Computación Ubicua

Smart Uni

Integración de sistemas IoT en la Universidad de Alcalá

**Javier Lombardía Ocaña, César Martín Guijarro,
Lucía Picado Joglar y Valeria Fernanda Villamares Félix**

**Universidad de Alcalá
21 de mayo de 2023**

Índice general

1. Introducción **2**
1.1. Objetivos . . . 2
1.2. Justificación . . . 3
1.3. Estructura de la memoria . . . 4

2. Inconvenientes en el primer proyecto y propuestas de solución **5**
2.1. Limitaciones en la comunicación . . . 5
2.2. Alojamiento del servidor . . . 5
2.3. Alojamiento de la BBDD . . . 6
2.4. Eliminación de la máquina virtual . . . 6
2.5. Inconvenientes adicionales . . . 6
2.6. Testing lento . . . 7

3. Implementación y soluciones a los problemas **8**
3.1. Implementación con *FastApi* . . . 8
3.2. Servidores locales . . . 8
3.3. Entornos virtuales . . . 9
3.4. Base de datos online . . . 9
3.5. Testing con Postman . . . 9
3.6. Mejoras adicinoales . . . 9

4. Desarrollo del Backend **10**
4.1. Estructura clara y organizada . . . 10
4.2. Manejo de errores . . . 11
4.3. Impresión interrumpida . . . 11
4.4. Utilidades para la base de datos . . . 11
4.5. Endpoints . . . 13

Capítulo 1

Introducción

En este capítulo se va a realizar una introducción detallada sobre todos los temas

1.1. Objetivos

1.2. Justificación

1.3. Estructura de la memoria

Capítulo 2

Inconvenientes en el primer proyecto y propuestas de solución

En este capítulo trataremos sobre todos los inconvenientes con los que nos topamos durante el desarrollo de la primera práctica y que intentaremos mejorar para el óptimo desarrollo de la práctica actual.

2.1. Limitaciones en la comunicación

Para la comunicación efectiva de los componentes *Back-End*, *Front-End*, *MQTT*, y *Arduino* en el proyecto anterior, se requería la presencia física de todos los miembros. La configuración pasada basada en una máquina virtual contenedora del servidor *Tomcat* y de la base de datos dificultó la compartición de recursos, resultando en tener que limitar la funcionalidad del *Back-End* a un solo ordenador.
Para abordar esta problemática, César desarrolló una solución mediante la creación de una *red privada virtual* (*VPN*) a través de *Hamachi*, que permitió la comunicación remota a su máquina, dando acceso a la edición de la base de datos y la implementación de cambios en el *Back-End* por parte de cualquier miembro. Sin embargo, esta solución fue limitada debido a que César tenía que mantener su máquina encendida en todo momento para hospedar los servicios. Por otro lado la configuración de la *VPN* en el dispositivo *ESP-32* resultó imposible; implicando un lento desarrollo de sus funciones.

Como solución definitiva a este problema, César ha propuesto dos alternativas: la primera de estas era alquilar algún tipo de servicio online para poder llevar a cabo un hosting compartido sin necesidad de configurar redes privadas virtuales. En segundo lugar, investigar en otro tipo de tecnologías que permitan replicar los servidores de manera local, permitiendo así que todos los miembros del equipo puedan trabajar en el *Back-End* y *Front-End* eliminando así cualquier tipo de necesidad para realizar conexiones externas.

2.2. Alojamiento del servidor

Pese a que se ha mencionado parcialmente en el apartado anterior, alojar el servidor fue algo bastante complejo. Debido a que las herramientas que se nos proporcionaron no eran fáciles de compartir, lo más normal es que en todos los equipos sólamente una persona pudiese acceder al Back-End y a la base de datos. Con la VPN, se pudo mejorar mucho este aspecto. Además,

César configuro unos servicios SSH que permitían a cualquier compañero obtener acceso total a una shell del servidor. Pese a ello, esta situación no es ideal en absoluto. Si en un momento determinado dos personas desean realizar un testing de sus modificaciones, una persona tendrá que esperar a la otra.

2.3. Alojamiento de la BBDD

En el proyecto previo, se detectaron limitaciones en la configuración de la base de datos alojada en la máquina virtual proporcionada por los docentes. La complejidad en el acceso a la configuración de la base de datos limitaba su uso y debido a la manera en la que estaba estructurada, solo permitía la conexión de César, la persona que estaba hosteando los servicios. Para solucionar esta problemática, César implementó una red privada virtual (*VPN*), mediante la cual se permitió la conexión remota a la base de datos desde cualquier miembro del equipo. Además, César investigó sobre cómo poder configurar conexiones remotas en este sistema. Sin embargo, esta solución dependía de que César mantuviera su máquina encendida las 24 horas del día para hospedar los servicios.

