Desarrollo de Software - Práctica 2

Fernando Cuesta Bueno
Carlos Fernández Arrabal
Antonio Manuel García Mesa
17/04/2025

Índice

1. Introducción

Esta memoria corresponde a la parte grupal de la práctica 2 de la asignatura de Desarrollo de Software del Grado en Ingeniería Informática de la Universidad de Granada.

En esta práctica se van a realizar una serie de programas basados en las tecnologías de Flutter y Dart. Se tratatán diferentes objetivos tales como la adaptación de software, ampliación de la funcionalidad y aplicación de diferentes patrones arquitectónicos estudiados en la asignatura.

2. Ejercicio 3.1

En este ejercicio se pide que se realice reingeniería y se realicen una serie de mantenimientos a la actividad del patrón Filtros de Intercepcción de la práctica 1.

2.1. Mantenimiento Adaptativo

El mantenimiento adaptativo es el tipo de mantenimiento que se realiza para adaptar el software a un entorno cambiante. En este caso, se ha realizado un mantenimiento adaptativo al transformar el código de Java a Flutter/Dart, ya que la práctica 1 estaba implementada en Java y ahora se ha adaptado a un nuevo lenguaje y framework.

En la figura 1 se puede observar la pantalla de la aplicación en Flutter, donde se han adaptado los filtros de la práctica 1 a un nuevo entorno.

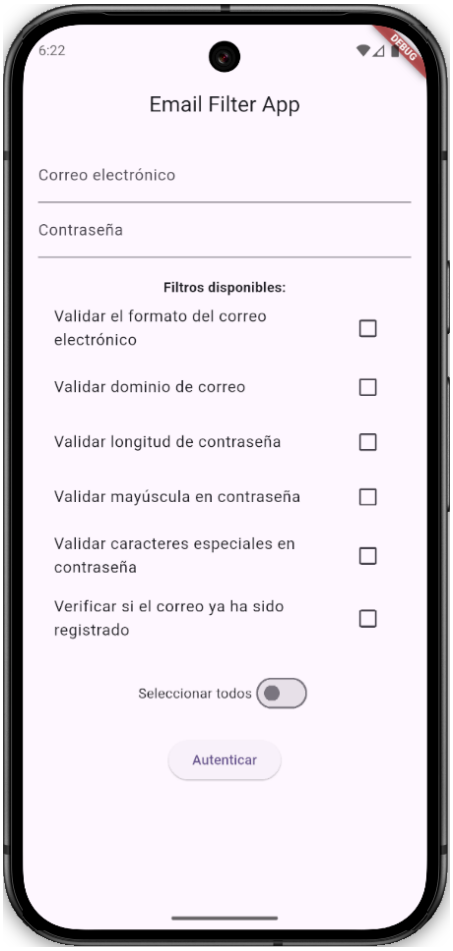

Figura 1: Pantalla de la aplicación Email Filter App en Flutter

2.2. Mantenimiento Perfectivo y Preventivo

El mantenimiento perfectivo es el tipo de mantenimiento que se realiza para mejorar la funcionalidad del software. En este caso, se ha realizado un mantenimiento perfectivo al agregar nuevas funcionalidades a la aplicación, como la posibilidad de seleccionar todos los filtros a la vez, así como la opción de quitar todos los filtros o seleccionar uno por uno.

El mantenimiento preventivo es el tipo de mantenimiento que se realiza para prevenir errores en el software. En este caso, se ha realizado un mantenimiento preventivo al agregar validaciones a los filtros, como la verificación de si el correo electrónico ya está registrado o no.

2.2.1. email_already_used_filter.dart

El archivo **email_already_used_filter.dart** implementa un nuevo filtro que verifica si el correo electrónico introducido por el usuario ha sido registrado anteriormente. Para ello implementa un método **execute()** que realiza lo siguiente: captura del **TextField** el email y comprueba si está presente en el campo privado **emailRegistados**, el cual es una lista con los emails que han sido registrados previemente. Si no está

presente, se añade a la lista, si está presente, se lanza una excepción con su notificación correspondiente.

