que este permiso se aplica a nivel de curso, porque queremos mostrarlo dentro de los cursos.

Por último, el parámetro '*archetypes*' contendrá los roles de usuario que tengan dicho permiso. Los vimos anteriormente en la tabla '*mdl_role*'. Queremos que solo sean capaces de ver dicho bloque los profesores del curso especifico, por lo que le indicamos los roles que existen para los profesores, '*teacher*', y '*editingteacher*'. Este sirve para los

profesores que puedan editar los contenidos del curso. A estos roles les asignamos '*CAP_ALLOW*', que es una constante que sirve para permitir que los roles puedan ver el bloque.

- **(lang/en/) block_teachers_info.php:** No confundir con el archivo del mismo nombre que está en la raíz del plugin. Sirve para guardar cadenas de texto predeterminadas del bloque, tales como etiquetas, mensajes de error u otros elementos.

```
$string['teachers_info'] = '(new teachers_info block)';
$string['pluginname'] = 'Alumnos sin iniciar sesión en una semana';
```

Código de (lang/en/) 'block_teachers_info.php'

Accederemos a estas cadenas de texto para poder mostrarlas en Moodle. Este archivo también es útil en páginas Moodle donde haya diferentes idiomas, así, Moodle intercambiará las cadenas de texto correspondientes al idioma que se desee dependiendo de la carpeta donde estén situadas. Este está situado en el directorio '*lang/en/*', por lo que lo adecuado sería usar estas cadenas en un Moodle con el idioma configurado en inglés.

- **block_teachers_info.php:** No confundir con el que está dentro de '*lang/en/*'. Este archivo sirve para definir el comportamiento del bloque y lo que vamos a enseñar dentro del mismo.

```
defined('MOODLE_INTERNAL') || die();

class block_teachers_info extends block_base {
    function init() {
        $this->title = get_string('pluginname', 'block_teachers_info');
    }

    function get_content() {

        if ($this->content !== NULL) {
            return $this->content;
        }

        $course = $this->page->course;
        $context = get_context_instance(CONTEXT_COURSE, $course->id);
        if (!has_capability('block/teachers_info:view', $context)) {
            return;
        }

        $url =
'https://tfgmoodleazureapp.azurewebsites.net/api/courselogins?code={codig
o} &id=' . $course->id;
        $html = file_get_contents($url);

        $this->content = new stdClass;
        $this->content->text = $html;
```

```
        return $this->content;
    }
}
```

Código de block_teachers_info.php

En la definición de la clase, extendemos de la clase '*block_base*', lo que hace que '*block_teachers_info*' sea una subclase de dicha clase y esto hace que tenga las funcionalidades de una clase de tipo bloque.

Lo primero que hacemos al iniciar la clase es declarar la función responsable de inicializar el bloque, '*init()*'. Lo único que hace es asignar el título del bloque de acuerdo con la cadena de texto almacenada en '*pluginname*', que como hemos visto antes, es '*Teacher info (TFG Block)*'. El segundo parámetro de '*init()*' sirve para indicar el archivo donde están almacenadas las cadenas de texto.

A continuación seguimos con la función que se utiliza para obtener el contenido del bloque, '*get_content()*'. Lo primero que hace es comprobar si el contenido que se debe mostrar en el bloque ya ha sido obtenido. Si es así devuelve dicho contenido, si no, sigue con la ejecución del código para obtenerlo.

Lo siguiente es obtener el contexto del curso en el que estamos. En '$course' guardamos información sobre el curso actual y usamos la función 'get_context_instance' para obtener el contexto del curso. El primer argumento es el nivel de contexto que se desea obtener, y el segundo ID del objeto para el que queremos obtener el contexto.

Con '*has_capability()*' podemos comprobar si, dado el permiso y contexto que le pasemos, el usuario actual tiene la capacidad necesaria para ver el bloque. En este caso lo negamos con una exclamación para indicar que, si no se tiene la capacidad, el bloque no muestre nada.

Si el usuario tiene la capacidad para ver los contenidos del bloque, hacemos la solicitud HTTP y guardaremos la respuesta del endpoint '*courselogins*' en una variable '*$html*' con la función '*file_get_contents()*'.

Creamos un nuevo objeto vacío con '*stdClass*' y lo almacenamos en el contenido del bloque para justo a continuación establecer el contenido que va a tener dicho objeto en su propiedad '*text*'.