Como solución definitiva a esta limitación, César ha propuesto la externalización de la base de datos a un servicio que esté disponible en todo momento. De esta forma, se eliminaría la dependencia de la máquina virtual y la VPN, y se evitaría la necesidad de mantener una máquina encendida constantemente para ejecutar el servicio. Esta solución no solo mejoraría la accesibilidad a la base de datos, sino que también eliminaría la necesidad de realizar configuraciones complejas para el acceso a la misma.

2.4. Eliminación de la máquina virtual

En el proyecto anterior, la configuración basada en una máquina virtual contenedora del servidor Tomcat y de la base de datos resultó ineficiente en términos de compartición de recursos. Además, la máquina virtual no estaba correctamente configurada y no estaba documentada, lo que dificultaba su uso y modificación para adaptarse a las necesidades del proyecto. Debido a su tamaño, también era intransferible a otras máquinas y tenía un rendimiento pobre.
Como alternativa, es necesario buscar un sistema que sea simple de compartir y transferir, y que pueda documentarse y configurarse rápidamente. Una propuesta realizada por César es la creación de un entorno virtual donde se pueden compartir los sistemas en un repositorio y transferir rápidamente mediante un archivo de requisitos. Esta solución permitiría una mayor eficiencia en la compartición de recursos, ya que todos los miembros del equipo tendrían acceso al mismo entorno virtual, lo que simplificaría el proceso de desarrollo y eliminaria la necesidad de alojar todos los servicios de backend y base de datos en una sola máquina encendida constantemente.
Esto supuso que César empezase a plantear la posibilidad de realizar el desarrollo del back-end en Python debido a la facilidad que supone la creación de entornos que cumplan estos requisitos.

2.5. Inconvenientes adicionales

Durante el desarrollo del primer proyecto se pudieron notar inconvenientes adicionales que afectaron al desarrollo del mismo. Principalmente la mayoría de problemas surgieron por haber

desarrollado el Back-End utilizadno Tomcat. Este sistema supuso muchos problemas, entre los que destacan conflictos dados por la colisión de versiones de Java con los diferentes equipos de los miembros del equipo. Otro inconveniente dado por el desarrollo en Tomcat era la falta de documentación en línea. Por otro lado, un gran problema que se presentó al utilizar Tomcat el proceso extremadamente lento que suponía realizar cualquier mínimo cambio en el backend: tras haber realizado un cambio, era necesario realizar una compilación completa del proyecto, conectarse remótamente a la máquina virtual, enviar el fichero compilado de la máquina donde se haya realizado el desarrollo a la máquina virtual, acceder a los servicios de TomCat, eliminar el proyecto del servicio y desplegar esta nueva versión. Este proceso es totalmente ridículo, sobre todo si tenemos en cuenta el hecho de que, obviamente, lo más probable que ocurra tras realizar un cambio, es que haya algún fallo menor y que para poder corregirlo, debe repetirse este proceso al completo.
Otro gran problema dado por TomCat era la extremadamente alta dificultad para configurar los logs del sistema. Cosa que jamás pudimos realizar correctamente ni con ayuda de nuestros docentes.

Un cambio de librería no sólo supondría una mejora muy positiva, si no que es totalmente necesario.

2.6. Testing lento

El proceso de testing fue también una labor compleja durante la pasada práctica. Debido a que no se nos proporcionó ni explicó a fondo ninguna herramienta de testing de todos los endpoints de la aplicación de Back-End; nuestra manera de hacer testing se basaba en probar a escribir en el navegador las direcciones completas de cada endpoint. Además, nos veíamos forzados a realizar todas las llamadas con request de HTTP GET para poder enviar parámetros en la query (ya que no podíamos enviar parámetros de ninguna otra manera). Esto era un proceso tedioso y rudimentario que también afectaba de manera negativa al desarrollo de la práctica. Como solución, César ha propuesto implementar una metodología de testing basada en el software Postman. Dicha metodología debería organizar de la mejor manera posible todos los endpoints y aprovechar al máximo las funciones de las variables de dicho programa para recortar al máximo el tiempo implementado en el testing.

Capítulo 3

Implementación y soluciones a los problemas

En este capítulo se va a detallar de manera más detallada cómo se ha implementado nuestro proyecto. También se podrá ver de qué maneras esta implementación soluciona los problemas expuestos en el capitulo anterior.