2.3. Sistema de Notificaciones

El sistema de notificaciones es una funcionalidad que se ha agregado a la aplicación para informar al usuario sobre el estado de los filtros. En este caso, una vez que el usuario pulsa en el botón de **Autenticar**, se muestra una notificación emergente en la parte inferior de la pantalla, informando al usuario si los filtros han pasado correctamente o si ha habido algún problema con alguno de ellos.

En las figuras 2 y 3 se pueden observar las notificaciones que se muestran al usuario. En esta segunda figura, se muestra que el filtro de correo electrónico ya está registrado funciona correctamente y se muestra un mensaje de error al usuario.

Figura 2: Notificación de aceptación de filtros

Figura 3: Notificación de rechazo de filtros

2.4. auth_target.dart

El archivo **auth_target.dart** define la interfaz **AuthTarget** y una implementación de la misma llamada **PrintAuthTarget**. Su propósito es proporcionar un objetivo de autenticación que será ejecutado después de que todos los filtros aplicados a las credenciales (correo y contraseña) hayan sido validados correctamente.

- **AuthTarget**: interfaz que define el método **authenticate()** que recibe un objeto de tipo **Credentials** con el correo y la contraseña del usuario.
- **PrintAuthTarget**: implementación de la interfaz **AuthTarget** que simplemente imprime en la consola el correo y la contraseña del usuario. Esta implementación es útil para propósitos de depuración y demostración, pero en un entorno de producción se debería implementar una lógica de autenticación real.

2.5. credentials.dart

El archivo **credentials.dart** define la clase **Credentials** que representa las credenciales de un usuario, es decir, su correo electrónico y contraseña. Esta clase tiene dos propiedades: **email** y **password**, que son de tipo **String**.

Presenta un widget **CredentialsForm** que es un formulario que permite al usuario introducir su correo electrónico y contraseña. Este formulario utiliza dos **TextEditingController** para gestionar la entrada de texto del usuario.

El widget **CredentialsForm** es un **StatefulWidget**, lo que significa que tiene un estado interno que puede cambiar a lo largo del tiempo. Este estado se gestiona mediante la clase **CredentialsFormState**, que es donde se definen los controladores de texto y se gestionan los eventos de entrada del usuario.

- **emailController**: controlador para el campo de correo electrónico.
- **passwordController**: controlador para el campo de contraseña.

Ambos controladores tienen un **listener** que se activa cuando el usuario introduce texto en los campos correspondientes. Cuando el usuario introduce su correo electrónico y contraseña, estos valores se almacenan en los controladores de texto.

2.6. filter_manager.dart

En este archivo se implementa la clase **FilterManager** que se encarga de gestionar los filtros aplicados a los credenciales del usuario. Esta clase permite agregar filtros y aplicarlos a los credenciales del usuario introducidas.

2.7. email_filter_screen.dart

Dentro de este archivo se crea la pantalla que será mostrada al usuario. En este caso, se trata de una pantalla que permite al usuario introducir su correo electrónico y contraseña, así como seleccionar los filtros que desea aplicar a su correo electrónico. Para ello, se utilizan varios widgets de Flutter, como **TextField**, **CheckboxListTile** y **ElevatedButton**.

La pantalla se divide en varias secciones, cada una de las cuales se encarga de mostrar un elemento diferente.

- **TextField**: se utiliza para que el usuario introduzca su correo electrónico y contraseña.
- **CheckboxListTile**: se utiliza para mostrar los filtros disponibles y permitir al usuario seleccionar los que desea aplicar.
- **ElevatedButton**: se utiliza para autenticar al usuario y aplicar los filtros seleccionados.
- **SnackBar**: se utiliza para mostrar un mensaje al usuario informando si los filtros han pasado correctamente o si ha habido algún problema con alguno de ellos.

2.8. Diagrama UML

En la figura 4 se puede observar el diagrama UML de la aplicación. En este diagrama se pueden observar las diferentes clases que componen la aplicación, así como sus relaciones y dependencias.