Devolvemos el contenido del bloque y obtenemos el siguiente resultado:

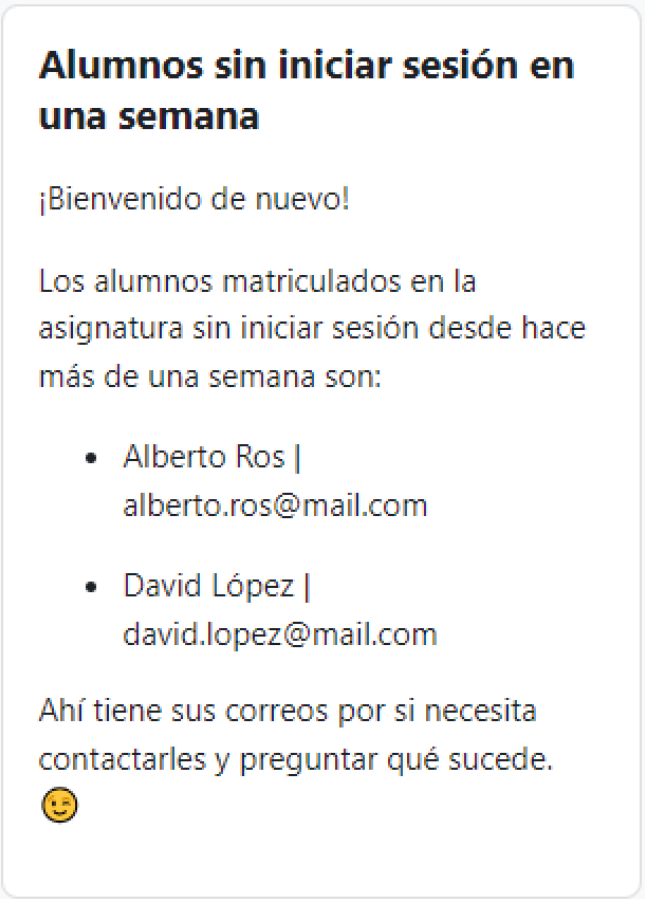

Figura 26: *Bloque que muestra los usuarios matriculados sin iniciar sesión en una semana*

Como vemos, es el HTML que esperábamos recibir del endpoint '*courselogins*'.

En el caso de que el alumno pueda llegar a crear en su página este bloque, no verá su contenido gracias a la comprobación que hicimos dentro de '*get_content()*' que se aseguraba que los usuarios sin la capacidad necesaria pudiera ver dicho contenido.

3.5.2. Bloque de los alumnos

A este bloque lo llamaremos '*block_students_info*' y su esquema de archivos será:

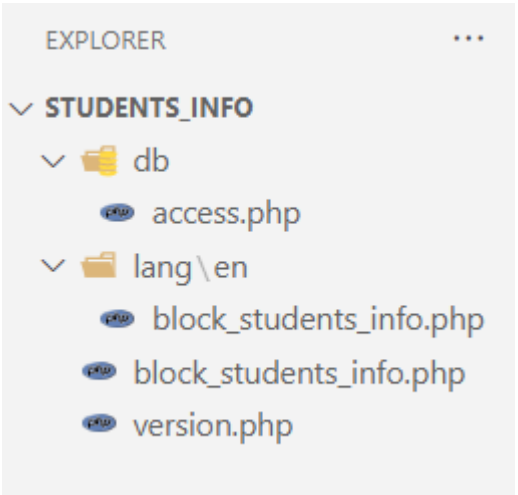

Figura 27: *Estructura de archivos del plugin 'block_students_info'*

Como vemos, comparte la misma estructura de archivos que el bloque de los profesores, por lo que no es necesario explicar todo lo visto antes. Lo único que vamos a ver van a ser los cambios que hace que este bloque se distinga del otro:

```
$url =
'https://tfgmoodleazureapp.azurewebsites.net/api/userlogins?code={código}
&id=' . $USER->id;
$html = file_get_contents($url);
```

Parte del código de 'block_students_php'

Lo único distinto del otro código es el endpoint al que vamos a realizar la solicitud HTTP cambia al de '*userlogins*'.

Es importante mencionar que en este caso no vamos a usar el archivo access.php para definir capacidades, ya que, aunque podríamos limitar la visibilidad de este bloque a solo los alumnos, la información no tiene por qué ir enfocada únicamente a alumnos, al contrario que en el caso de los profesores, cuya información no debería estar al alcance de dichos alumnos.

Con esto, el contenido del bloque de los alumnos queda así:

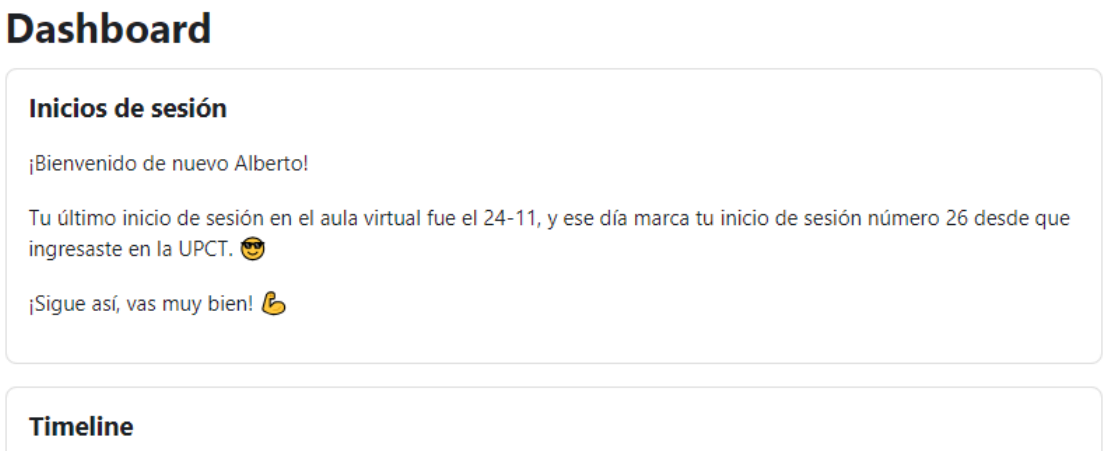

Figura 28: *Bloque que muestra la información sobre los inicios de sesión de los alumnos*

4. Conclusiones

En este trabajo se ha explorado la explotación de eventos generados por Moodle para la obtención de datos que puedan servir de ayuda tanto para profesores como para alumnos. Se ha visto el funcionamiento de Docker, la creación de dos tipos de plugin en PHP, la carga y descarga de los eventos almacenados en Azure, su análisis mediante Python, y finalmente, la visualización de los datos analizados en Moodle.

Como ya hemos mencionado más de una vez durante el desarrollo, este trabajo no es más que una prueba de concepto, y como hemos visto, bastante completa. Lo importante de esta prueba se ha conseguido, que es la demostración de que podemos llegar a conseguir datos que puedan servir para la monitorización de los profesores a sus alumnos con el objetivo de mejorar la motivación, prevenir el abandono temprano, ajustar el trabajo de una manera más eficiente y mejorar el entendimiento global, y todo ello haciendo llamadas a unas pocas copias de tablas de datos de Moodle, las cuales apenas cambian a lo largo del curso, consiguiendo así otro de nuestros objetivos, no afectar de ninguna manera el rendimiento de Moodle.

4.1 Líneas futuras

De cara al futuro, cuando se quieran implementar las ideas de esta prueba de concepto a un entorno de producción en el aula virtual de la UPCT, una de las tareas que se deberá realizar será:

- Automatización del análisis de los datos y su subida al contenedor.

En esta prueba de concepto todo esto se hace de manera manual, y en un entorno real sería ideal que esta tarea estuviese programada para ejecutarse cada cierto tiempo.

Además, para que todo este esfuerzo pueda dar como resultado en algo que se pueda implementar en el futuro y obtener unos análisis interesantes, es importante tener en cuenta que Moodle deberá implementar eventos que recojan más tipos de acciones que las de ahora, como por ejemplo, eventos que indiquen la creación de una tarea por parte del profesor, eventos que indiquen la eliminación de una entrega por parte del alumno, etc., para así poder obtener un posible espectro de datos muchísimo más amplio que los que se pueden conseguir en la actualidad únicamente a través de los eventos.

4.2. Conocimientos adquiridos

En el proceso de desarrollo de este proyecto, algunos de los conocimientos adquiridos han sido los siguientes:

- Se han reforzado conocimientos sobre la aplicación Docker y el lenguaje de programación Python, herramientas muy populares en la actualidad en entornos profesionales.
- Se ha aprendido a trabajar con el entorno Azure de Microsoft y las herramientas que este ofrece, reforzando a su vez los conocimientos sobre endpoints.
- Como funciona por dentro la plataforma de aprendizaje Moodle, el funcionamiento de sus eventos y plugins.

- Se han reforzado los conocimientos sobre Data Science aprendidos en la asignatura de cuarto de carrera '*Introducción a Data Science*'.

5. Referencias

INTEF. (2017). Analíticas de Aprendizaje: evidencias e investigación sobre su uso. Recuperado el 26 de enero de 2024, de https://intef.es/wp-content/uploads/2017/05/Learning-Analytics_JRC_INTEF_Abri2017.pdf

UNESCO. (s. f.). Aprendizaje Personalizado. Recuperado el 26 de enero de 2024, de https://unesdoc.unesco.org/ark:/48223/pf0000250057_spa

IDdocente. (s. f.). EdPuzzle: La herramienta para crear y editar tus videos y darle la vuelta a tu clase. Recuperado el 26 de enero de 2024, de https://iddocente.com/edpuzzle-herramienta-crear-editar-videos/

INTEF. (s. f.). Classcraft. Convierte la clase en una aventura épica. Recuperado el 26 de enero de 2024, de https://intef.es/observatorio_tecno/classcraft-convierte-la-clase-en-una-aventura-epica/