3.1. Implementación con *FastApi*

El desarrolo del Back-End se ha llevado a cabo utilizando el *framework* de *FastApi* para Python. Se ha elegido este lenguaje ya que es un lenguaje interpretado. Esto eliminará la necesidad de compilar el proyecto y realizar un despliegue por cada cambio que se realice. Se ha elegido *FastApi* debido a su destacada rapidez, su extensa documentación, su extremadamente simple sintaxis, la posibilidad de usar tipado estático y las facilidades que brinda para la documentación del proyecto.

3.2. Servidores locales

Se ha decidido utilizar la librería *uvicorn* para poder levantar con un simple comando una réplica local del servidor. Esta librería es totalmente compatible con *FastApi*. Además, *uvicorn* podrá refrescar el servidor cada vez que detecte un cambio en el código. De esta manera, para poder introducir modificaciones y testearlas, será tan simple como pulsar *Ctrl+S* con el servidor lanzado. Ya no será necesario hacer todo el proceso extremadamente rudimentario y absurdo que se tenía que llevar a cabo para introducir cualquier modificación utilizando *Tomcat*.

3.3. Entornos virtuales

3.4. Base de datos online

3.5. Testing con Postman

3.6. Mejoras adicinoales

Capítulo 4

Desarrollo del Backend

Como se ha detallado anteriormente, este apartado ha sido desarrollado por completo utilizando el *framework* de *FastApi* para el lenguaje *Python*. Un desarrollo de un *Backend* complejo y sotisficado siempre es una tarea de alta dificultad, sin embargo; gracias a utilizar este *framework*, nuestro trabajo se ha simplificado y acelerado en gran medida.
A continuación se detallarán aspectos importantes a destacar de la implementación realizada.

4.1. Estructura clara y organizada

Desde el inicio del proyecto, César hizo hincapié en la importancia de contar con una estructura de código clara y organizada, que permitiera localizar rápidamente cualquier aspecto del mismo.

En este contexto, la implementación adoptada por César se basa en un archivo principal denominado `main.py`, el cual contiene la aplicación *FastAPI*. A este archivo se agregan todos los nodos terminales que se encuentran en la carpeta *endpoints*. Estos endpoints se encuentran distribuidos en varios archivos *.py*, clasificados de acuerdo con el aspecto de la *API* que manejan. Por ejemplo, se pueden encontrar archivos como *taquillas.py*, *productos.py*, entre otros.

Además de la organización de los endpoints, esta implementación también proporciona una forma sencilla de agregar todos los archivos necesarios para mostrar la interfaz de usuario (*frontend*). Esto implica que la estructura del proyecto permite incorporar de manera eficiente los archivos relevantes para la interfaz de usuario, facilitando así su desarrollo y mantenimiento.

A continuación se muestra un diagrama de la estructura utilizada:

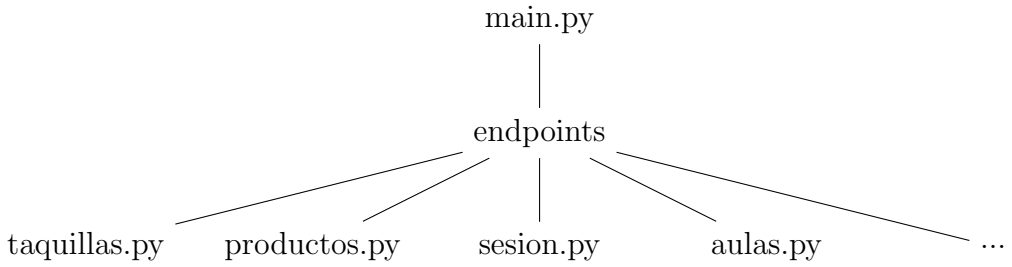

Figura 4.1: Diagrama de la estructura del proyecto

4.2. Manejo de errores

César ha implementado un sistema que permite lanzar errores que se le pueden mostrar al cliente de manera detallada; donde se le muestra el código de estado de la llamada *HTTP*, junto con un mensaje detallando la causa del problema. La mayor complicación ha sido implementar el sistema haciendo que en el momento en el que se quiera mandar dicha respuesta al cliente, se interrumpa toda la ejecución y se devuelva únicamente el error especificado.

4.3. Impresión interrumpida

Implementado por César, el método de depuración `printerrupt()` ofrece una solución eficiente para la depuración de nuestra *API*.

El método `printerrupt()` permite interrumpir rápidamente la ejecución del programa y devuelve al cliente una cadena de texto plano especificada como argumento. Esta función resulta especialmente útil en situaciones en las que es difícil determinar la causa de un error en un momento determinado. Al proporcionar una forma de interrumpir la ejecución y obtener información en forma de texto plano, este método simplifica el proceso de identificación y resolución de cualquier tipo de error o fallo (*bug*).