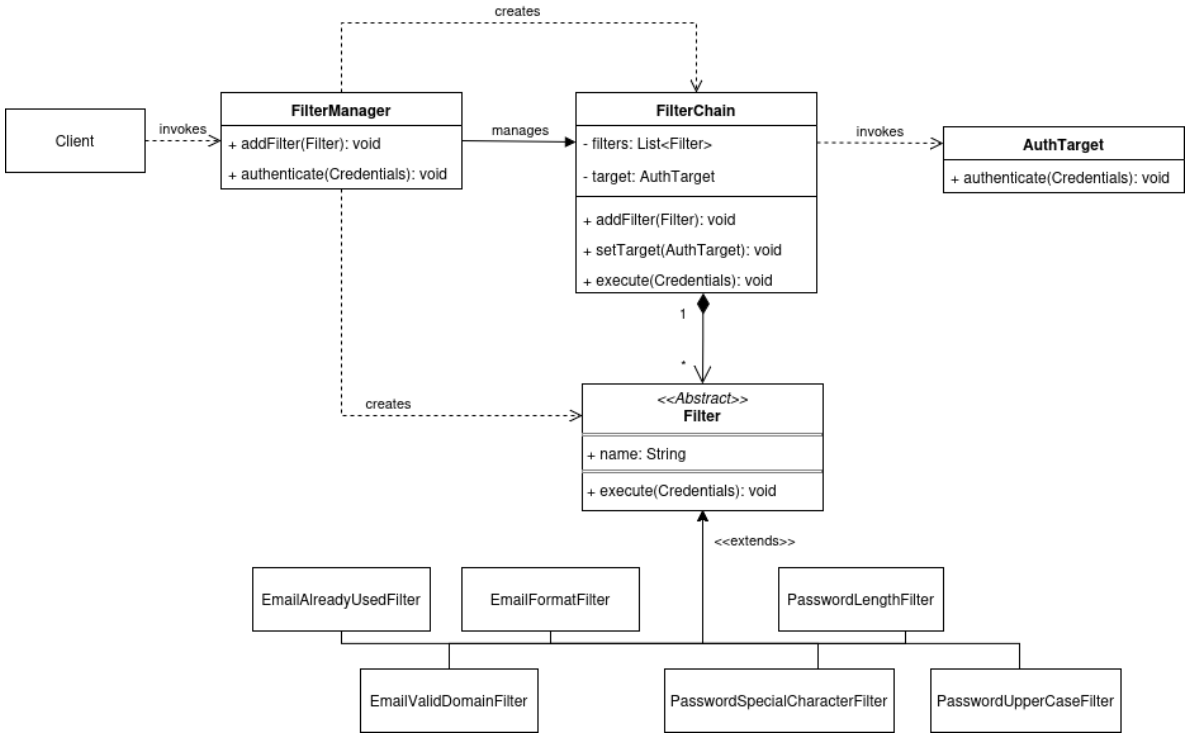

Figura 4: Diagrama UML de la aplicación Email Filter App

3. Ejercicio 3.2

Este ejercicio se presenta como opcional para la realización de la práctica 2. Se pide investigar cómo conectarse a la API de Hugging Face para poder hacer uso de ciertos LLMs. Hugging Face se trata de una web que actúa como repositorio de miles de modelos de Inteligencia Artificial. Para esta tarea, se require implementar una arquitectura basada en el patrón Strategy. El ejercicio se ha organizado en varios archivos con su correspondiente funcionalidad.

3.1. language_model_strategy.dart

En este archivo se define una interfaz abstracta donde se implementan dos elementos fundamentales:

- modelName: nombre del modelo en Hugging Face
- generateResponse(): método que genera una respuesta a partir del texto.

3.2. gpt2_strategy.dart

En este archivo se realiza una implementación de la interfaz anterior. En este se utiliza el modelo **gpt2** de Hugging Face. El modo de funcionamiento de esta clase es la siguiente:

- Se guarda una referencia a la clase HuggingFaceAPI, encargada de realizar la petición.
- Se define que el modelo a utilizar será **gpt2**.
- En el metodo **generateResponse()** se envía la petición mediante la API de Hugging Face junto con modelo seleccionado y el mensaje.

3.3. gpt3_strategy.dart

En este archivo se realiza la segunda implementación de la interfaz. La intención original fue la de utilizar el modelo **gpt3**, pero este no se encuentra accesible en la API de Hugging Face. Por lo que la siguiente idea fue la de usar un modelo similar, el **EleutherAI/gpt-neo-2.7B**. A la hora de hacer uso del mismo encontramos que el tamaño de este modelo era demasiado grande, tal y como se muestra en la figura 5. Por lo que finalmente se optó por el modelo **distilbert/distilgpt2**.