IdeasPropiasEditorial (s. f.) Plataforma LMS: Qué es el Learning Management System. Recuperado el 6 de febrero de 2024, de https://www.ideaspropiaseditorial.com/blog/plataforma-lms/

Asana. (s. f.). Prueba de concepto (POC): qué es y cómo implementarla. Recuperado el 5 de febrero de 2024, de https://asana.com/es/resources/proof-of-concept

Bitnami. (2023). Bitnami LMS powered by Moodle™ LMS. Recuperado el 13 de septiembre de 2023, de https://github.com/bitnami/containers/tree/main/bitnami/moodle

Moodle. (2023). Plugins FAQ. Recuperado el 29 de septiembre de 2023, de https://docs.moodle.org/403/en/Plugins_FAQ

MoodleDev. (s. f.). Plugin types. Recuperado el 29 de septiembre de 2023, de https://moodledev.io/docs/apis/plugintypes

Moodle. (2023). Events API. Recuperado el 29 de septiembre de 2023, de https://docs.moodle.org/dev/Events_API

LDTalent. (2021). How to create a Moodle event listener as a local plug-in. Recuperado el 2 de octubre de 2023, de https://blog.ldtalentwork.com/2021/11/08/how-to-create-a-moodle-event-listener-as-a-local-plug-in/

Microsoft. (2023). Storage account overview. Recuperado el 16 de enero de 2024, de https://learn.microsoft.com/en-us/azure/storage/common/storage-account-overview

Examulator. (2022). Mdl40_erd Database. Recuperado el 16 de octubre de 2023, de https://www.examulator.com/er/4.0/

Python. (s. f.). Data Structures. Recuperado el 9 de enero de 2024, de https://docs.python.org/3/tutorial/datastructures.html#dictionaries

Microsoft. (2023). Lectura y escritura de datos de Avro en streaming. Recuperado el 17 de enero de 2024, de https://learn.microsoft.com/es-es/azure/databricks/structured-streaming/avro-dataframe

Microsoft. (2023). Introducción a Azure Blob Storage. Recuperado el 10 de enero de 2024, de https://learn.microsoft.com/es-es/azure/storage/blobs/storage-blobs-introduction

Python Basics. (s. f.). Flask Tutorial: Routes. Recuperado el 20 de noviembre de 2023, de https://pythonbasics.org/flask-tutorial-routes/

Microsoft Azure. (2019). How to deploy Azure Functions with Visual Studio Code | Azure Tips and Tricks. [Vídeo]. Youtube. Recuperado el 26 de enero de 2024, de https://www.youtube.com/watch?v=RD3vUCdRf8o

6. Anexo

Los archivos del proyecto se han subido a un repositorio de GitHub llamado ‘TFG-Moodle’:

https://github.com/albegalia/TFG-Moodle

El proyecto está dividido en multiples 'workspaces', por tanto, habrá una carpeta por workspace, y dentro, todos sus archivos correspondientes.

Explicación de lo más relevante a comentar de cada archivo/carpeta:

Archivo "docker-compose.yml": Su contenido nos permite tanto instalar e iniciar Moodle y MariaDB. Contiene además las variables de entorno que se usaron para este proyecto.

Carpeta "Plugin local": Contiene los archivos del plugin local 'send_events'. En 'observer.php' se ha omitido tanto la URL del microservicio como su correspondiente token de autorización.

Carpeta "Analisis eventos": Contiene los archivos que se usaron para los análisis de los eventos. Prácticamente todo el trabajo se realizó dentro del archivo de Jupyter Notebook llamado 'tests.ipynb'. Dentro está todo el proceso que se realizó en el análisis de los eventos, además de contener trabajo que no ha terminado en el proyecto final. El archivo 'tfg-functions.py' contiene funciones que fui usando para realizar las pruebas, pero al final todo lo he ido realizando dentro de 'tests.ipynb'. La carpeta 'square_analytics' fue proporcionada por los directores del proyecto. Se ha omitido el archivo '.env', que es el que contenia la cadena de conexión con Azure.

Carpeta "Endpoints Azure": Contiene los archivos que se usaron para desplegar los endpoints en Azure. El archivo que se ocupa de desplegar dichos endpoints es 'function_app.py', que contiene los dos endpoints del proyecto mas uno de prueba que venía por defecto cuando se creó la función Azure en VSCode. El archivo 'tfg_functions.py' contiene las funciones que crean la respuesta HTML. Se ha omitido el archivo 'local.settings.json', que es el que contenia la cadena de conexión con Azure.

Carpeta "Plugins bloque": Contiene los archivos de los plugins de tipo bloque, 'students_info' y 'teachers_info'. Se ha omitido el código en la URL que requieren los endpoints para solicitar información tanto en 'block_teachers_info.php' como en 'block_students_info.php'.