El uso de `printerrupt()` permite una depuración más eficiente al proporcionar una herramienta que ayuda a acotar el problema en cuestión. Al interrumpir la ejecución y enviar una respuesta al cliente, se facilita la identificación de puntos críticos en el código y se agiliza la solución de problemas.

4.4. Utilidades para la base de datos

César ha creado una serie de métodos que facilitan muy notoriamente el manejo de las conexiones a la base de datos donde se almacenan todos los datos.

Entre estos métodos destacan:

- `realizar_insercion`:
  Realiza la inserción de datos en una tabla de la base de datos, símplemente pasándole como argumentos el nombre de la tabla y un diccionario (*JSON*) cuyos campos serán las columnas de la tabla. El método inteligentemente realiza una serie de verificaciones previas para asegurar que todo funcionará correctamente y cuenta con un manejo de errores detallado. A continuación, se muestran los pasos que sigue este método:

  1. Establece una conexión a la base de datos utilizando una función llamada `get_connection()`.
  2. Verifica si la tabla especificada existe en la base de datos. Para ello, ejecuta una consulta de selección limitada a un solo registro en la tabla. Si la consulta arroja un error, se interpreta como que la tabla no existe y se genera un mensaje de error.
  3. Determina el nombre de la columna que actúa como clave primaria en la tabla. Esto se hace consultando el esquema de la base de datos y buscando la columna que tiene la restricción de "*PRIMARY KEY*".
  4. Verifica que la columna de la clave primaria no se haya proporcionado en el diccionario de datos o que su valor sea '*None*'. Si se proporciona o su valor no es '*None*',

se genera un mensaje de error. Los IDs de la base de datos implementada son del tipo Serial y se establecen automáticamente, por lo que no se deben enviar a la hora de realizar inserciones en la base de datos.

5. Obtiene los nombres de todas las columnas de la tabla desde el esquema de la base de datos.
6. Revisa que todas las columnas enviadas en el diccionario de datos existan en la tabla. Si alguna columna no existe, se genera un mensaje de error.
7. Elimina las columnas de la lista que no están presentes en el diccionario de datos. Esto asegura que solo se inserten los valores para las columnas proporcionadas.
8. Verifica que no falten campos requeridos que no pueden ser nulos en la base de datos. Esto se hace consultando los campos no nulos de la tabla y comparándolos con los campos proporcionados en el diccionario de datos. Si falta algún campo requerido, se genera un mensaje de error.
9. Agrega '*None*' como valor para las columnas que no están presentes en el diccionario de datos. Esto garantiza que se inserten '*NULL*' en esas columnas.
10. Construye una consulta de inserción automáticamente utilizando el nombre de la tabla y las columnas correspondientes. La columna de la clave primaria se omite en esta consulta por lo que se mencionó anteriormente.
11. Ejecuta la consulta de inserción en la base de datos utilizando una función llamada '`insertar_datos_conexion()`'. Si ocurre un error de integridad debido a una violación de clave primaria, se genera un mensaje de error. Realmente, no debería haberlo nunca, pero pese a ello, se maneja dicho error.
12. Obtiene el valor de la clave primaria del nuevo registro insertado utilizando una consulta que obtiene el valor actual de la secuencia de la clave primaria.
13. Cierra la conexión a la base de datos.
14. Retorna el valor de la clave primaria del nuevo registro insertado.

En resumen, este método se encarga de realizar la inserción de datos en una tabla de una base de datos creando inteligentemente una consulta procesada. Esto se realiza verificando la existencia de la tabla, la validez de los datos proporcionados, la presencia de campos requeridos y la integridad de la clave primaria. Además, se encarga de manejar los mensajes de error y retorna el valor de la clave primaria del nuevo registro.

- `realizar_actualizacion` Este método, de manera similar al anterior, se encarga de realizar una actualización en una tabla de una base de datos reciviéndo únicamente el nombre de la tabla, el id del registro y los un diccionario con aquellos campos que se deseen actualizar. Este método realiza la actualizacion tras verificar y manejar cualquier tipo de problema. A continuación, se detallan los pasos que sigue este método:

  1. Establece una conexión a la base de datos utilizando una función llamada '`get_connection()`'.
  2. Verifica si la tabla especificada existe en la base de datos. Para ello, ejecuta una consulta de selección limitada a un solo registro en la tabla. Si la consulta arroja un error, se interpreta como que la tabla no existe y se genera un mensaje de error.