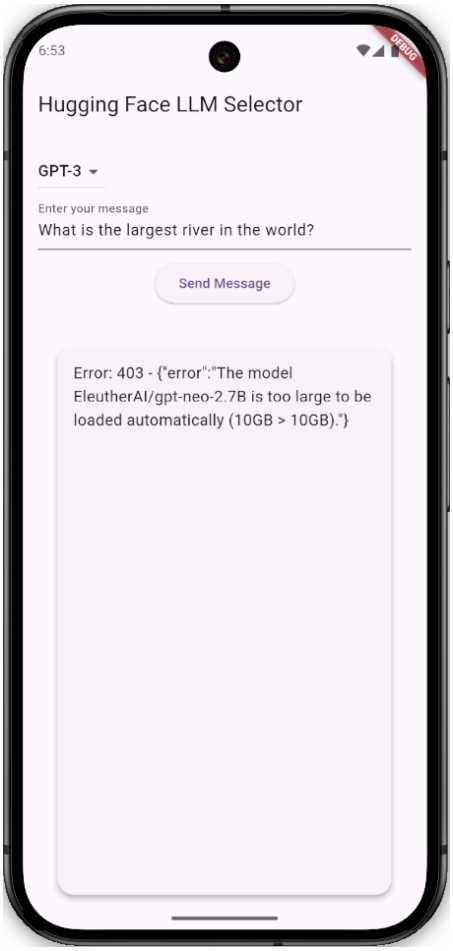

Figura 5: Problema de tamaño con el modelo **EleutherAI/gpt-neo-2.7B**

El modo de funcionamiento de esta clase es similar al de **gpt2_strategy.dart**, pero con el modelo **distilbert/distilgpt2**.

3.4. hugging_face_api.dart

Esta clase es la encargada de conectarse a la API de Hugging Face y enviar la solicitud deseada. Posteriormente, se recibirá la respuesta generada por el modelo elegido. Entrando más en detalle en el funcionamiento del **sendRequest()** tenemos varios pasos a seguir:

- Se definen las dos variables necesarias para hacer la solicitud: el token y la URL. Estas variables se encuentran en el archivo **secrets.dart**, el cual comentaremos más adelante.
- Creación de la URL a utilizar mediante la URL principal y el nombre del modelo seleccionado.
- Se añade el token de autenticación y se especifica que la respuesta enviada será de tipo JSON.
- Se define la petición y envío de la solicitud POST.

- Se procesa la respuesta obtenida

Esta clase trabaja con el tipo de datos **Future**, lo que significa que la respuesta no se obtiene de forma inmediata, sino que se espera a que la petición se complete. Por lo que el método **sendRequest()** devuelve un **Future¡String¿**.

También hace uso de la librería **http**, que es la encargada de realizar las peticiones HTTP. Esta librería se encuentra especificada en el archivo **pubspec.yaml**, con una versión **http: ˆ1.3.0**.

3.5. home_screen.dart

Esta clase define la interfaz visual que dispondrá el programa y que utilizará el usuario. En esta se definen varios elementos como el título, cuadros de texto, botones, menús desplegables.... La clase permite elegir el tipo de modelo a aplicar en la solicitud. Así como escribir el mensaje deseado en el cuadro de texto correspondiente. Por último, con el botón **Send Message**se llama a **generateResponse()**, que es el encargado de procesar la solicitud y obtener la respuesta, la cual se muestra en un cuadro de texto independiente.

Esta pantalla se puede observar en la figura 6. En la figura 7 se puede observar el funcionamiento de la aplicación mientras se está procesando la solicitud. En este caso, se muestra un círculo de carga que indica que la respuesta está siendo generada.