3. Determina el nombre de la columna que actúa como clave primaria en la tabla. Esto se hace consultando el esquema de la base de datos y buscando la columna que tiene la restricción de "PRIMARY KEY".
4. Verifica que la columna de la clave primaria no se haya enviado en el diccionario de datos y que el parámetro 'id' no sea 'None'. Si se envía la columna o 'id' es 'None', se genera un mensaje de error.
5. Obtiene los nombres de todas las columnas de la tabla desde el esquema de la base de datos.
6. Verifica que exista un registro en la tabla con el valor de clave primaria especificado en el parámetro 'id'. Si no existe un registro con ese valor de clave primaria, se genera un mensaje de error.
7. Verifica que todas las columnas enviadas en el diccionario de datos existan en la tabla. Si alguna columna no existe, se genera un mensaje de error.
8. Verifica que los campos con valor 'None' puedan ser nulos en la base de datos. Esto se hace consultando los campos no nulos de la tabla y comparándolos con los campos y sus valores proporcionados en el diccionario de datos. Si se intenta asignar un valor 'None' a un campo no nulo, se genera un mensaje de error.
9. Construye la consulta de actualización y los valores a actualizar. Las columnas y sus respectivos valores se obtienen del diccionario de datos, excluyendo la columna de la clave primaria y las columnas que tienen un valor 'None'. La consulta de actualización se construye utilizando la sintaxis SQL adecuada.
10. Ejecuta la consulta de actualización en la base de datos utilizando una función llamada '`actualizar_datos_conexion()`'. Si ocurre un error de integridad debido a una violación de clave primaria, se genera un mensaje de error.
11. Cierra la conexión a la base de datos.
12. Retorna el valor del parámetro 'id'.

En resumen, este método se encarga de realizar una actualización en una tabla de una base de datos, verificando la existencia de la tabla, la validez de los datos proporcionados, la presencia de la clave primaria y las columnas, y la posibilidad de asignar valores '*None*' a campos nulos. Además, se encarga de manejar los mensajes de error y retorna el valor del parámetro 'id'.

4.5. Endpoints

Como se ha mencionado en apartados anteriores, los *endpoints* han sido organizados de manera categórica dentro de diversos ficheros. Nuestros endpoint se han distinguido en las siguientes categorías

- Sesión
  Dentro del fichero *sesion.py* se pueden encontrar los *endpoints* relacionados con el inicio de sesión y registro e usuarios en la plataforma.

- Páginas HTML
  Dentro del fichero *paginasHTML.py* se alojan todos los endpoints que retotnan las páginas *HTML* junto con su hoja de estilos *CSS* y código *JavaScript* asociado. Es una manera simple y organizada de conectar el *FrontEnd* al *BackEnd*.

- Taquillas
  En el fichero *taquillas.py* podemos encontrar todos los nodos terminales relacionados con la gestión de taquillas. Entre dichos endpoints encontramos:

  - `GET` *Lista Taquillas*
    Retorna un listado de todas las taquillas del sistema.
    Localizado en: `(host)/taquillas`
  - `GET` *Detalle taquilla*
    Retorna la información completa y detallada de una taquilla en específico.
    Para ello recibe como argumento el id de la taquilla deseada como path parameter.
    Localizado en: `(host)/taquillas/{id_taquilla}`
  - `POST` *Insertar taquilla*
    Permite insertar una nueva taquilla en el sistema con la información que se envíe en el body de la solicitud.
    Localizado en: `(host)/taquillas`

- Aulas
  En el fichero *aulas.py* podemos encontrar todos los nodos terminales relacionados con la gestión de todas las aulas. Entre dichos endpoints encontramos:

  - `GET` *Lista Aulas*
    Retorna un listado de todas las aulas de la Escuela Politécnica Superior. Permite también realizar una búsqueda filtrada según el piso del aula, su tipo (Laboratorio o Aula de teoría), ala (Norte, Sur, Este y Oeste)... Dicho filtro se realiza a partir de los query params recibidos.
    Localizado en: `(host)/aulas`
  - `GET` *Info aula*
    Retorna la información completa y detallada de un aula en concreto.
    Para ello recibe como argumento el id del aula deseada como path parameter.
    Localizado en: `(host)/aulas/{id_aula}`
  - `POST` *Insertar aula*
    Permite insertar una nueva aula en el sistema con la información que se envíe en el body de la solicitud.
    Localizado en: `(host)/aulas`
  - `PUT` *Actualizar aula*
    Permite actualizar los valores de un aula concreta del sistema con la información que se envíe en el body de la solicitud. Este endpoint es utilizado por parte del Arduino para actualizar valores como la temperatura en tiempo real.
    Localizado en: `(host)/aulas/{id_aula}`

- Cafetería