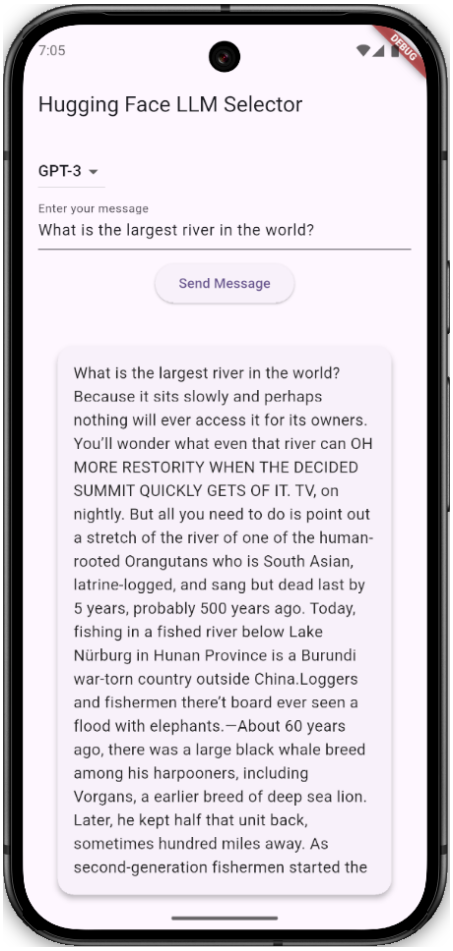

Figura 6: Pantalla de la aplicación Email Filter App en Flutter

Figura 7: Pantalla de la aplicación Email Filter App en Flutter

3.6. secrets.dart

En este archivo se definen las dos variables necesarias para realizar peticiones a la API. Se declaran el token de acceso y la URL base:

- **huggingFaceToken**: token de autenticación para acceder a la API.
- **huggingFaceModel**: URL base de la API.

Debido a la confidentialidad de estos datos, no se han incluido en el repositorio. En su lugar, se ha creado un archivo **secrets_template.dart** que contiene la misma estructura que el archivo **secrets.dart**, pero sin los datos reales. Este archivo sirve como plantilla para que cada usuario pueda crear su propio archivo **secrets.dart** con sus propios datos.

3.7. response_display.dart

Este archivo define un widget encargado de mostrar la respuesta generada por el modelo seleccionado en la solicitud enviada a la API. Para ello se utiliza el widget **build** al cual se le aplica decoración para que se vea bien visualmente y se definen las secciones de texto donde aparecerá la respuesta generada.

3.8. main.dart

Este archivo es el punto principal de ejecución de la app. Define el widget raíz, donde se configura la estructura y se carga el HomeScreen como pantalla principal.

3.9. Diagrama UML

En la figura 8 se puede observar el diagrama UML de la aplicación. En este diagrama se pueden observar las diferentes clases que componen la aplicación, así como sus relaciones y dependencias.

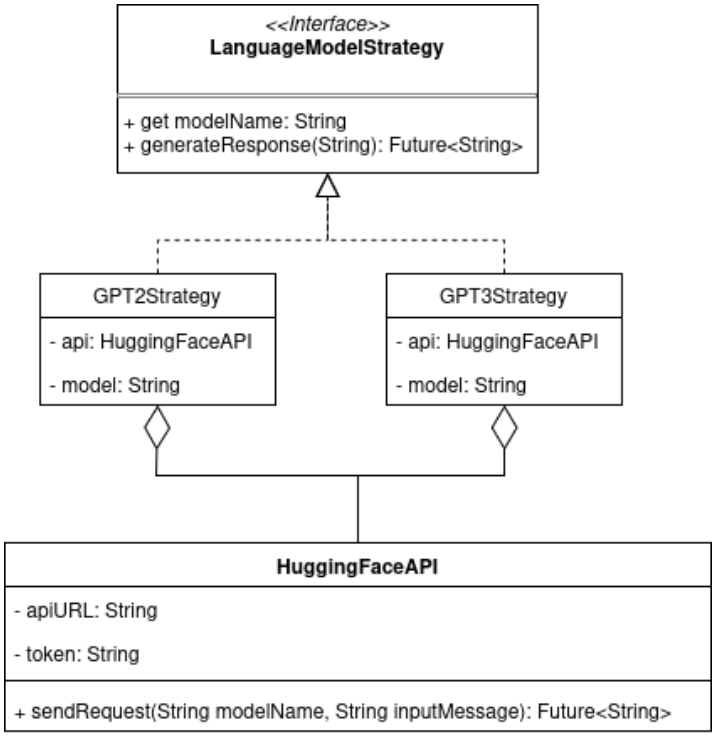

Figura 8: Diagrama UML de la aplicación Email Filter App